Travel Booking System : Quick Access to Travel Services

Project Description:

At Greenfield University, students and faculty often face difficulties in managing travel bookings for academic trips, conferences, and internships. Manual booking processes are slow, unorganized, and prone to miscommunication.

To address this, the university's Cloud Solutions Department developed the Travel Booking System—a virtual platform that enables students and staff to book, manage, and track travel arrangements seamlessly. Built using Flask for backend logic, AWS EC2 for hosting, DynamoDB for storing booking data, and AWS SNS for sending real-time travel notifications, this system modernizes and streamlines the entire travel management process.

Scenario 1: Streamlined Travel Booking for Students and Staff

With the Travel Booking System, users can register and log in securely. After logging in, students or faculty can easily access the travel booking interface to schedule transportation or accommodations for academic purposes. AWS EC2 ensures reliable performance, handling concurrent requests efficiently, even during peak usage. Flask manages user sessions and booking logic in real-time, while DynamoDB keeps track of all travel requests and bookings.

Scenario 2: Real-Time Travel Notifications

Whenever a new travel request is made or updated, AWS SNS instantly notifies the requester and the travel management team. For example, a faculty member books a flight for a conference—once submitted, Flask processes the booking and SNS sends confirmation emails to both the requester and the travel administrator. This ensures prompt communication and prevents delays or miscommunication.

Scenario 3: Easy Access to Travel Details

Users can log into the platform and view upcoming and past travel bookings. They can filter by date, destination, or status. The interface is intuitive, and backed by DynamoDB, which offers real-time data retrieval. AWS EC2 ensures the site remains available even under high demand, while Flask handles dynamic content rendering for each user.

.

Cloud Architecture Overview :-

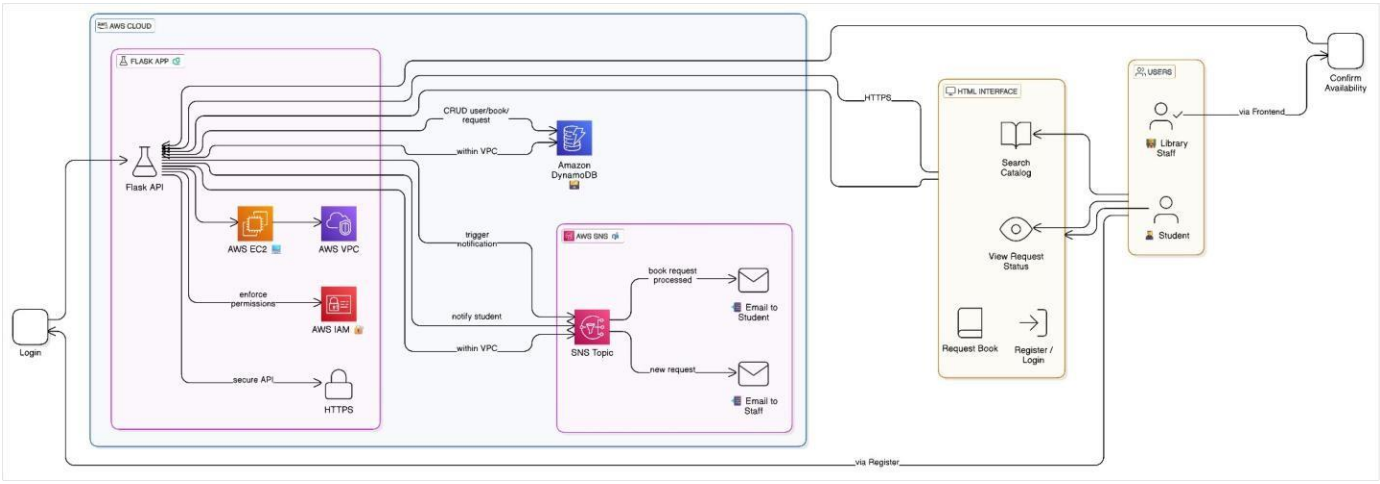
- **Frontend: HTML templates rendered by Flask (with routes for registration, login, booking, etc.)**
- **Backend: Flask (Python) application hosted on EC2**
- **Database: AWS DynamoDB (storing users and travel bookings)**

- **Notifications: AWS SNS (email alerts for booking confirmations)**
- **Deployment: Hosted on AWS EC2 Linux instance with Flask and Boto3**

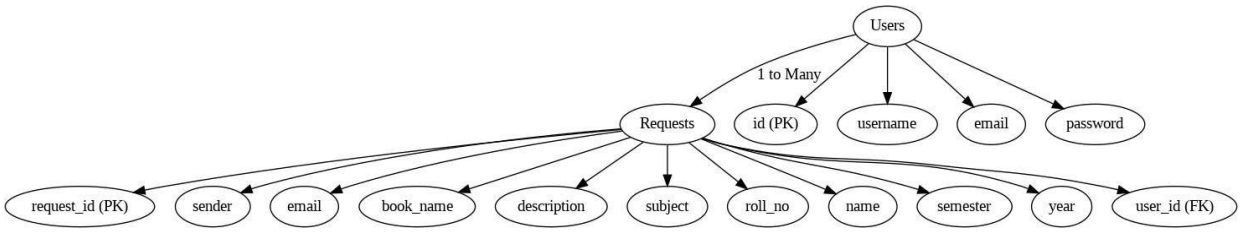
Cloud Architecture Overview :

- **Frontend: HTML templates rendered by Flask (with routes for registration, login, booking, etc.)**
- **Backend: Flask (Python) application hosted on EC2**
- **Database: AWS DynamoDB (storing users and travel bookings)**
- **Notifications: AWS SNS (email alerts for booking confirmations)**
- **Deployment: Hosted on AWS EC2 Linux instance with Flask and Boto3**

AWS ARCHITECTURE



Entity Relationship (ER)Diagram:



Pre-Requisites :-

1. **AWS Account**
2. **IAM Configuration**
3. **EC2 Instance Setup**
4. **DynamoDB Tables for Users and Bookings**
5. **SNS Topics for Travel Notifications**
6. **Flask App Code (uploaded via GitHub)**
7. **Git for version control**

Project WorkFlow:

Milestone 1: AWS Setup

- Create AWS account
- Configure IAM roles for EC2 instance

Milestone 2: DynamoDB Setup

- Create tables: Users (with Email as partition key), Bookings (with BookingID or Email as key)

Milestone 3: SNS Notifications

- Create topic: travel-booking-alerts
- Subscribe admin/staff and users via email

Milestone 4: Backend Development (Flask + Boto3)

- Flask routes: /register, /login, /book-travel, /view-bookings
- Store and retrieve data using DynamoDB
- Send SNS notifications on travel booking

Milestone 5: EC2 Hosting

- Launch EC2 instance
- Install Flask, Git, Boto3
- Clone GitHub repo and run the Flask server

Milestone 6: Testing and Deployment

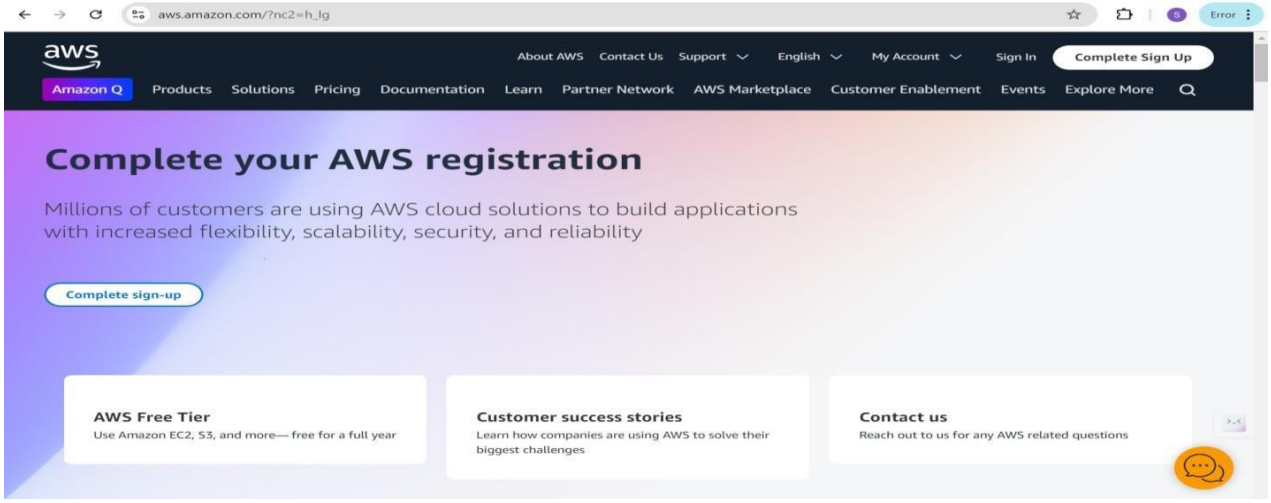
- **Test user registration, login, booking functionality, and email alerts**

1. Testing and Deployment

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

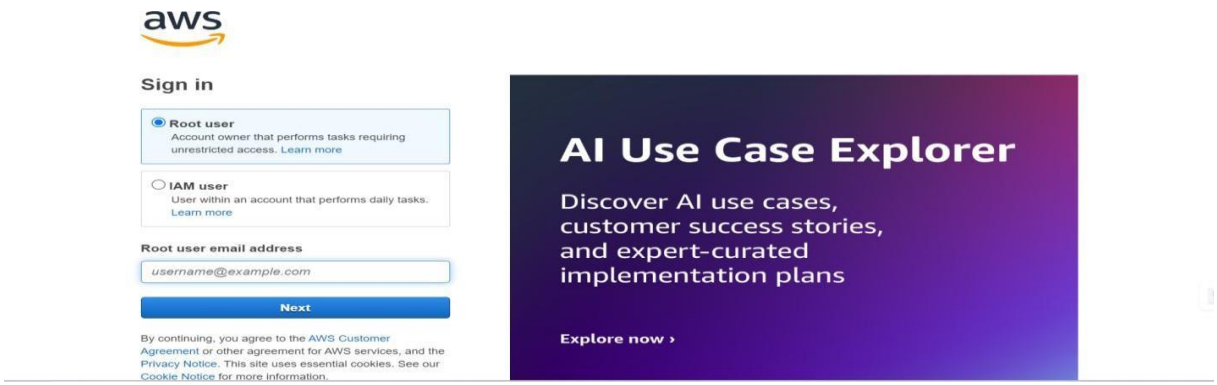
Milestone 1: AWS Account Setup and Login

- **Activity 1.1: Set up an AWS account if not already done.**
  - **Sign up for an AWS account and configure billing settings.**

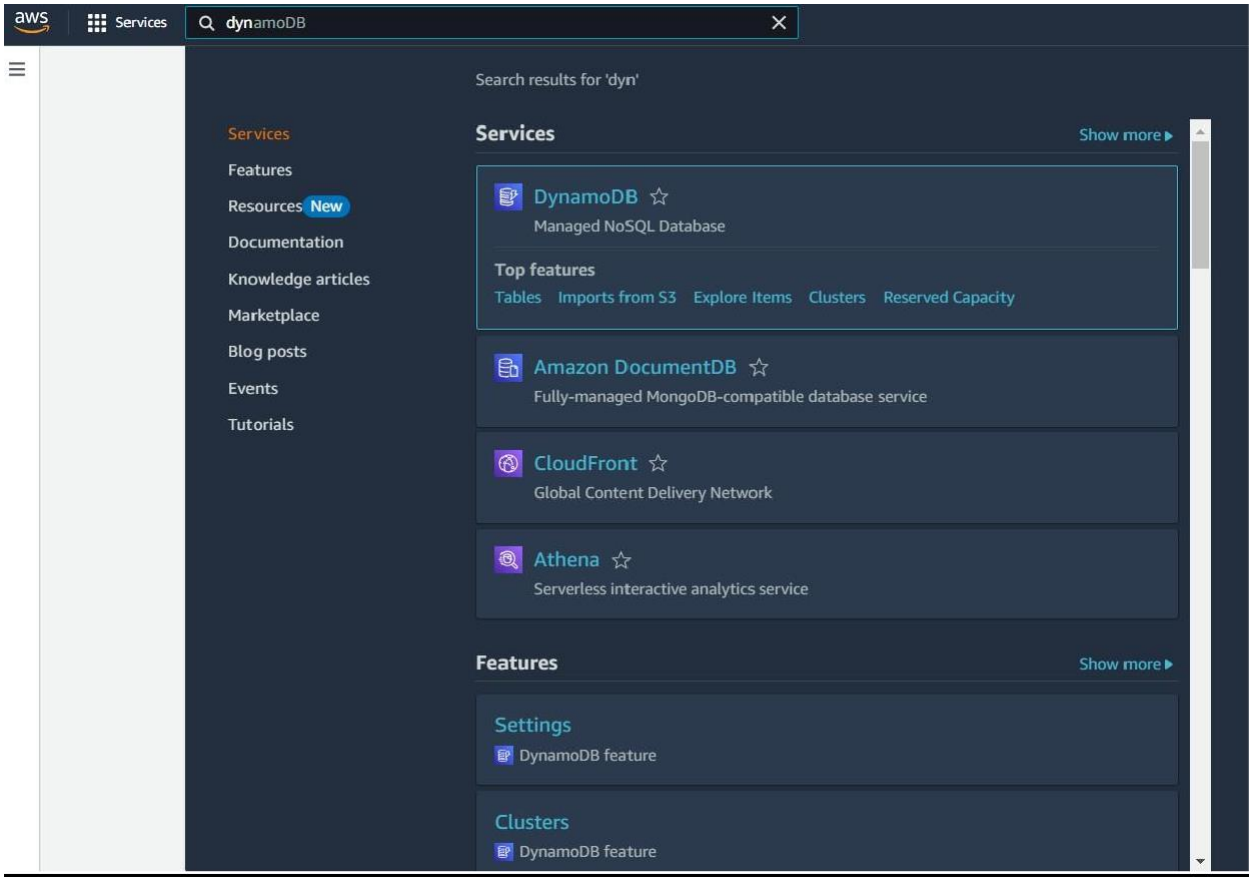


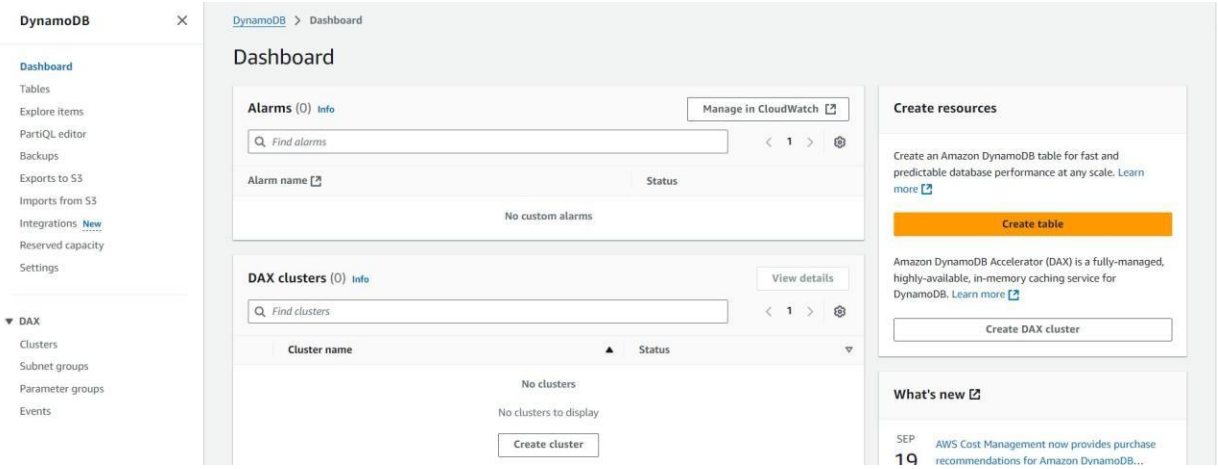
- **Activity 1.2: Log in to the AWS Management Console**

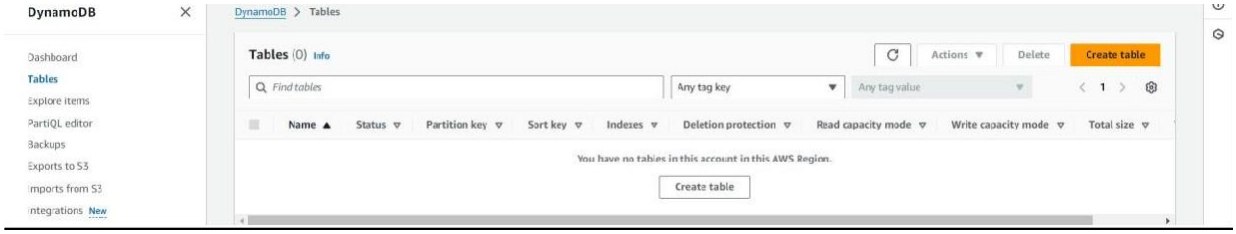
  - **After setting up your account, log in to the AWS Management Console.**



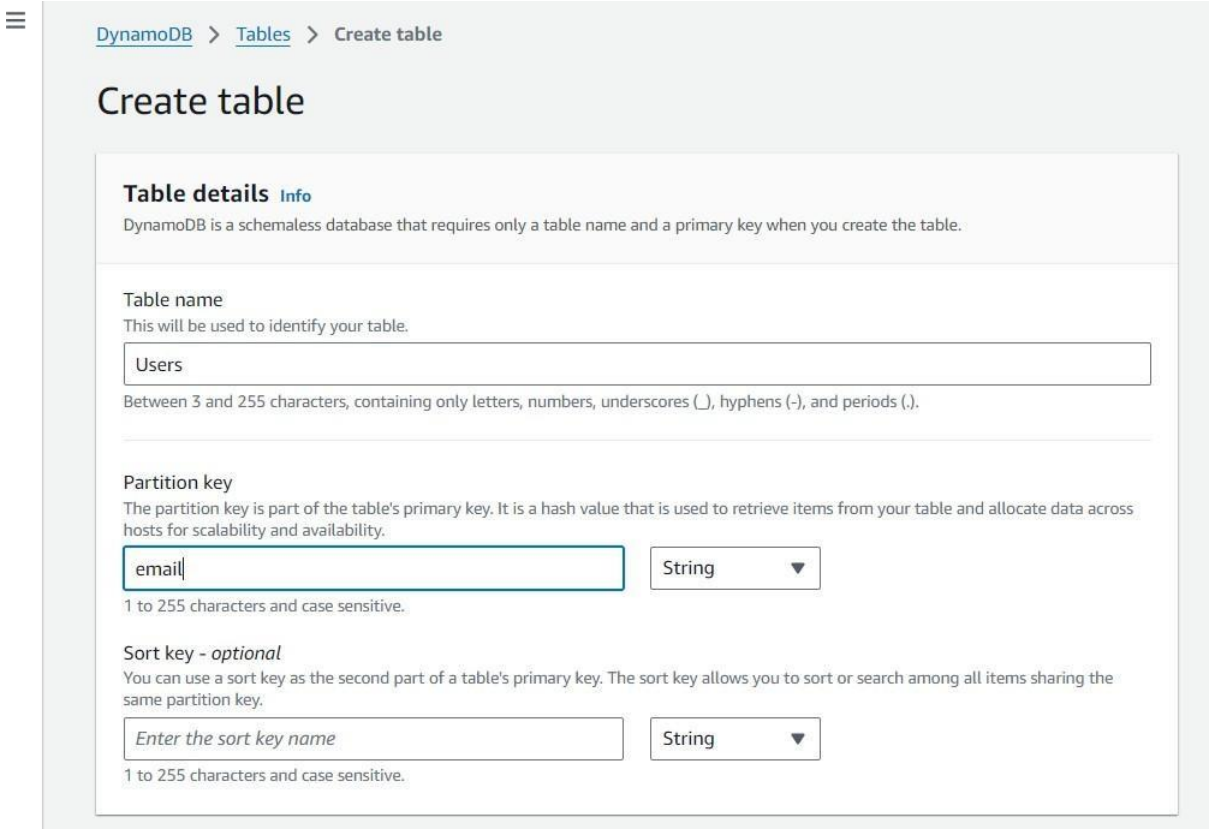
Milestone 2: DynamoDB Database Creation and Setup

- ## Activity 2.1:Navigate to the DynamoDB
  - **In the AWS Console, navigate to DynamoDB and click on create tables.**
  - 





- **Activity 2.2:Create a DynamoDB table for storing registration details and book requests.**
    - **Create Users table with partition key "Email" with type String and click on create tables.**



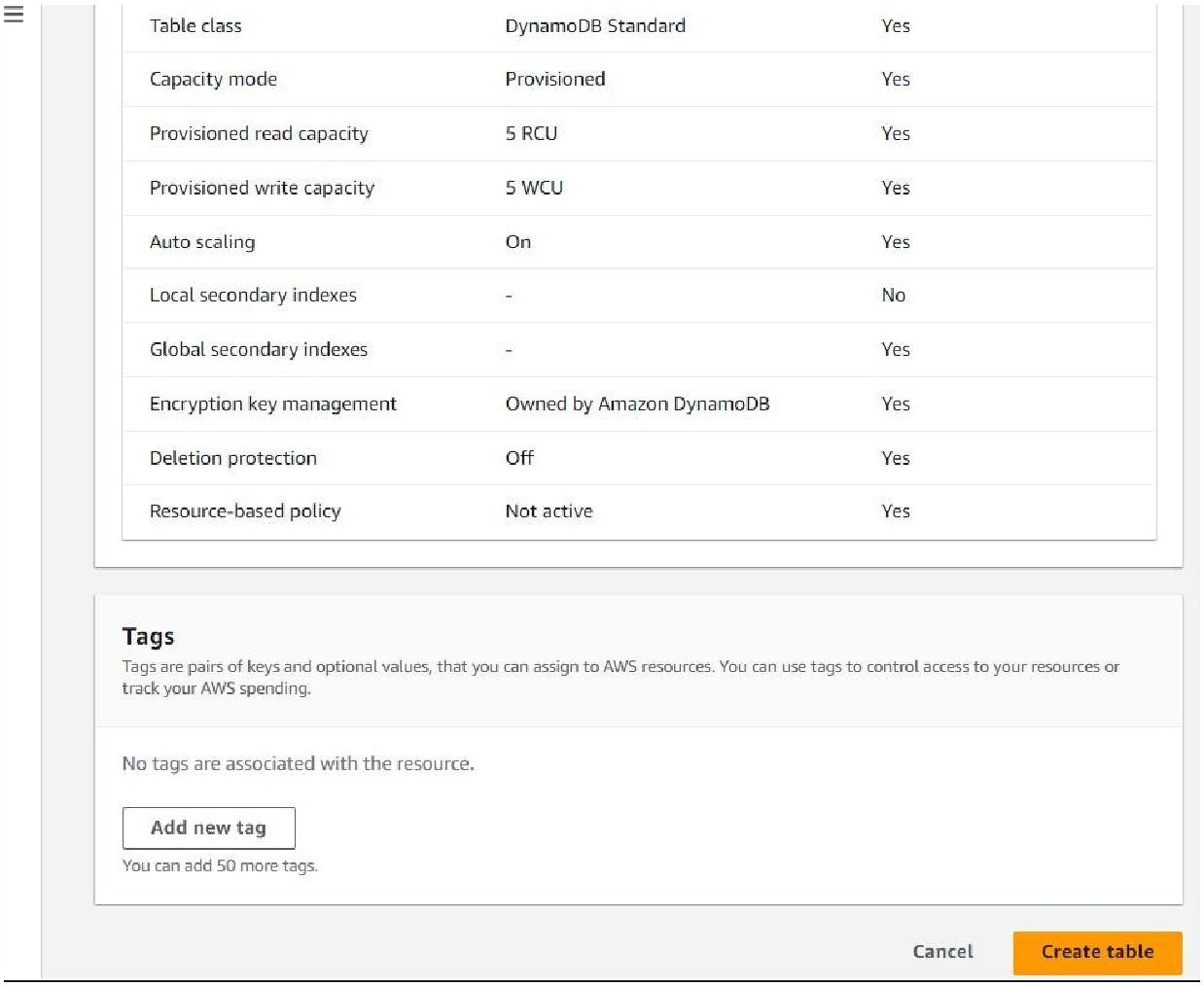
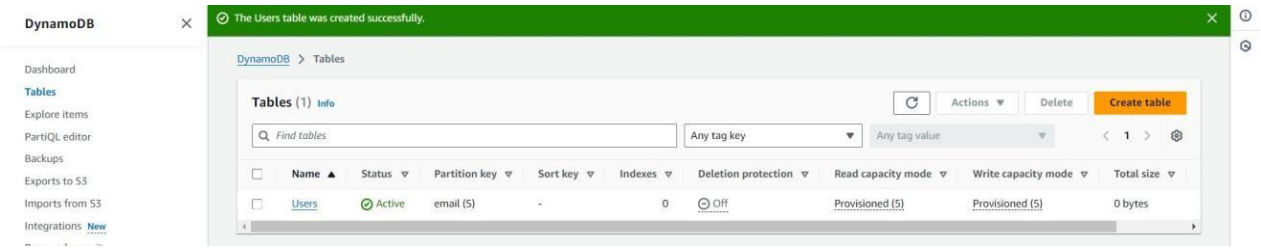


Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based policy	Not active	Yes



- **<u>Follow the same steps to create a requests table with Email as the primary key for book requests data.</u>**

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.

Requests

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.

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.

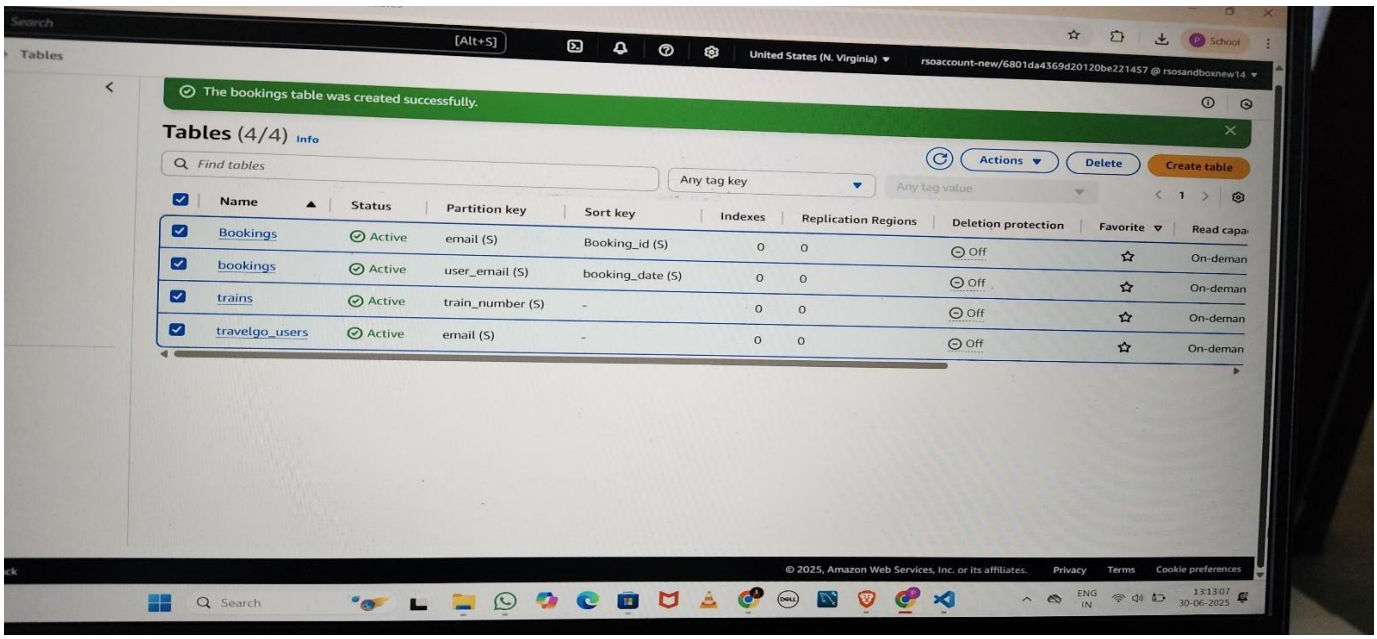
Enter the sort key name | String

1 to 255 characters and case sensitive.

Table settings

Default settings

Customize settings



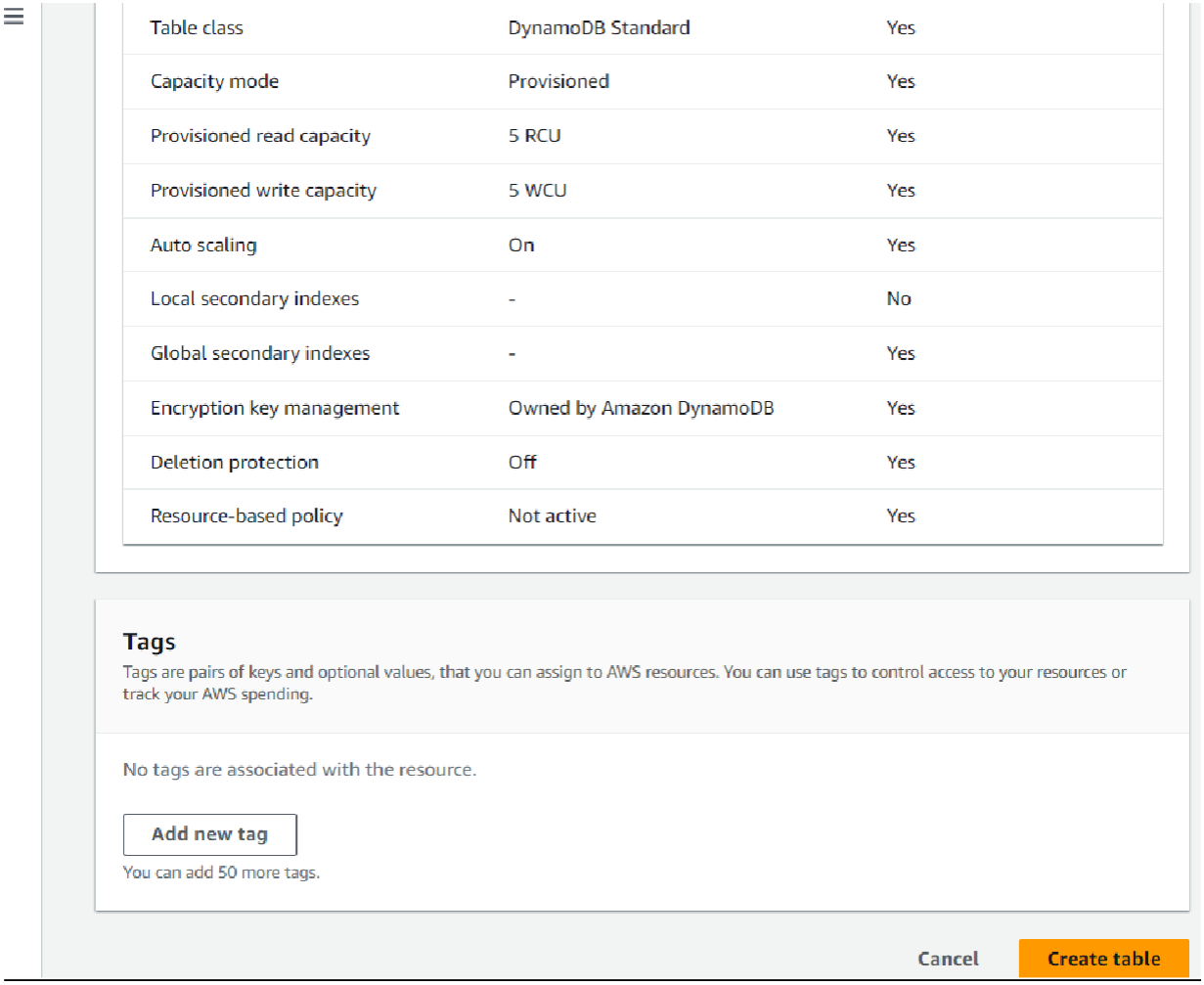
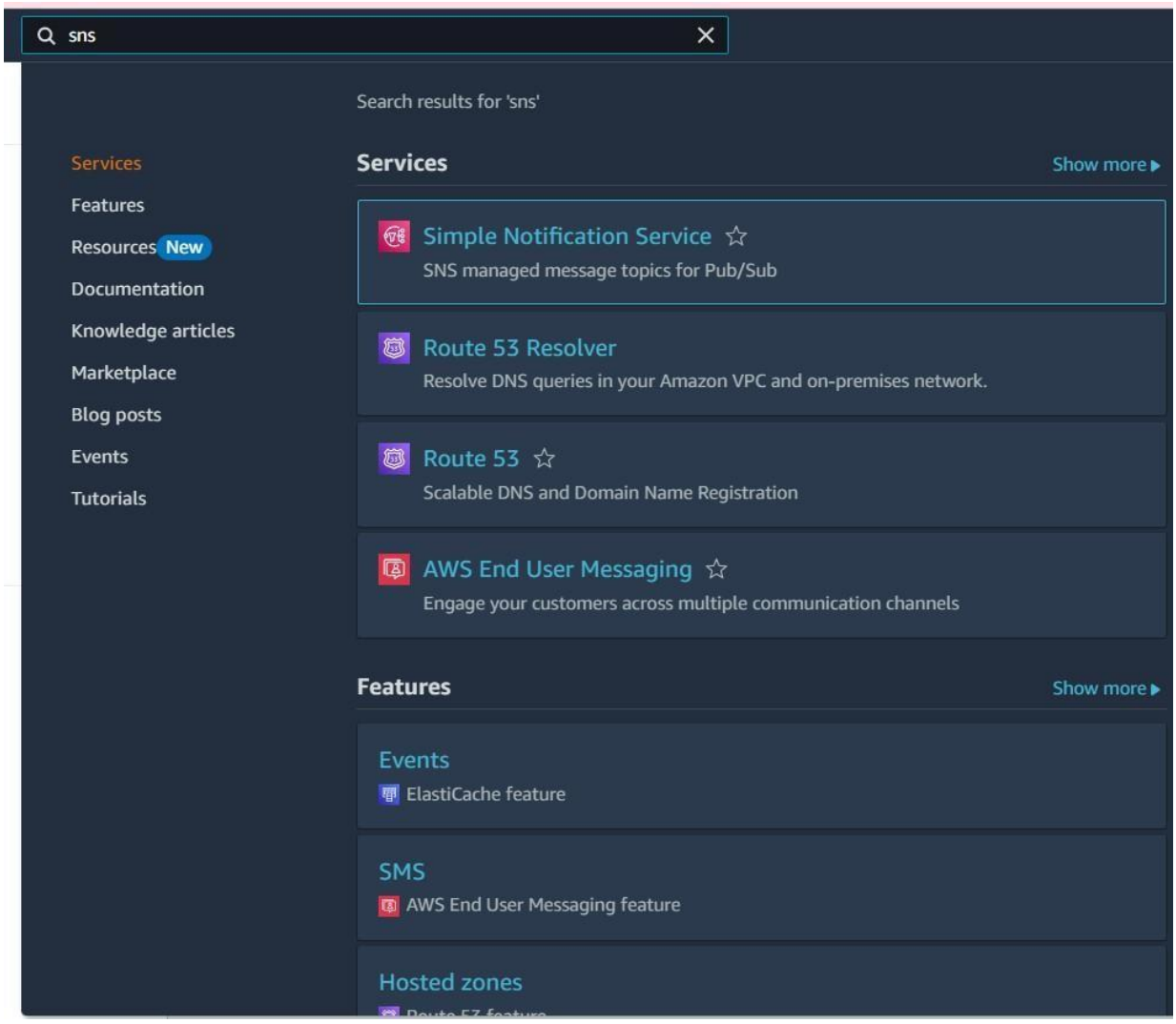


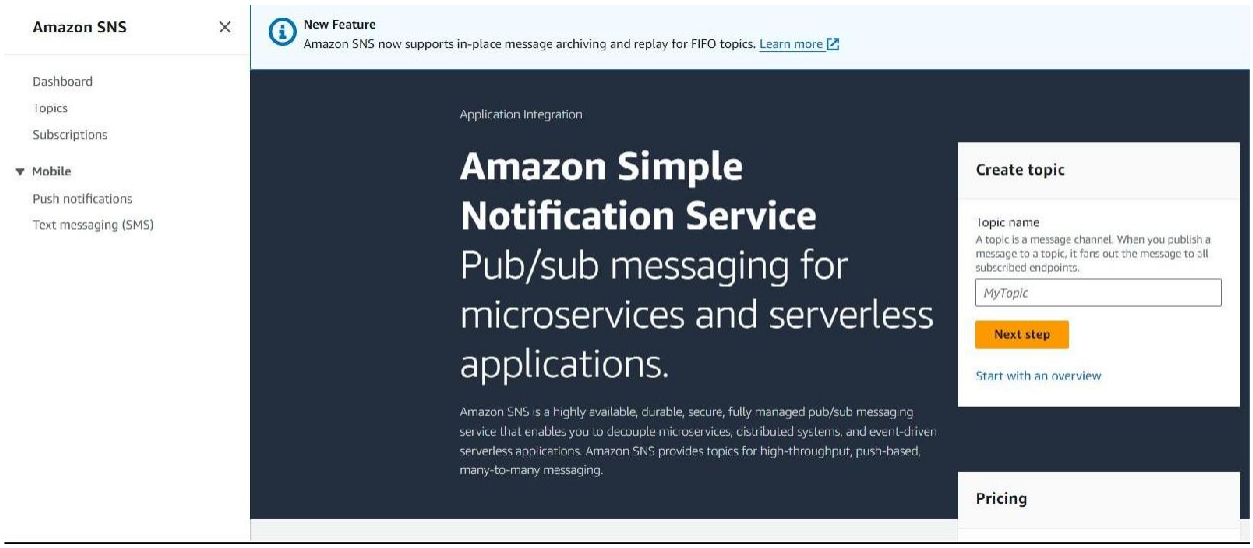
Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based policy	Not active	Yes

Milestone 3: SNS Notification Setup

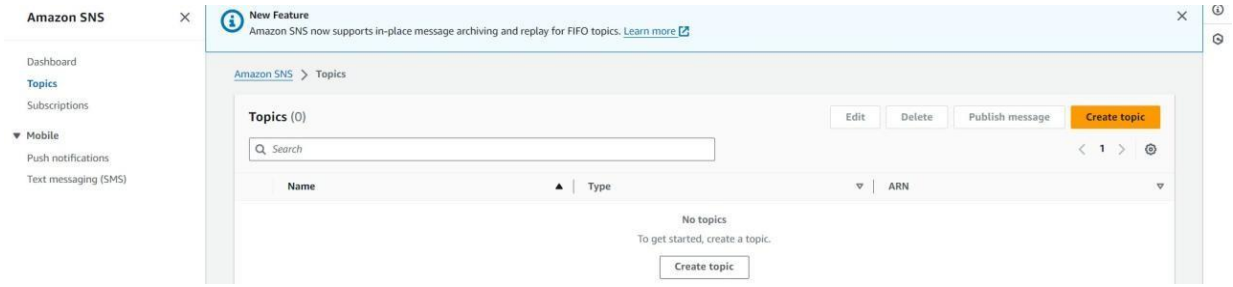
- **Activity 3.1: Create SNS topics for sending email notifications to users and library staff.**

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

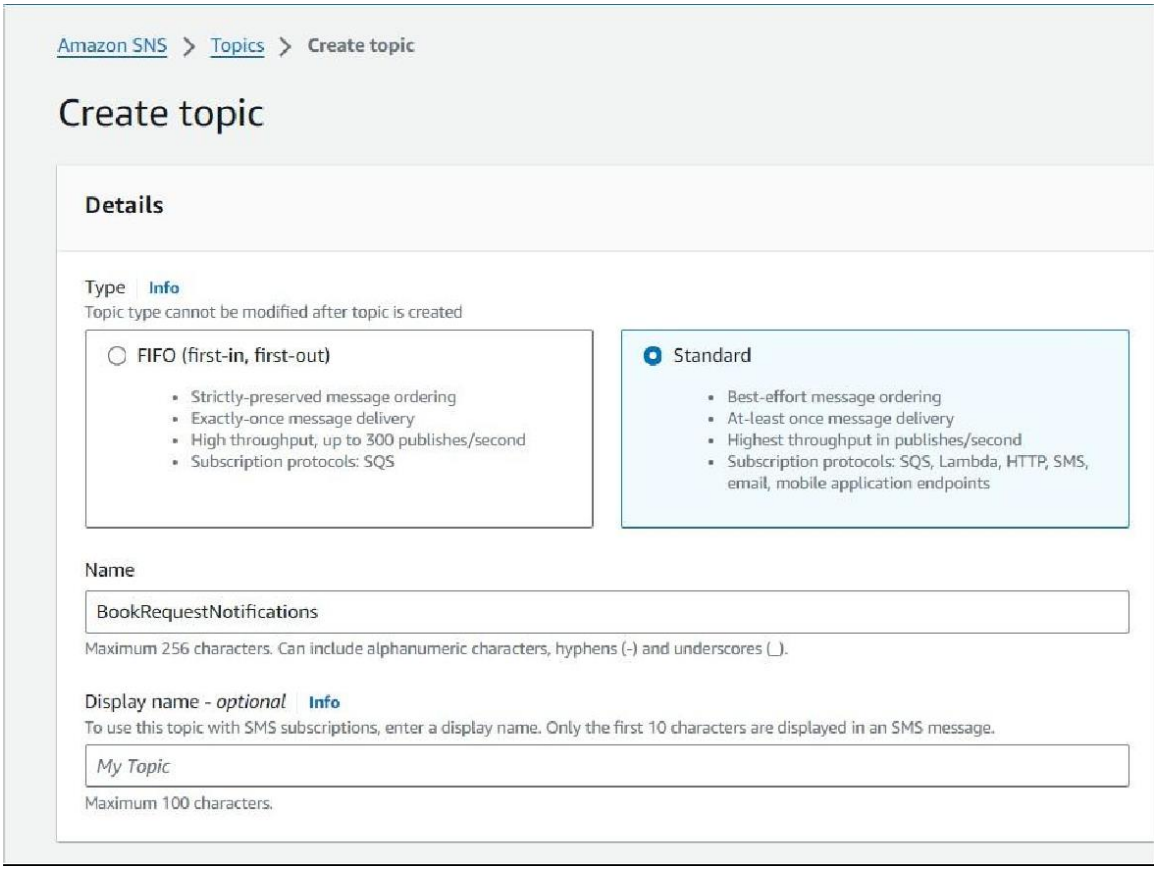


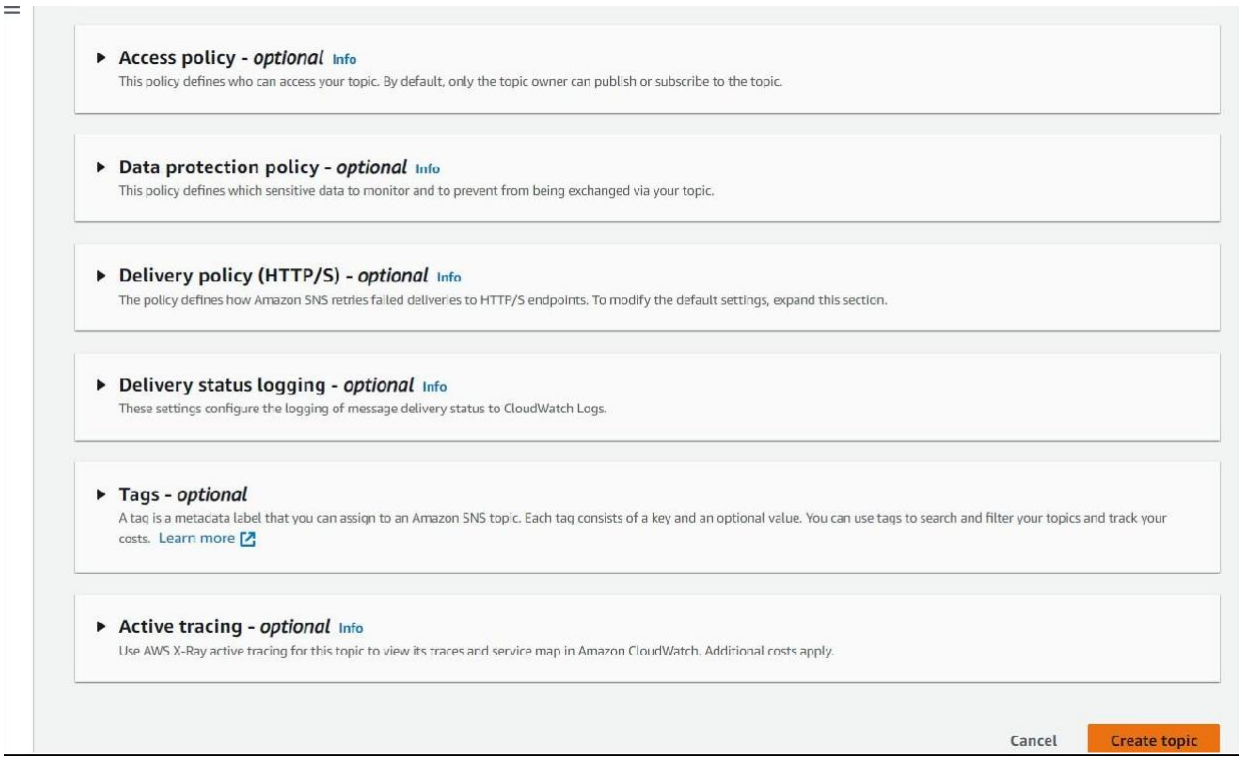


- **Click on Create Topic and choose a name for the topic.**

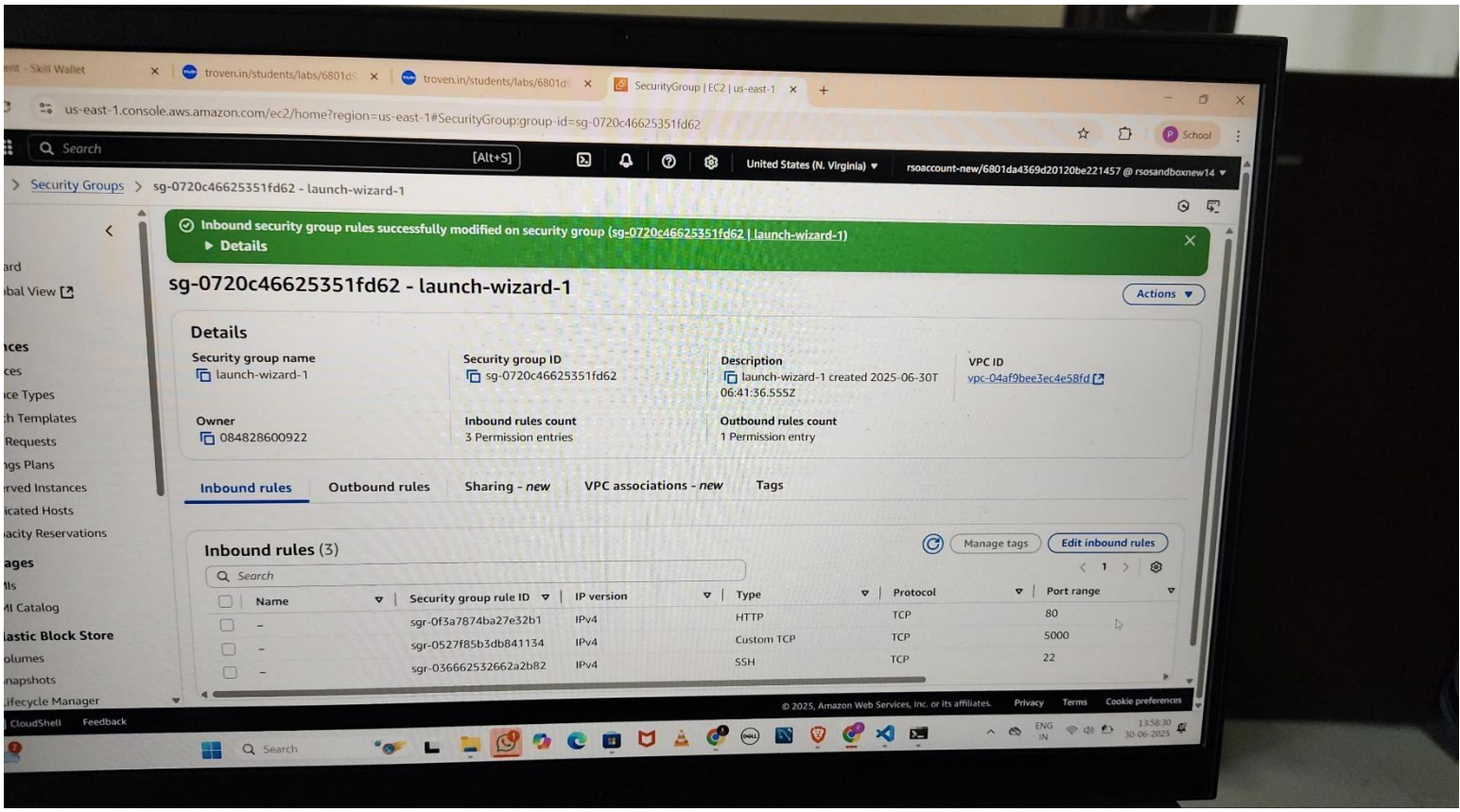


- **Choose Standard type for general notification use cases and Click on Create Topic.**


Amazon SNS > Topics > Create topic
Create topic
Details
Type Info
Topic type cannot be modified after topic is created
FIFO (first-in, first-out)
Strictly-preserved message ordering
Exactly-once message delivery
High throughput, up to 300 publishes/second
Subscription protocols: SQS
Standard
Best-effort message ordering
At-least once message delivery
Highest throughput in publishes/second
Subscription protocols: SQS, Lambda, HTTP, SMS, email, mobile application endpoints
Name
BookRequestNotifications
Maximum 256 characters. Can include alphanumeric characters, hyphens (-) and underscores (_).
Display name - optional Info
To use this topic with SMS subscriptions, enter a display name. Only the first 10 characters are displayed in an SMS message.
My Topic
Maximum 100 characters.

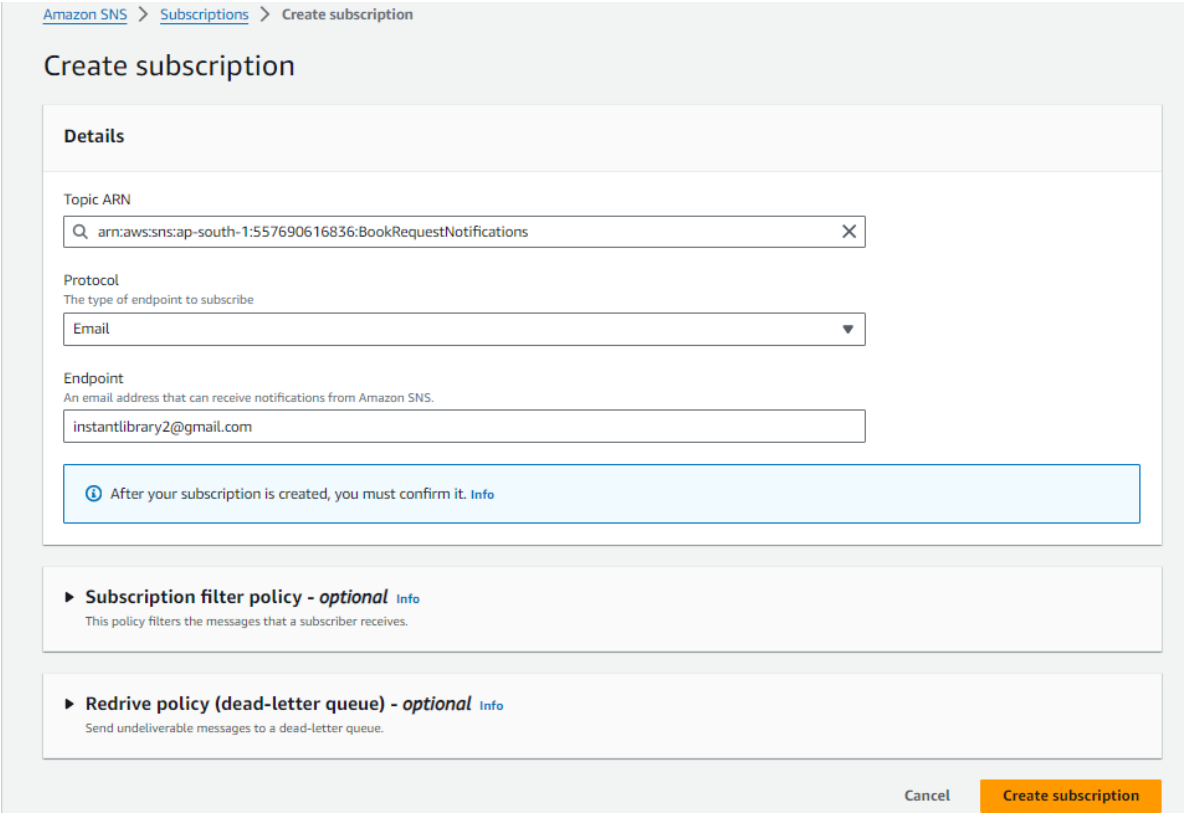


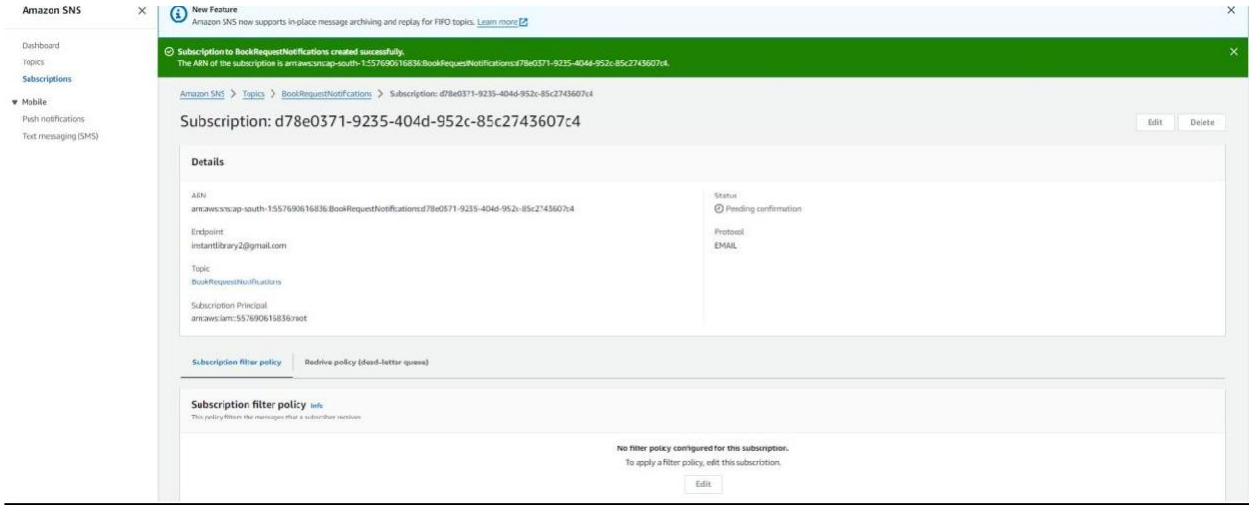
- **<u>Configure the SNS topic and note down the Topic ARN.</u>**



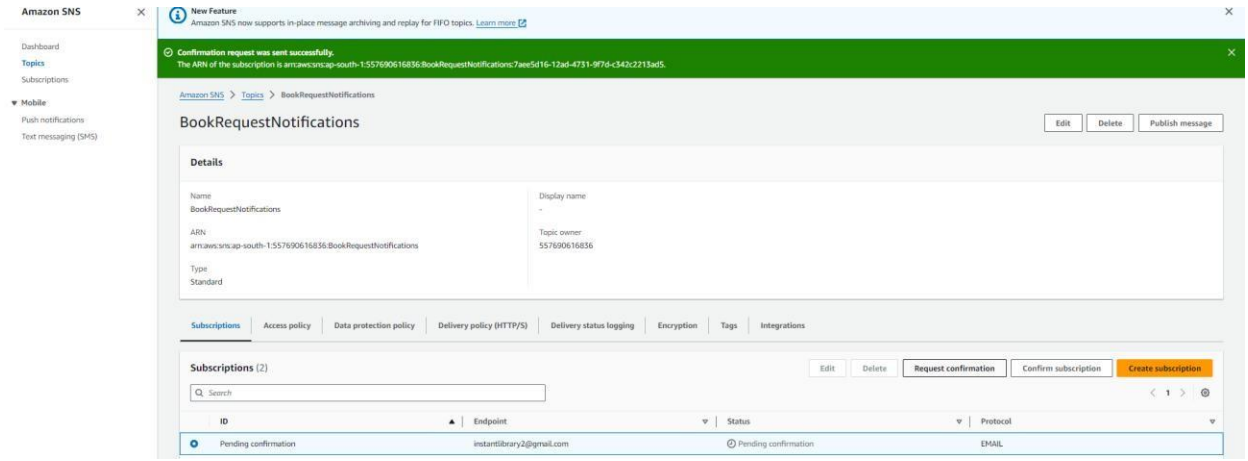
- **Activity 3.2: Subscribe users and staff to relevant SNS topics to receive real-time notifications when a book request is made.**

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

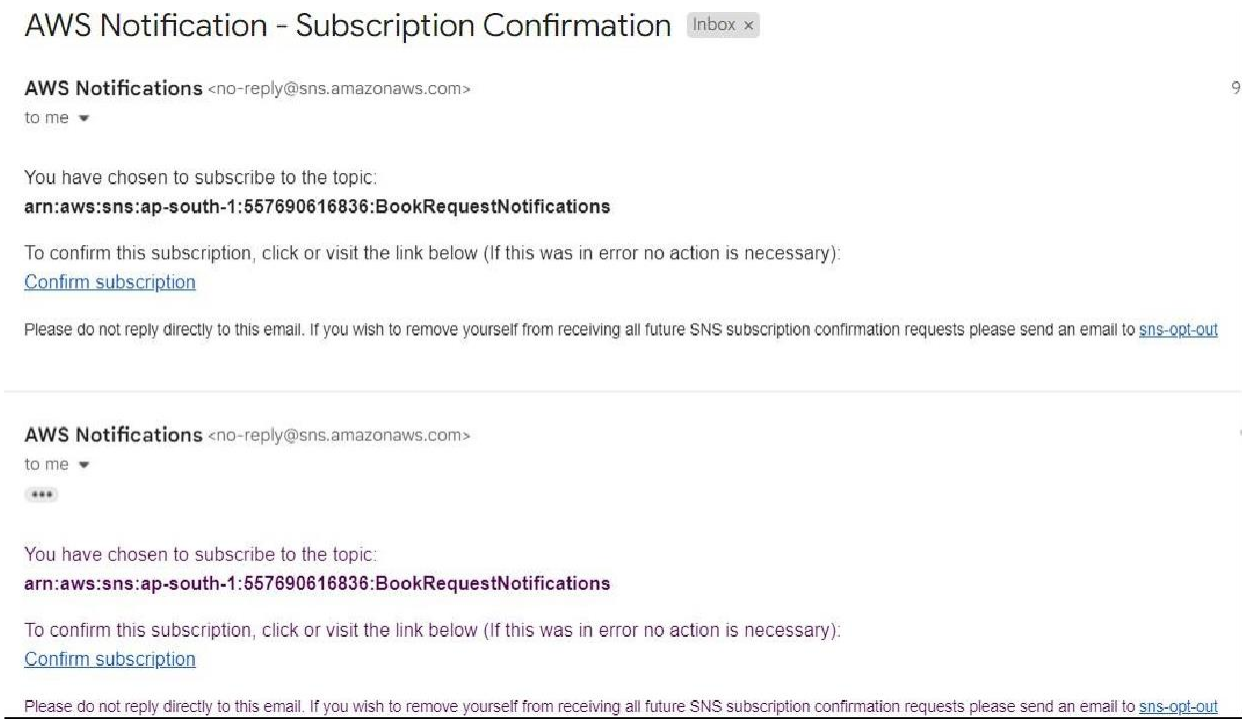


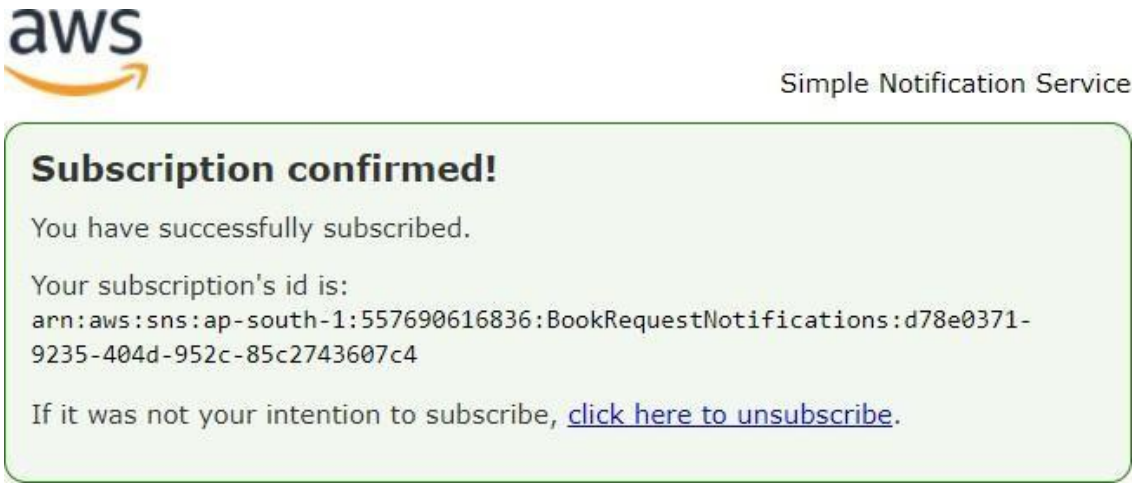


- **<u>After subscription request for the mail confirmation</u>**

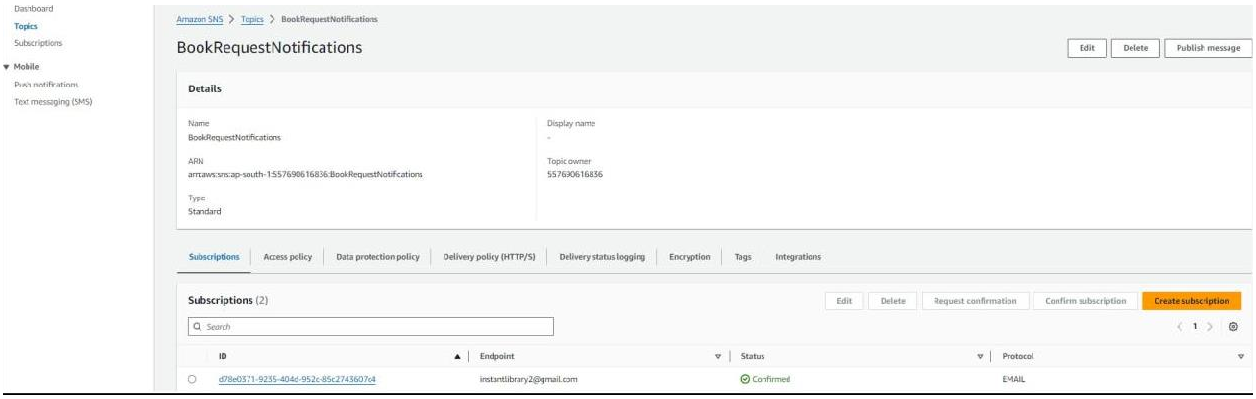


- **<u>Navigate to the subscribed Email account and Click on the confirm subscription in the AWS Notification- Subscription Confirmation mail.</u>**





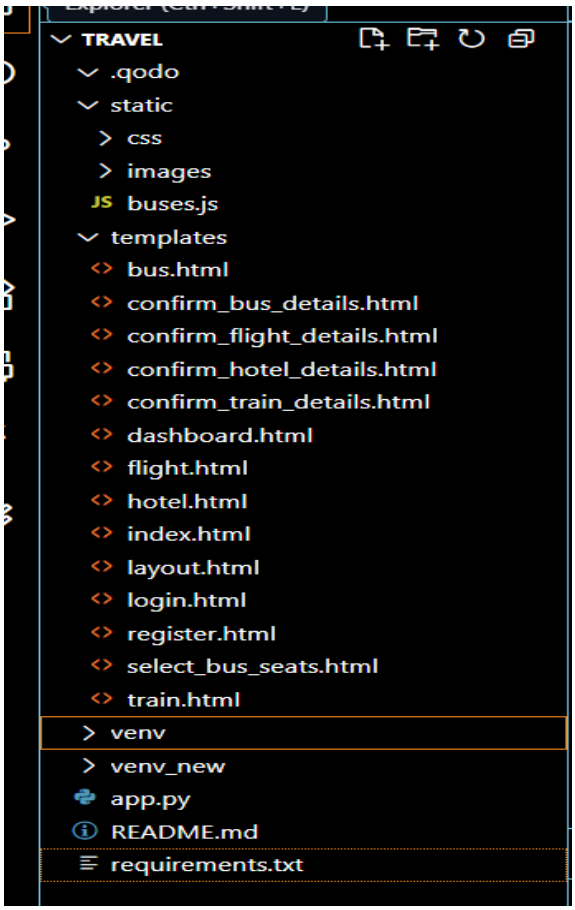
- **Successfully done with the SNS mail subscription and setup, now store the ARN link.**



Milestone 4:Backend Development and Application Setup

- ## Activity 4.1: Develop the backend using Flask

  - ### File Explorer Structure



Description: set up the INSTANT LIBRARY project with an app.py file, a static/ folder for assets, and a templates/ directory containing all required HTML pages like home, login, register, subject-specific pages (e.g., computer_science.html, data_science.html), and utility pages (e.g., request-form.html, statistics.html).

Description of the code :

- **Flask App Initialization**

```
from flask import Flask, render_template, request, redirect, url_for
import boto3
from boto3.dynamodb.conditions import Key
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from bcrypt import hashpw, gensalt, checkpw
```

Description: import essential libraries including Flask utilities for routing, Boto3 for DynamoDB operations, SMTP and email modules for sending mails, and Bcrypt for password hashing and verification

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

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

- **Dynamodb Setup:**

```
REGION = 'ap-south-1'  # Replace with your actual AWS region
dynamodb = boto3.resource('dynamodb', region_name=REGION)
sns_client = boto3.client('sns', region_name=REGION)

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

Description: initialize the DynamoDB resource for the ap-south-1 region and set up access to the Users and Requests tables for storing user details and book requests.

- **SNS Connection**

Description: Configure SNS to send notifications when a book request is submitted. Paste your stored ARN link in the sns_topic_arn space, along with the region_name where the SNS topic is created. Also, specify the chosen email service in SMTP_SERVER (e.g., Gmail, Yahoo, etc.) and enter the subscribed email in the SENDER_EMAIL section. Create an 'App password' for the email ID and store it in the SENDER_PASSWORD section.

- **Routes for Web Pages**
  - **Home Route:**

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

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

```python
SNS_TOPIC_ARN = 'arn:aws:sns:ap-south-1:353250843450:TravelGoBookingTopic'

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: {e}")


def send_sns_notification(subject, message):
    try:
        sns_client.publish(
            TopicArn=SNS_TOPIC_ARN,
            Subject=subject,
            Message=message
        )
    except Exception as e:
```

- **Register Route:**

```
# Routes
@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']
        existing = users_table.get_item(Key={'email': email})
        if 'Item' in existing:
            flash('Email already exists!', 'error')
            return render_template('register.html')
        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')
```

- **login Route (GET/POST):**

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

Logout and Dashboard :-

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

@app.route('/dashboard')
def dashboard():
    if 'email' not in session:
        return redirect(url_for('login'))
    user_email = session['email']
    response = bookings_table.query(
        KeyConditionExpression=Key('user_email').eq(user_email),
        ScanIndexForward=False
    )
    bookings = response.get('Items', [])
    for booking in bookings:
        if 'total_price' in booking:
            try:
                booking['total_price'] = float(booking['total_price'])
            except Exception:
                booking['total_price'] = 0.0
    return render_template('dashboard.html', username=user_email, bookings=bookings)
```

Train :-

```
@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'),
        'booking_type': 'train',
        'user_email': session['email'],
        'total_price': Decimal(request.args.get('price')) * int(request.args.get('persons'))
    }
```

```
response = bookings_table.query(
    IndexName='GSI_ItemDate',
    KeyConditionExpression=Key('item_id').eq(booking_details['item_id']) & Key('travel_date').eq(booking_details['travel_date'])
)

booked_seats = set()
for b in response.get('Items', []):
    if 'seats_display' in b:
        booked_seats.update(b['seats_display'].split(', '))

all_seats = [f"S{i}" for i in range(1, 101)]
available_seats = [seat for seat in all_seats if seat not in booked_seats]

if len(available_seats) < booking_details['num_persons']:
    flash("Not enough seats available.", "error")
    return redirect(url_for("train"))

session['pending_booking'] = booking_details
return render_template('confirm_train_details.html', booking=booking_details, available_seats=available_seats[:booking_details['num_persons']])
```

Buses:-

```
    response = bookings_table.query(
        IndexName='GSI_ItemDate',
        KeyConditionExpression=Key('item_id').eq(booking['item_id']) & Key('travel_date').eq(booking['travel_date'])
    )

    booked_seats = set()
    for b in response.get('Items', []):
        if 'seats_display' in b:
            booked_seats.update(b['seats_display'].split(', '))

    all_seats = [f"S{i}" for i in range(1, 41)]
    session['pending_booking'] = booking

    return render_template("select_bus_seats.html", booking=booking, booked_seats=booked_seats, all_seats=all_seats)

app.route('/final_confirm_bus_booking', methods=['POST'])
ef final_confirm_bus_booking():
    if 'email' not in session:
        return redirect(url_for('login'))

    booking = session.pop('pending_booking', None)
    selected_seats = request.form['selected_seats']

    if not booking or not selected_seats:
        flash("Booking failed! Missing data.", "error")
        return redirect(url_for("bus"))

    # Prevent double booking
    response = bookings_table.query(
        IndexName='GSI_ItemDate',
        KeyConditionExpression=Key('item_id').eq(booking['item_id']) & Key('travel_date').eq(booking['travel_date'])
    )
    existing = set()
    for b in response.get('Items', []):
        if 'seats_display' in b:
            existing.update(b['seats_display'].split(', '))

    selected = selected_seats.split(', ')
    if any(s in existing for s in selected):
        flash("One or more selected seats are already booked!", "error")
        return redirect(url_for("bus"))

    booking['seats_display'] = selected_seats
    booking['booking_id'] = str(uuid.uuid4())
    booking['booking_date'] = datetime.now().isoformat()

    bookings_table.put_item(Item=booking)

    send_sns_notification(
        subject="Bus Booking Confirmed",
        message=f"Your bus from {booking['source']} to {booking['destination']} on {booking['travel_date']} is confirmed.\nSeats: {booking['seats_display']}\nTotal: ₹{booking['total_price']}"
    )

    flash('Bus booking confirmed!', 'success')
    return redirect(url_for('dashboard'))
```

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

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

    booking_details = {
        'name': request.args.get('name'),
        'source': request.args.get('source'),
        'destination': request.args.get('destination'),
        'time': request.args.get('time'),
        'type': request.args.get('type'),
        '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('busId'),
        'booking_type': 'bus',
        'user_email': session['email'],
        'total_price': Decimal(request.args.get('price')) * int(request.args.get('persons'))
    }
    session['pending_booking'] = booking_details
    return render_template('confirm_bus_details.html', booking=booking_details)

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

    booking = {
        'name': request.args.get('name'),
        'source': request.args.get('source'),
        'destination': request.args.get('destination'),
        'time': request.args.get('time'),
        'type': request.args.get('type'),
        '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('busId'),
        'booking_type': 'bus',
        'user_email': session['email'],
        'total_price': Decimal(request.args.get('price')) * int(request.args.get('persons'))
    }
```

Flight :

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

@app.route('/confirm_flight_details')
def confirm_flight_details():
    booking = {
        'flight_id': request.args['flight_id'],
        'airline': request.args['airline'],
        'flight_number': request.args['flight_number'],
        'source': request.args['source'],
        'destination': request.args['destination'],
        'departure_time': request.args['departure'],
        'arrival_time': request.args['arrival'],
        'travel_date': request.args['date'],
        'num_persons': int(request.args['passengers']),
        'price_per_person': float(request.args['price']),
    }
    booking['total_price'] = booking['price_per_person'] * booking['num_persons']
    return render_template('confirm_flight_details.html', booking=booking)

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

    booking = {
        'booking_type': 'flight',
        'flight_id': request.form['flight_id'],
        'airline': request.form['airline'],
        'flight_number': request.form['flight_number'],
        'source': request.form['source'],
        'destination': request.form['destination'],
        'departure_time': request.form['departure_time'],
        'arrival_time': request.form['arrival_time'],
        'travel_date': request.form['travel_date'],
        'num_persons': int(request.form['num_persons']),
        'price_per_person': Decimal(request.form['price_per_person']),
        'total_price': Decimal(request.form['total_price']),
        'user_email': session['email'],
        'booking_date': datetime.now().isoformat(),
        'booking_id': str(uuid.uuid4())
    }

    bookings_table.put_item(Item=booking)

    # ✅ SNS for Flight
    send_sns_notification(
        subject="Flight Booking Confirmed",
        message=f"Your flight booking on {booking['travel_date']} from {booking['source']} to {booking['destination']} with {booking['airline']} is confirmed.\nTotal: ₹{booking['total_price']}"
    )

    flash('Flight booking confirmed successfully!', 'success')
    return redirect(url_for('dashboard'))
```

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

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

    booking = {
        'name': request.args.get('name'),
        'location': request.args.get('location'),
        'checkin_date': request.args.get('checkin'),
        'checkout_date': request.args.get('checkout'),
        'num_rooms': int(request.args.get('rooms')),
        'num_guests': int(request.args.get('guests')),
        'price_per_night': Decimal(request.args.get('price')),
        'rating': int(request.args.get('rating'))
    }

    ci = datetime.fromisoformat(booking['checkin_date'])
    co = datetime.fromisoformat(booking['checkout_date'])
    nights = (co - ci).days
    booking['nights'] = nights
    booking['total_price'] = booking['price_per_night'] * booking['num_rooms'] * nights

    return render_template('confirm_hotel_details.html', booking=booking)

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

    booking = {
        'booking_type': 'hotel',
        'name': request.form['hotel_name'],
        'location': request.form['location'],
        'checkin_date': request.form['checkin'],
        'checkout_date': request.form['checkout'],
        'num_rooms': int(request.form['rooms']),
        'num_guests': int(request.form['guests']),
        'price_per_night': Decimal(request.form['price']),
        'rating': int(request.form['rating']),
        'user_email': session['email'],
        'booking_date': datetime.now().isoformat(),
        'booking_id': str(uuid.uuid4())
    }

    ci = datetime.fromisoformat(booking['checkin_date'])
    co = datetime.fromisoformat(booking['checkout_date'])
    nights = (co - ci).days
    booking['total_price'] = booking['price_per_night'] * booking['num_rooms'] * nights

    bookings_table.put_item(Item=booking)
```

Response :-

```
# ✅ SNS for Flight
send_sns_notification(
    subject="Flight Booking Confirmed",
    message=f"Your flight booking on {booking['travel_date']} from {booking['source']} to {booking['destination']} with {booking['airline']} is confirmed.\nTotal: ₹{booking['total_price']}"
)

flash('Flight booking confirmed successfully!', 'success')
return redirect(url_for('dashboard'))
```

```
# ✅ SNS for Hotel
send_sns_notification(
    subject="Hotel Booking Confirmed",
    message=f"Hotel booking at {booking['name']} in {booking['location']} from {booking['checkin_date']} to {booking['checkout_date']} is confirmed.\nTotal: ₹{booking['total_price']}"
)

flash('Hotel booking confirmed successfully!', 'success')
return redirect(url_for('dashboard'))

op.route('/cancel_booking', methods=['POST'])
f cancel_booking():
if 'email' not in session:
    return redirect(url_for('login'))

booking_id = request.form.get('booking_id')
user_email = session['email']

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

try:
    bookings_table.delete_item(
        Key={'user_email': user_email, 'booking_date': request.form.get('booking_date')}
    )
    flash(f"Booking cancelled successfully!", 'success')
except Exception as e:
    flash(f"Failed to cancel booking: {str(e)}", 'error')

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

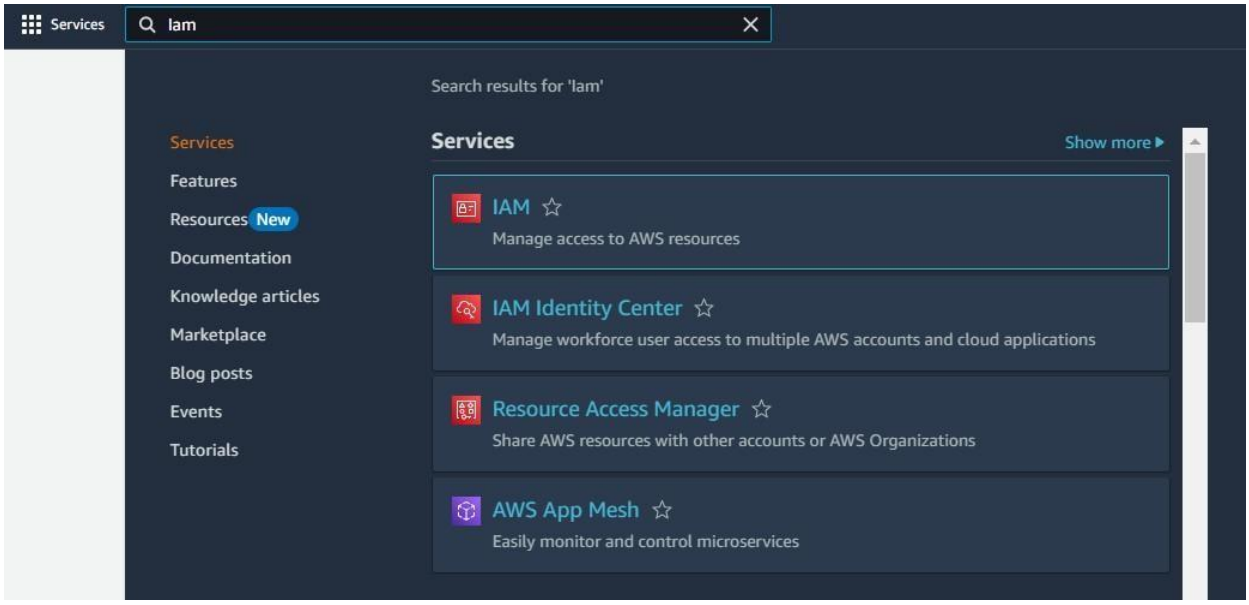
Deployment Code:

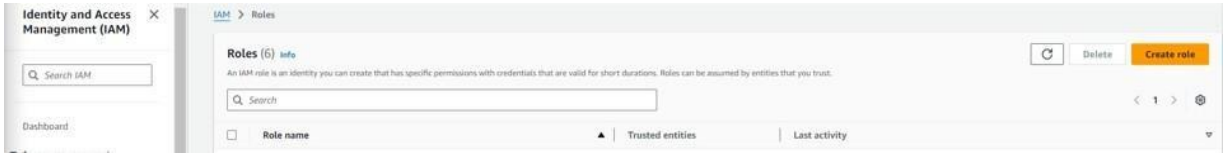
```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

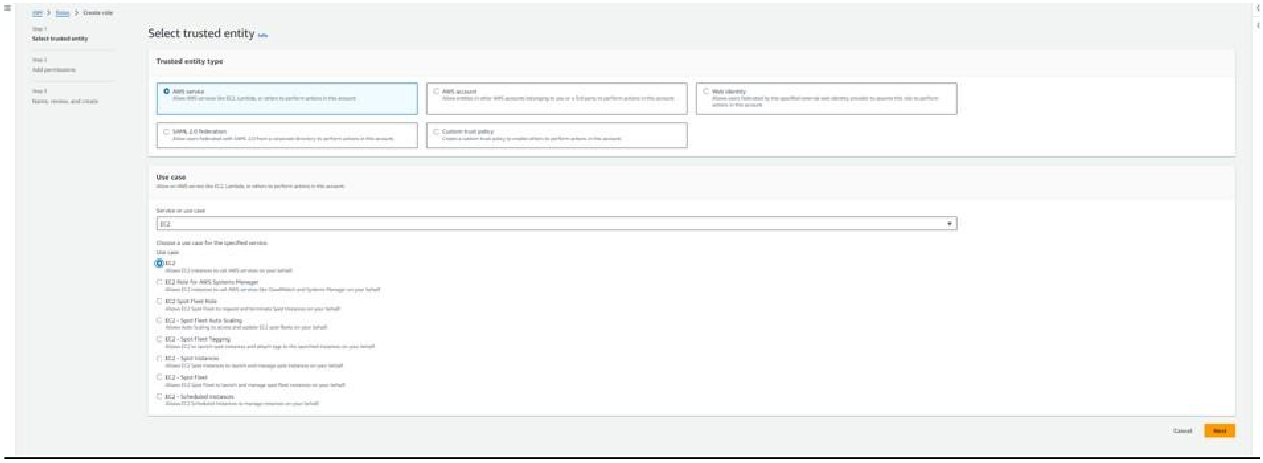
Description: start the Flask server to listen on all network interfaces (`0.0.0.0`) at port `80` with debug mode enabled for development and testing.

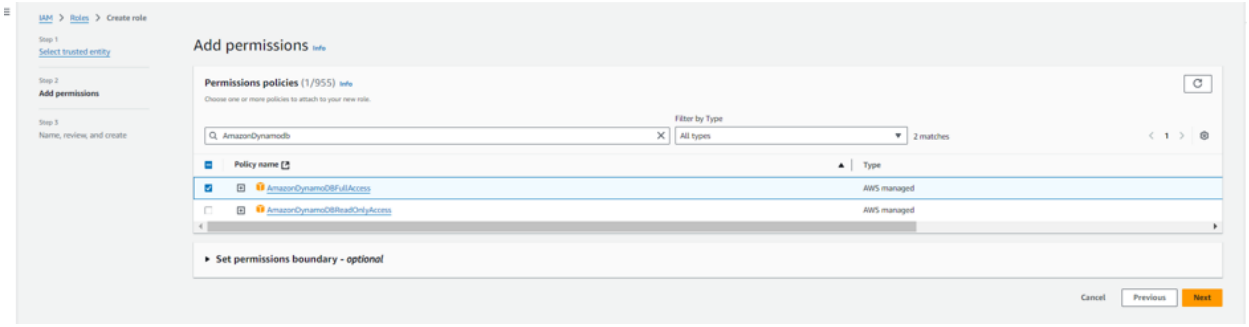
Milestone 5: IAM Role Setup

- **Activity 5.1:Create IAM Role.**
    - **In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.**





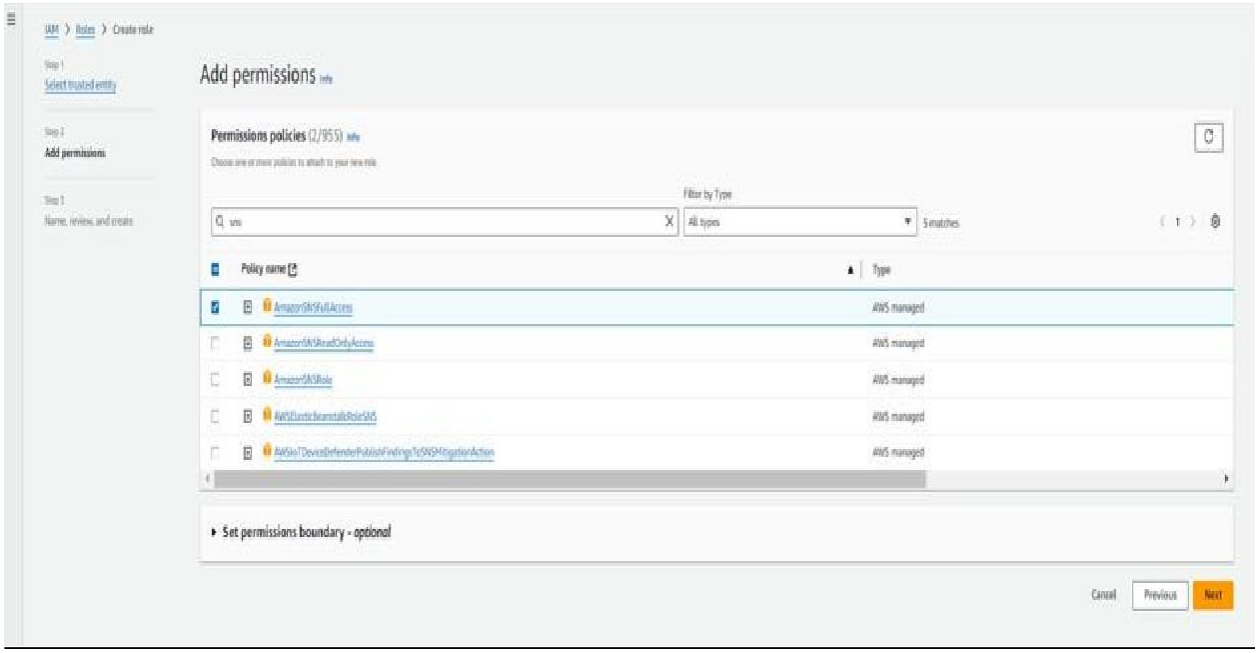


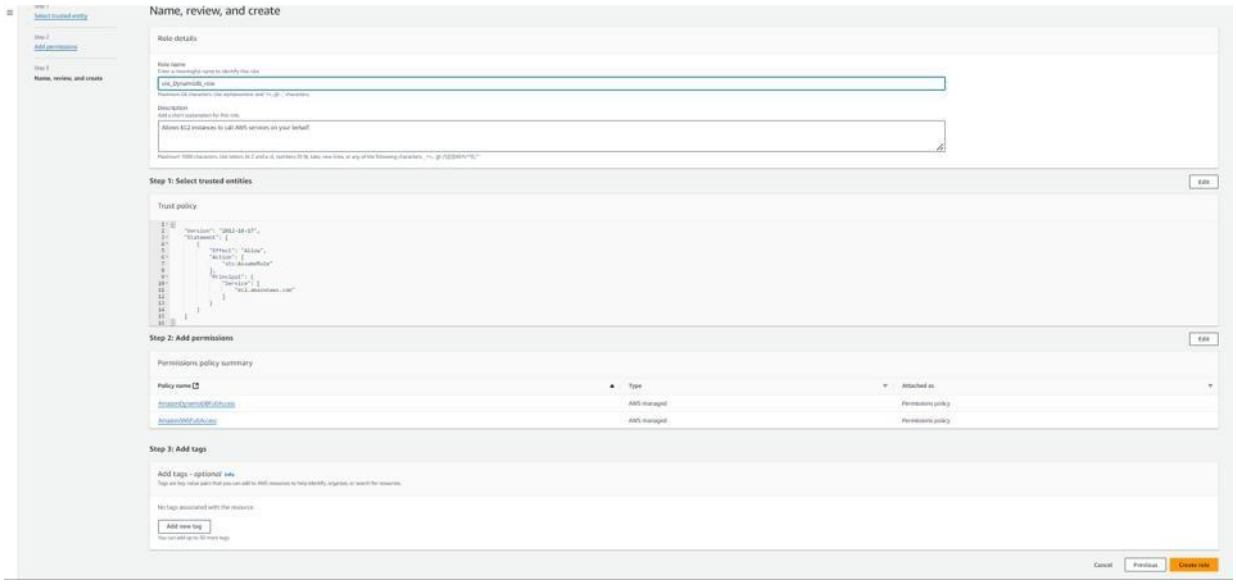


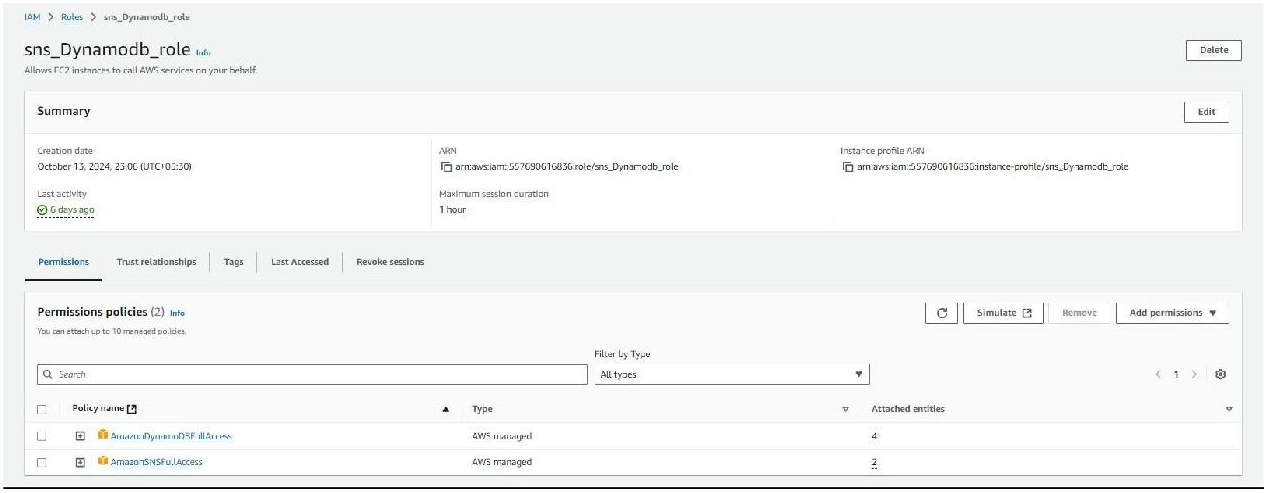
- **Activity 5.2: Attach Policies.**

  Attach the following policies to the role:

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

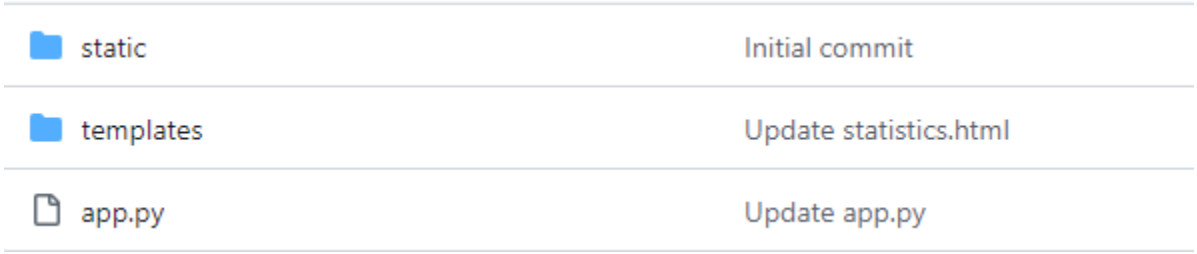

IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Add permissions Info
Permissions policies (2/955) Info
Choose one or more policies to attach to your new role.
Filter by Type
sns
All types
5 matches
Policy name
Type
AmazonSNSFullAccess
AWS managed
AmazonSNSReadOnlyAccess
AWS managed
AmazonSNSRole
AWS managed
AWSElasticBeanstalkRoleSNS
AWS managed
AWSIoTDeviceDefenderPublishFindingsToSNSMitigationAction
AWS managed
Set permissions boundary - optional
Cancel
Previous
Next

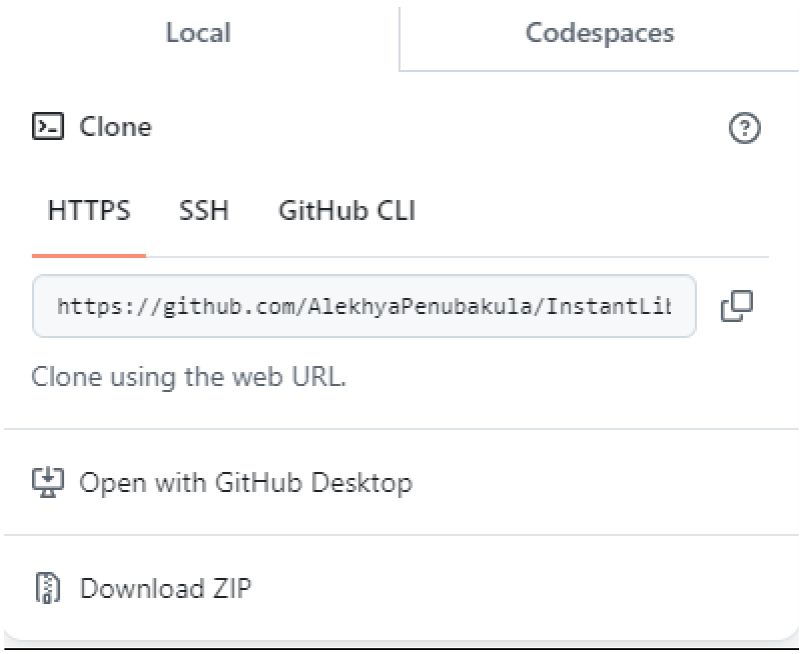

Name, review, and create
Role details
Step 1: Select trusted entities
Edit
Trust policy
Step 2: Add permissions
Edit
Permissions policy summary
Policy name
Type
Attached as
Step 3: Add tags
Add tags - optional
Add new tag
Cancel
Previous
Create role



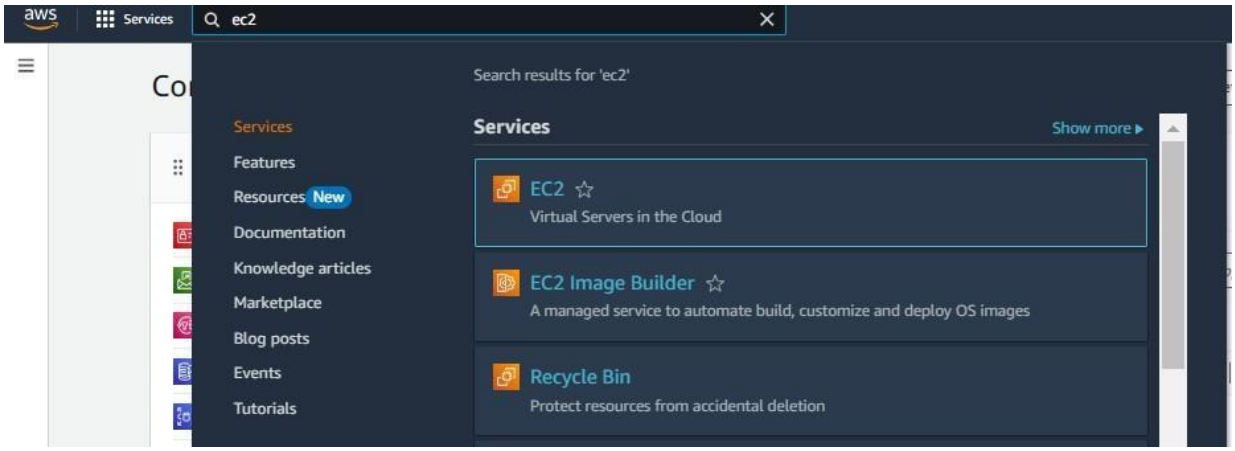
Milestone 6: EC2 Instance Setup

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

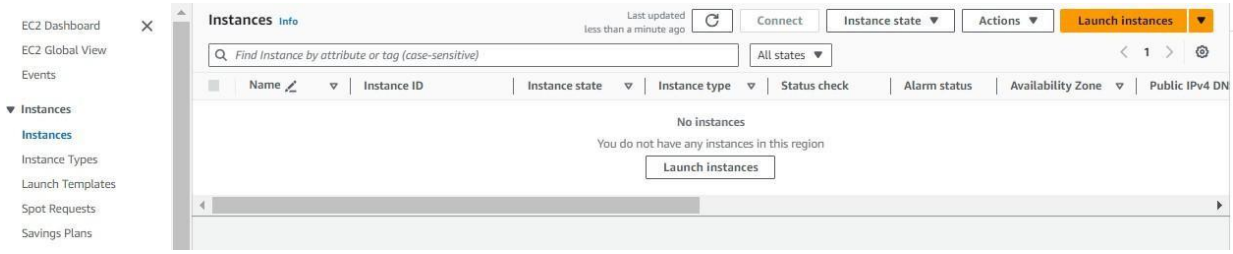


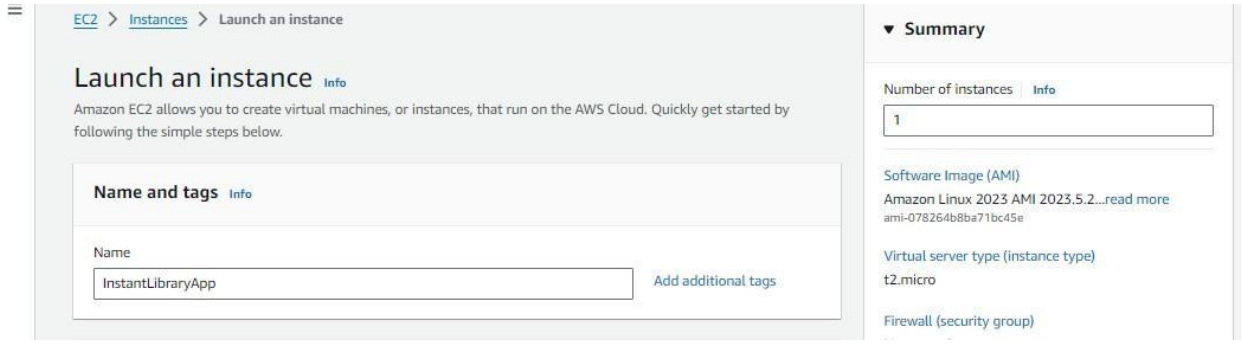


- **Activity 6.1: Launch an EC2 instance to host the Flask application.**
    - **Launch EC2 Instance**
        - **In the AWS Console, navigate to EC2 and launch a new instance.**

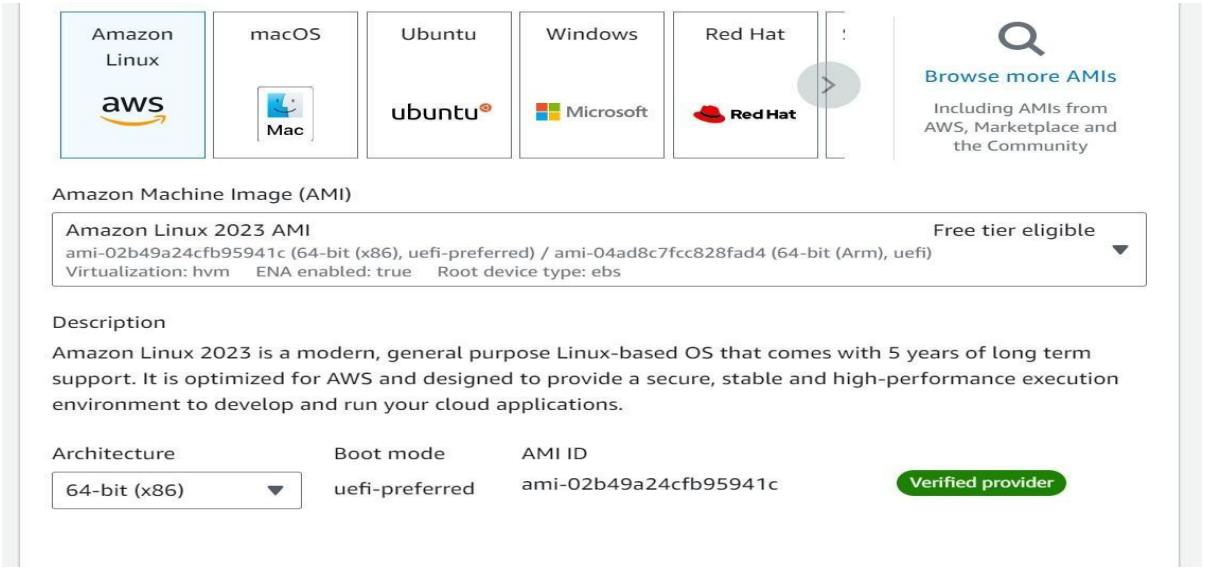


- **Click on Launch instance to launch EC2 instance**

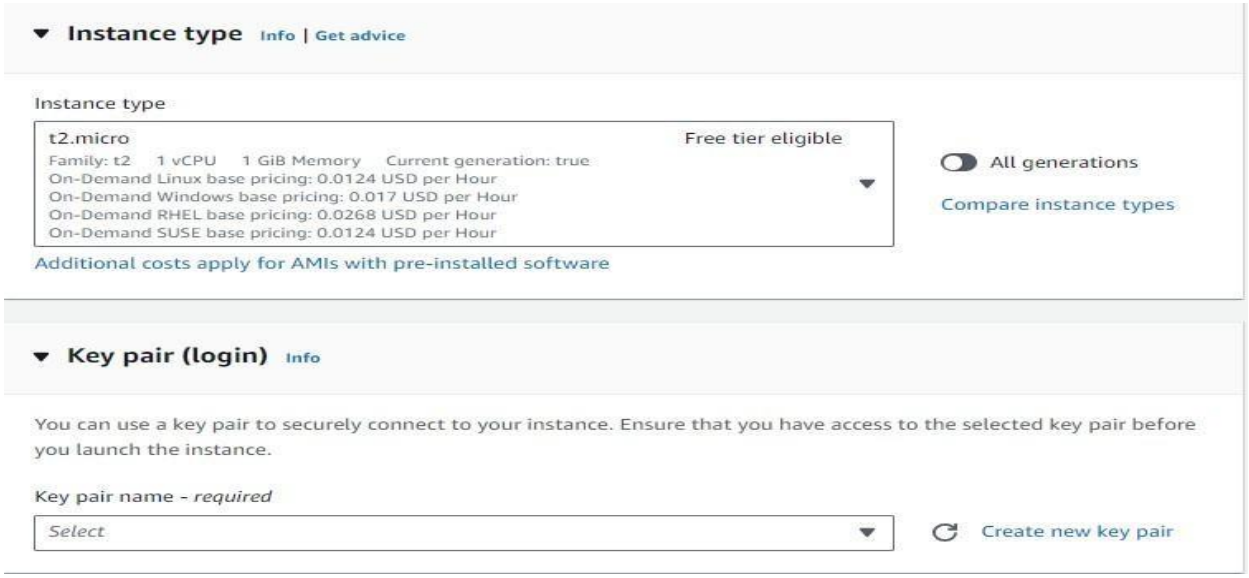


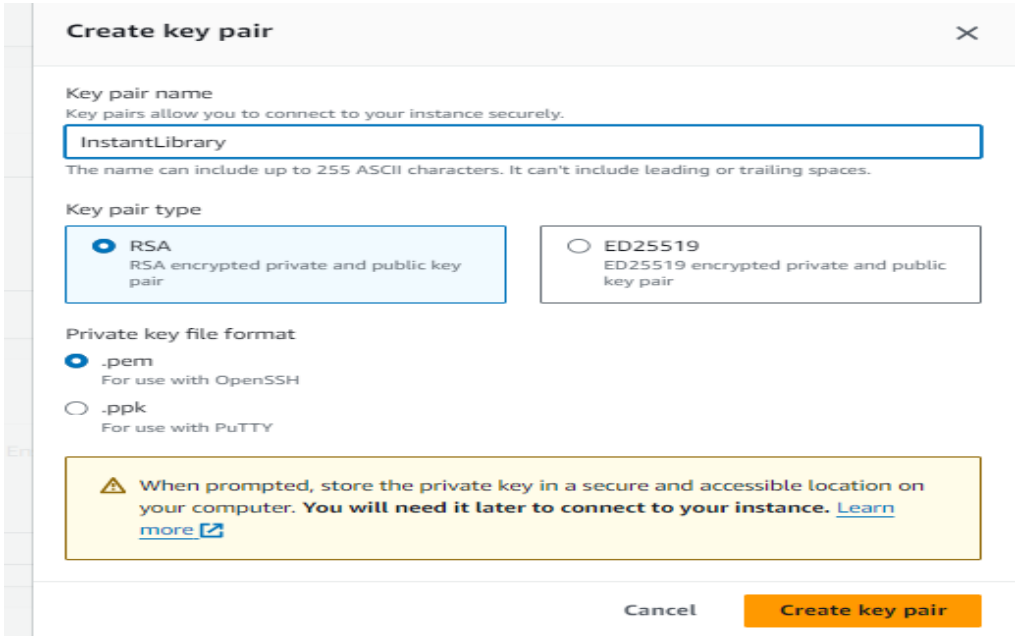


- **Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).**

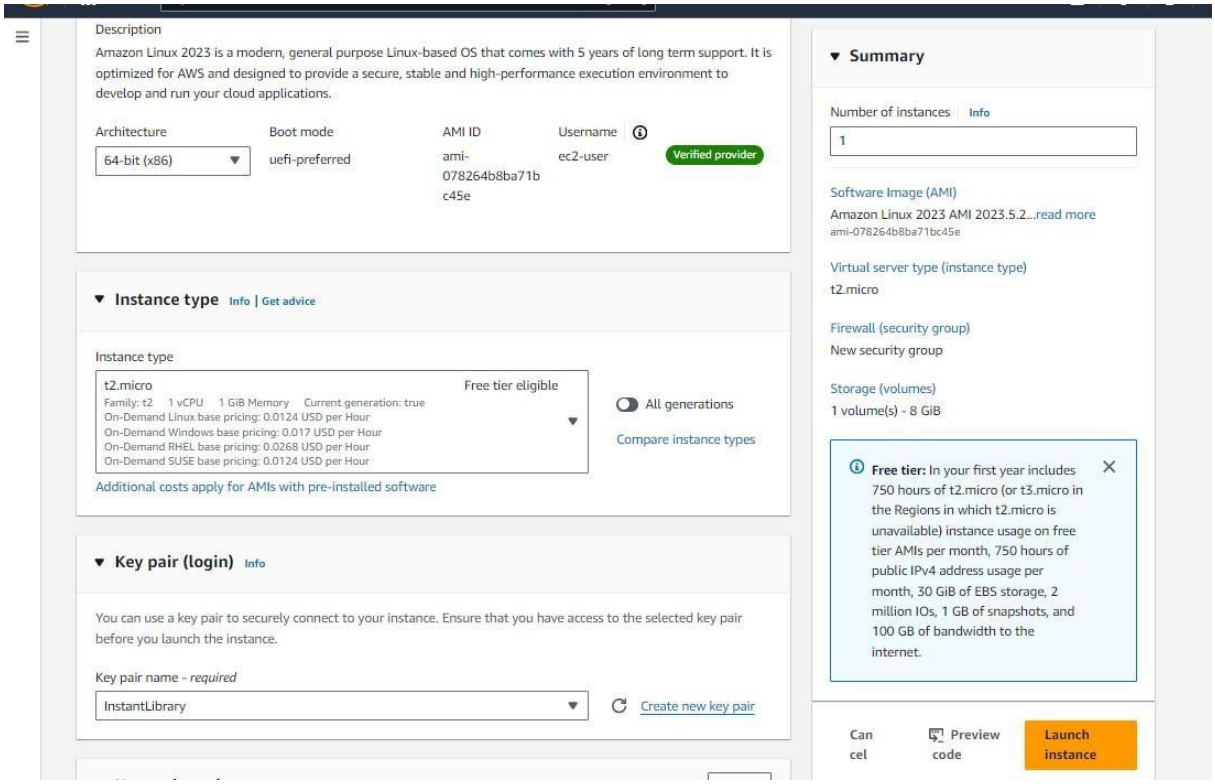


- **Create and download the key pair for Server access.**

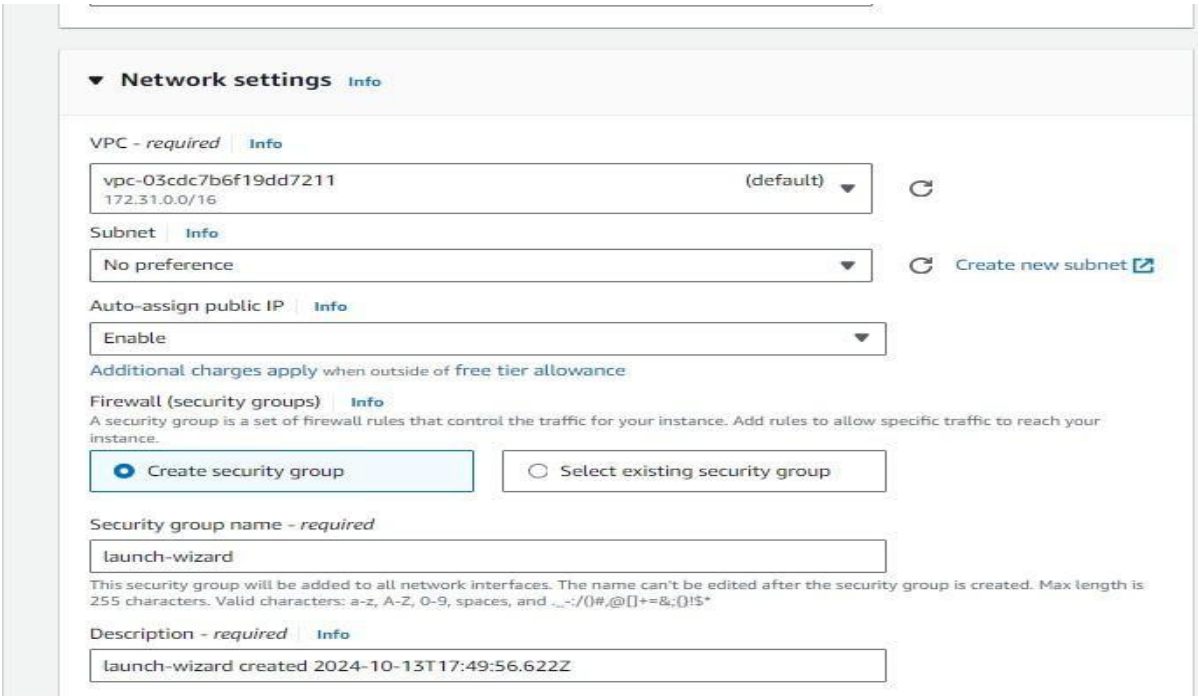


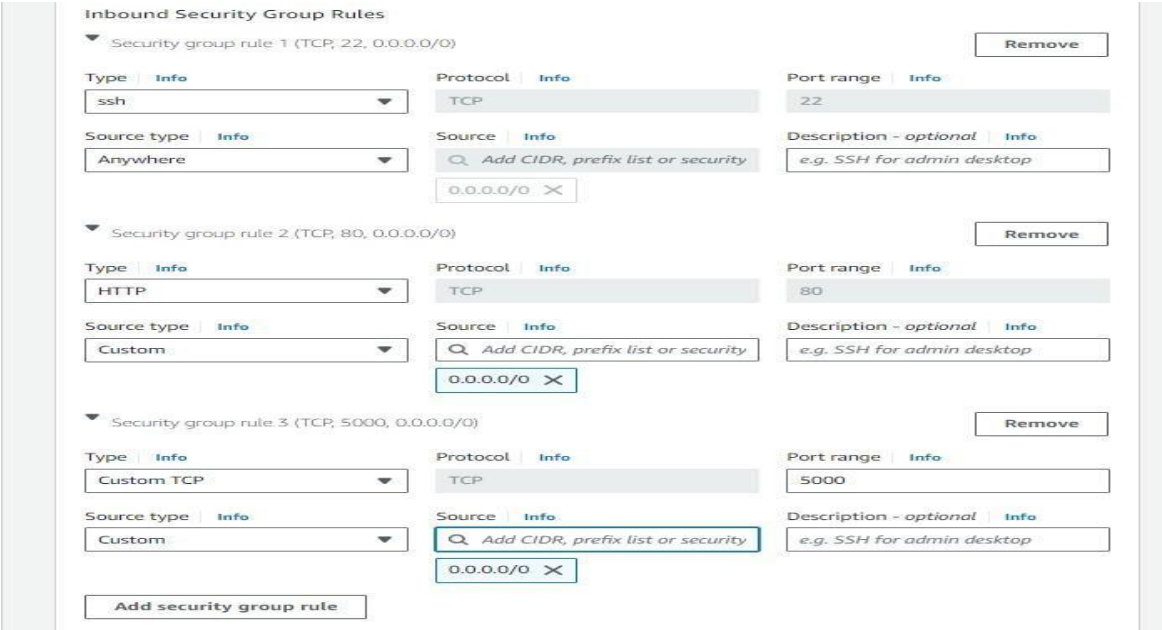


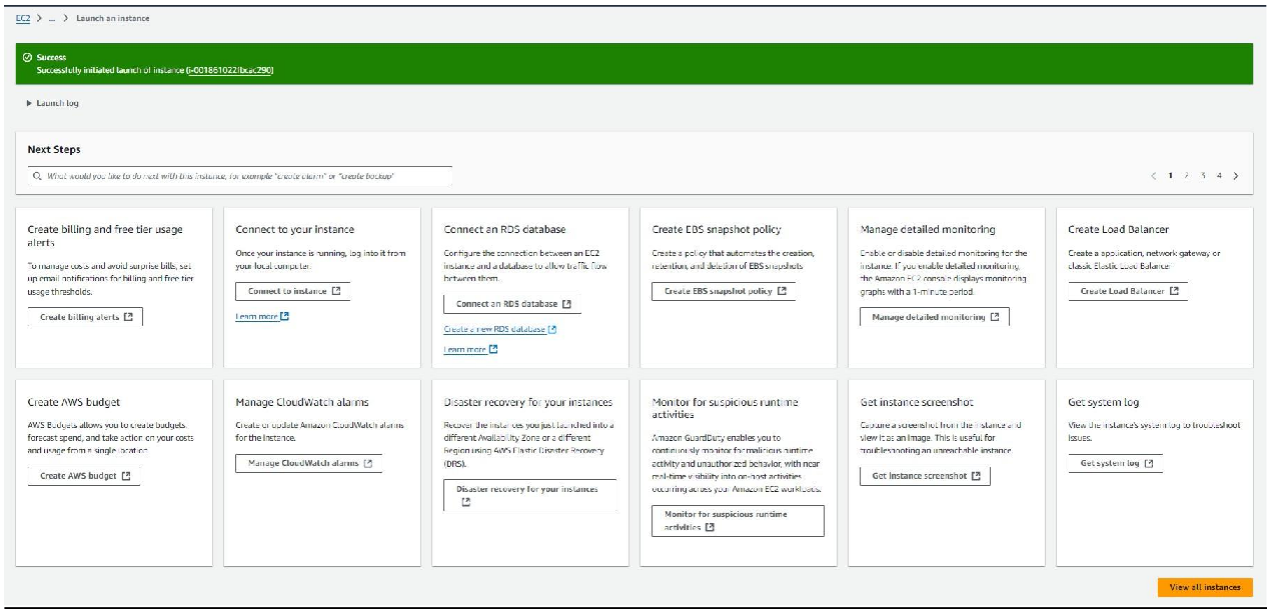
InstantLibrary.pem



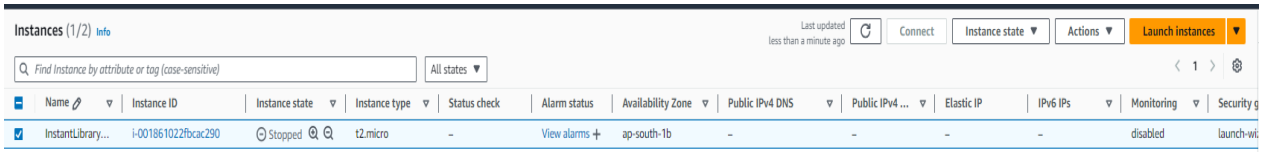
- **Activity 6.2:Configure security groups for HTTP, and SSH access.**

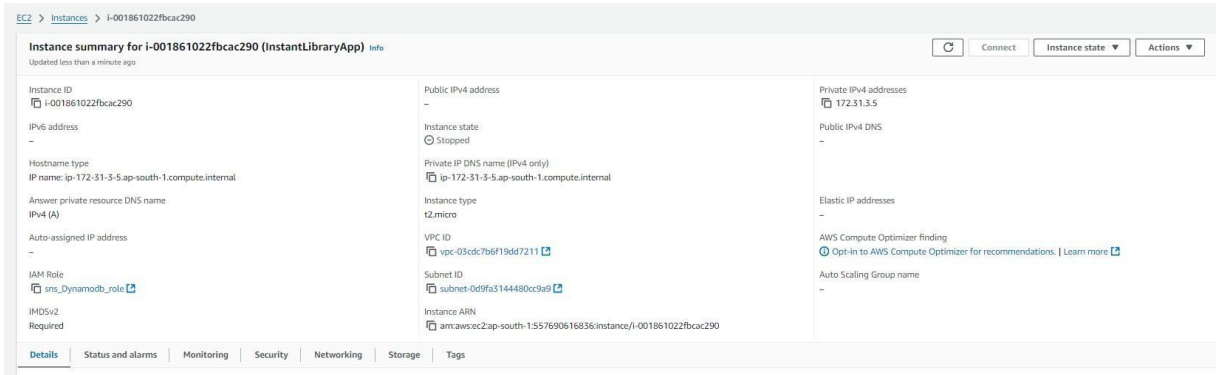


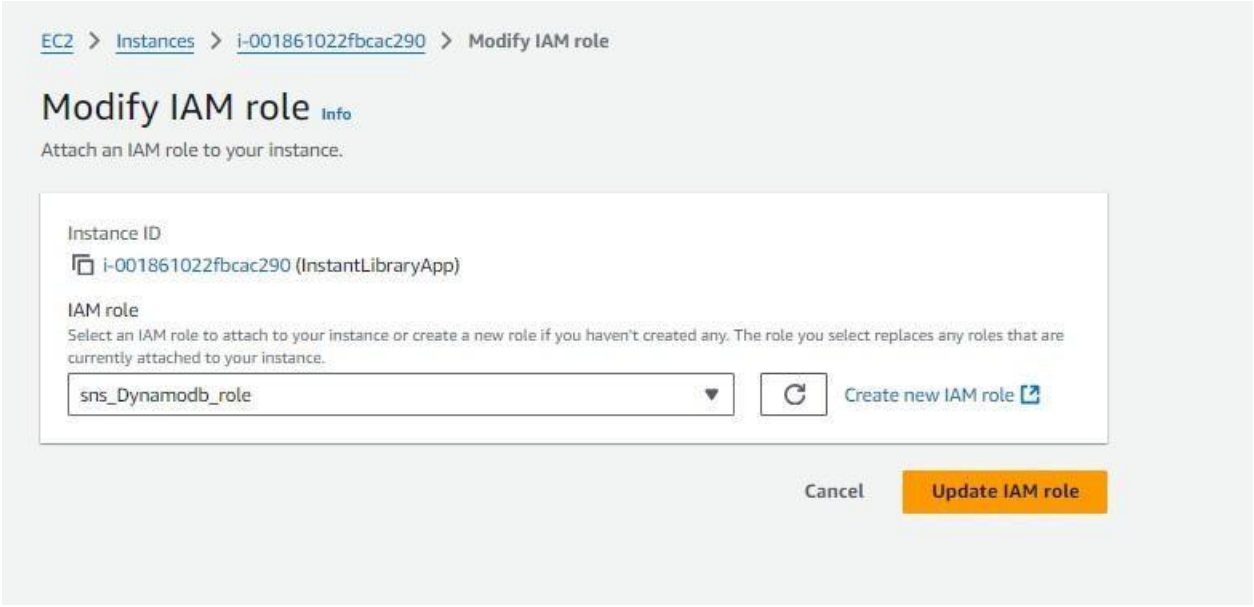




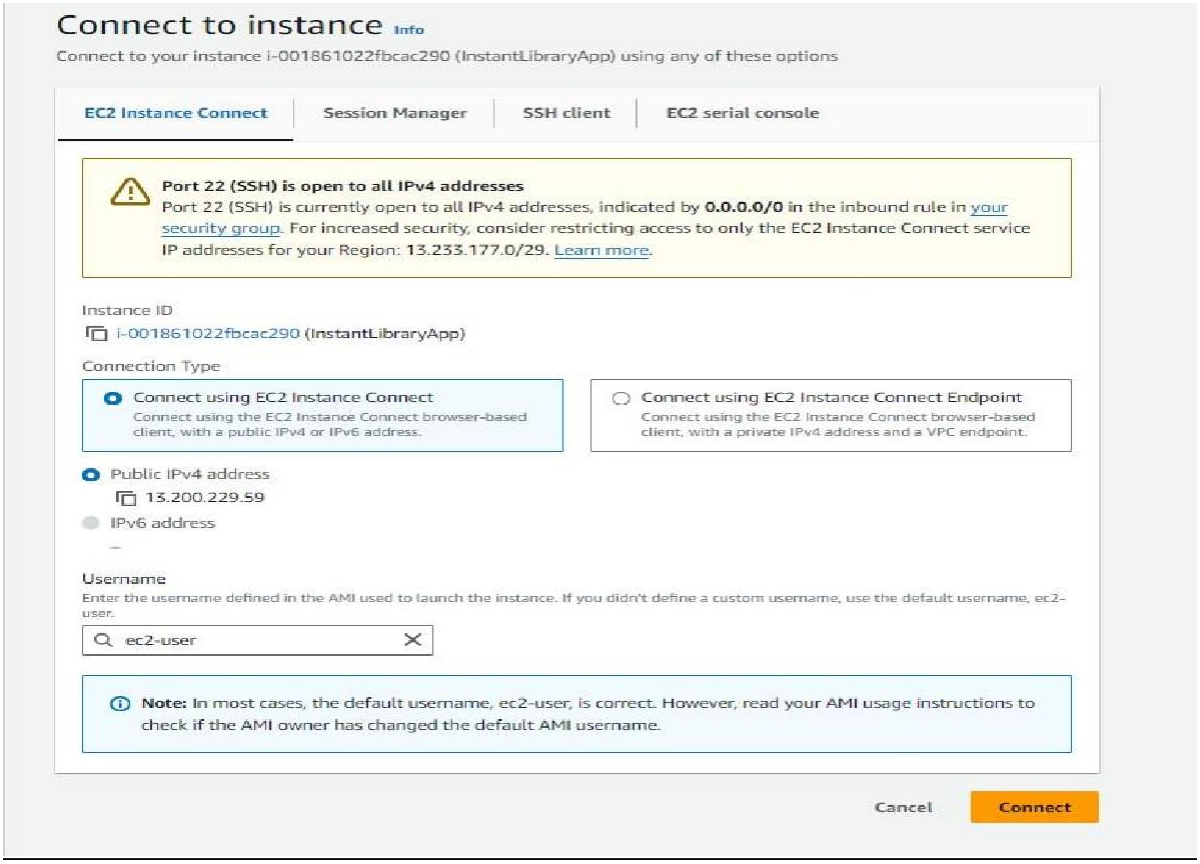
- **To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected, navigate to the EC2 section in the AWS Management Console. Select the EC2 instance you wish to connect to. At the top of the EC2 Dashboard, click the Connect button. From the connection methods presented, choose EC2 Instance Connect. Finally, click Connect again, and a new browser-based terminal will open, allowing you to access your EC2 instance directly from your browser.**

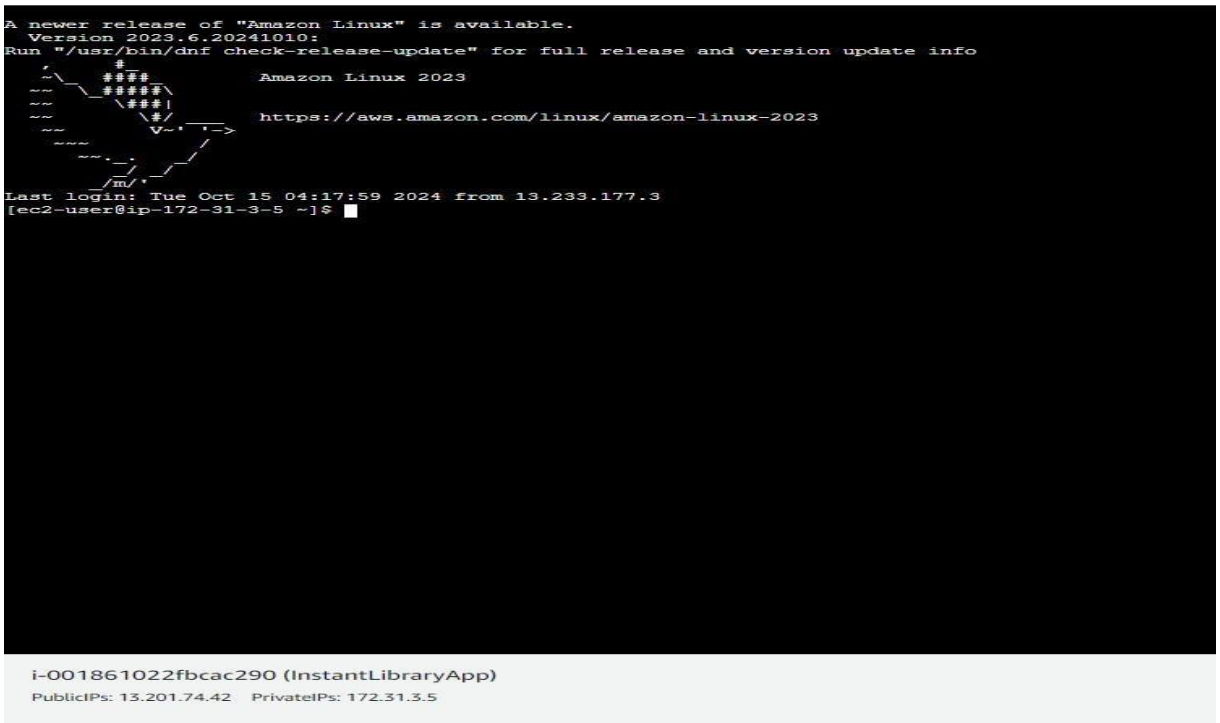






- **<u>Now connect the EC2 with the files</u>**


Connect to instance Info
Connect to your instance i-001861022fbcac290 (InstantLibraryApp) using any of these options
EC2 Instance Connect
Session Manager
SSH client
EC2 serial console
Port 22 (SSH) is open to all IPv4 addresses
Port 22 (SSH) is currently open to all IPv4 addresses, indicated by 0.0.0.0/0 in the inbound rule in your security group. For increased security, consider restricting access to only the EC2 Instance Connect service IP addresses for your Region: 13.233.177.0/29. Learn more.
Instance ID
i-001861022fbcac290 (InstantLibraryApp)
Connection Type
Connect using EC2 Instance Connect
Connect using the EC2 Instance Connect browser-based client, with a public IPv4 or IPv6 address.
Connect using EC2 Instance Connect Endpoint
Connect using the EC2 Instance Connect browser-based client, with a private IPv4 address and a VPC endpoint.
Public IPv4 address
13.200.229.59
IPv6 address
–
Username
Enter the username defined in the AMI used to launch the instance. If you didn't define a custom username, use the default username, ec2-user.
ec2-user
Note: In most cases, the default username, ec2-user, is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI username.
Cancel
Connect


A newer release of "Amazon Linux" is available.
Version 2023.6.20241010:
Run "/usr/bin/dnf check-release-update" for full release and version update info
Amazon Linux 2023
https://aws.amazon.com/linux/amazon-linux-2023
Last login: Tue Oct 15 04:17:59 2024 from 13.233.177.3
[ec2-user@ip-172-31-3-5 ~]$
i-001861022fbcac290 (InstantLibraryApp)
PublicIPs: 13.201.74.42 PrivateIPs: 172.31.3.5

Milestone 7: Deployment on EC2

Activity 7.1: Install Software on the EC2 Instance

Install Python3, Flask, and

Git: On Amazon Linux

2:

```
sudo yum update -y

sudo yum install python3 git

sudo pip3 install flask boto3
```

Verify Installations:

```
flask --version

git --version
```

Activity 7.2:Clone Your Flask Project from GitHub

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

Run: 'git clone https://github.com/vamsi200527/travelgoaws

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

'git clone https://github.com/SoftwareKarthik/travelgoproject

- **This will download your project to the EC2 instance.**

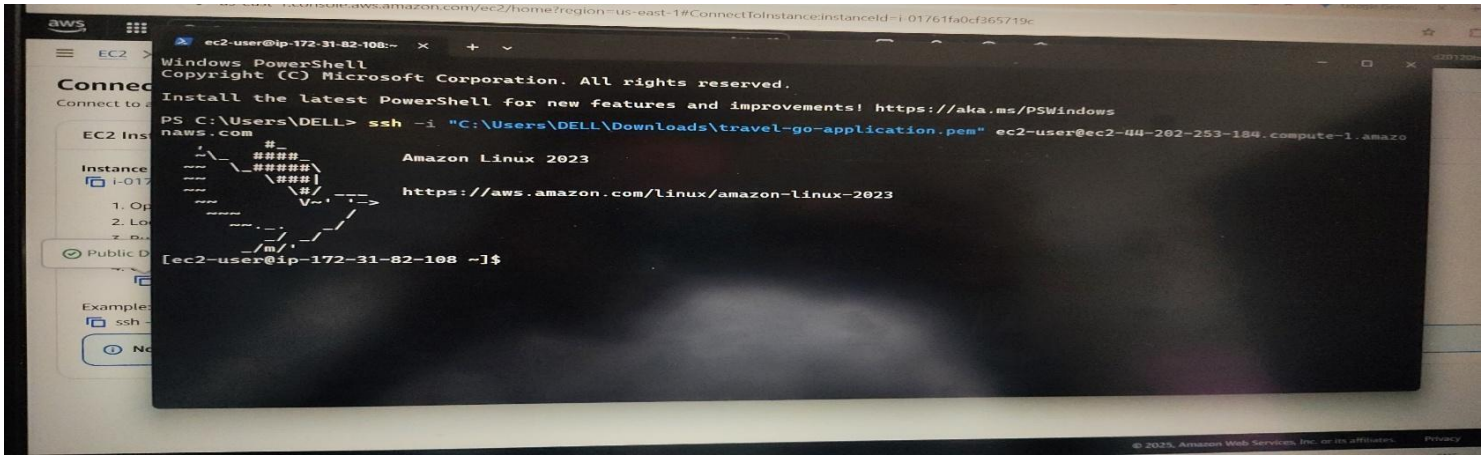
To navigate to the project directory, run the following command:

```
cd InstantLibrary
```

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

Run the Flask Application

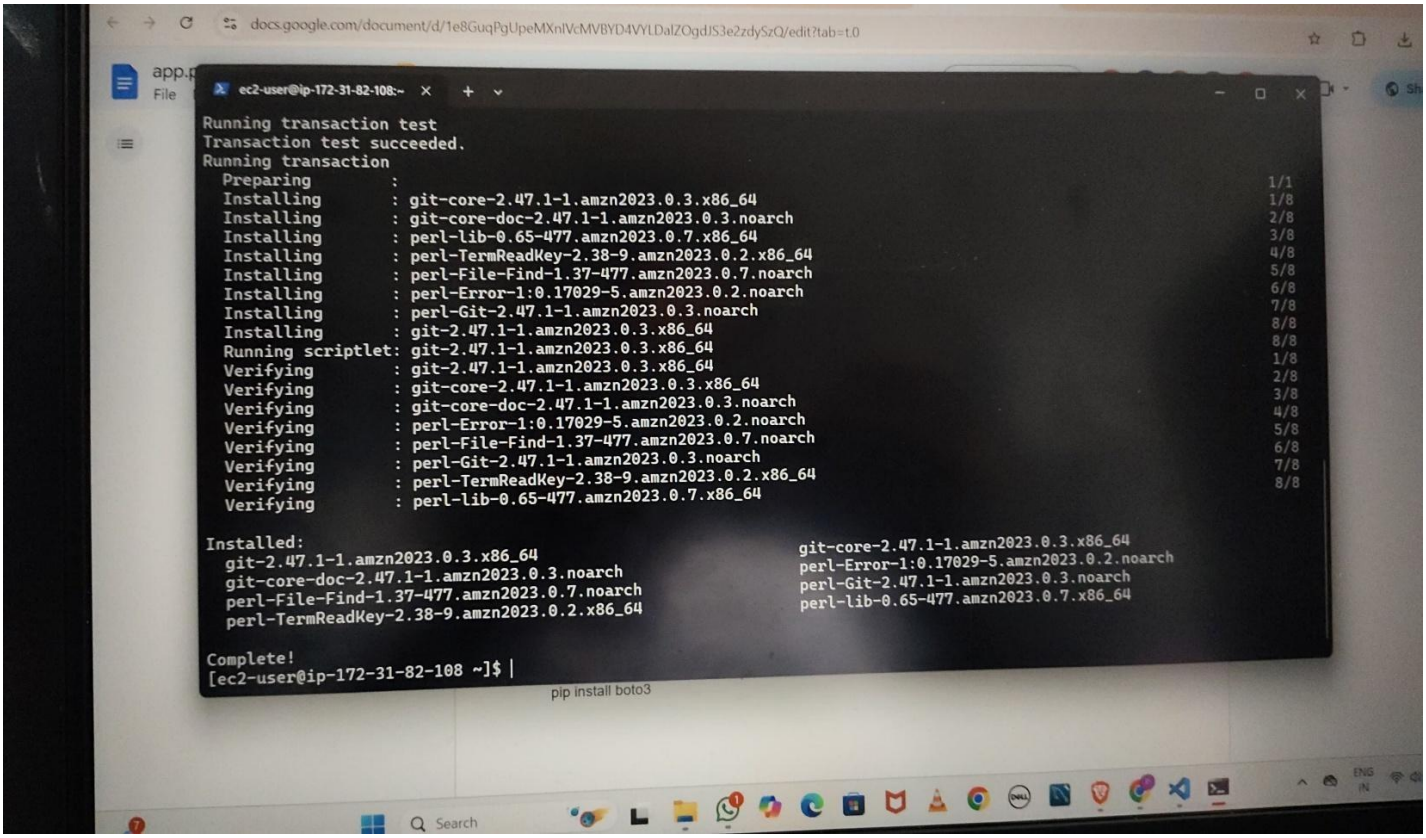
```
sudo flask run --host=0.0.0.0 --port=80
```



Verify the Flask app is running:

http://your-ec2-public-ip

- **Run the Flask app on the EC2 instance**



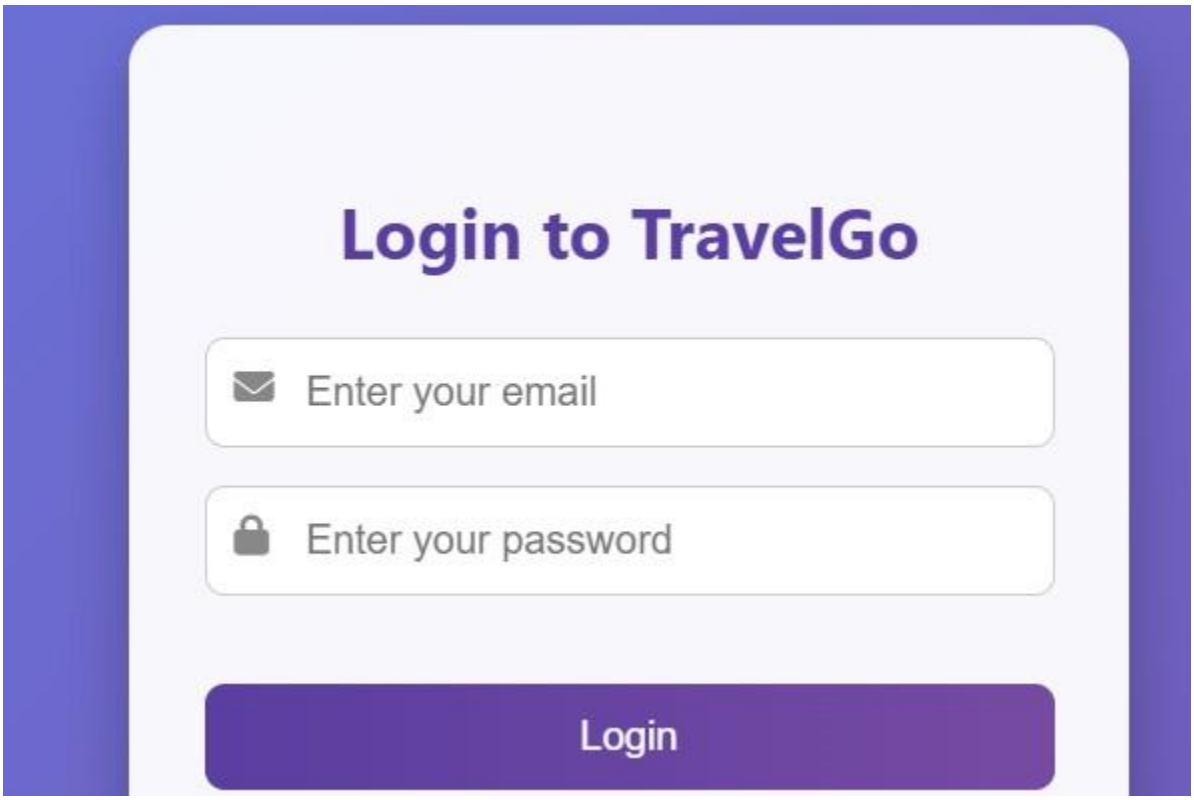
Access the website through:

http://172.31.16.68:5000/

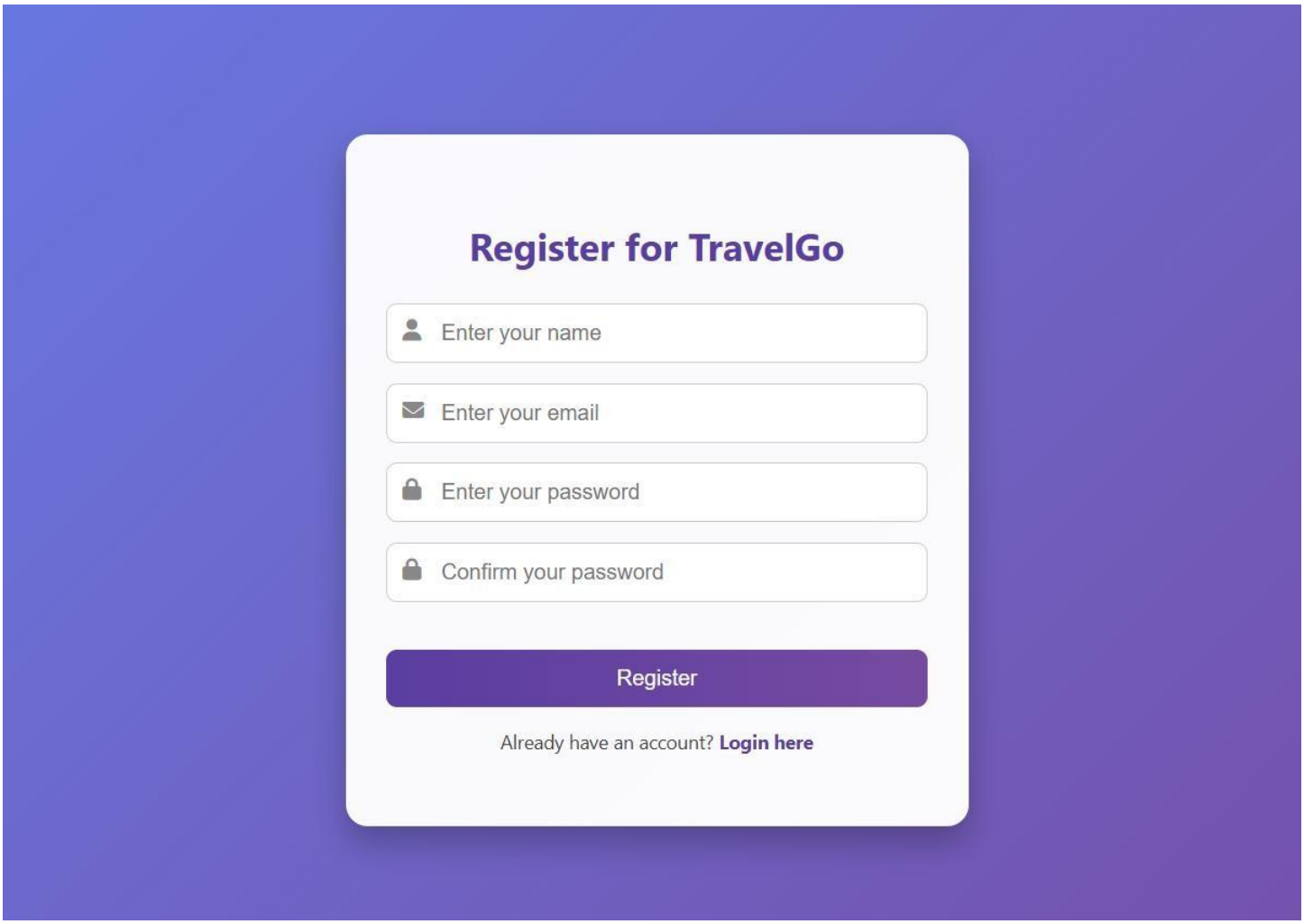
Milestone 8: Testing and Deployment

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

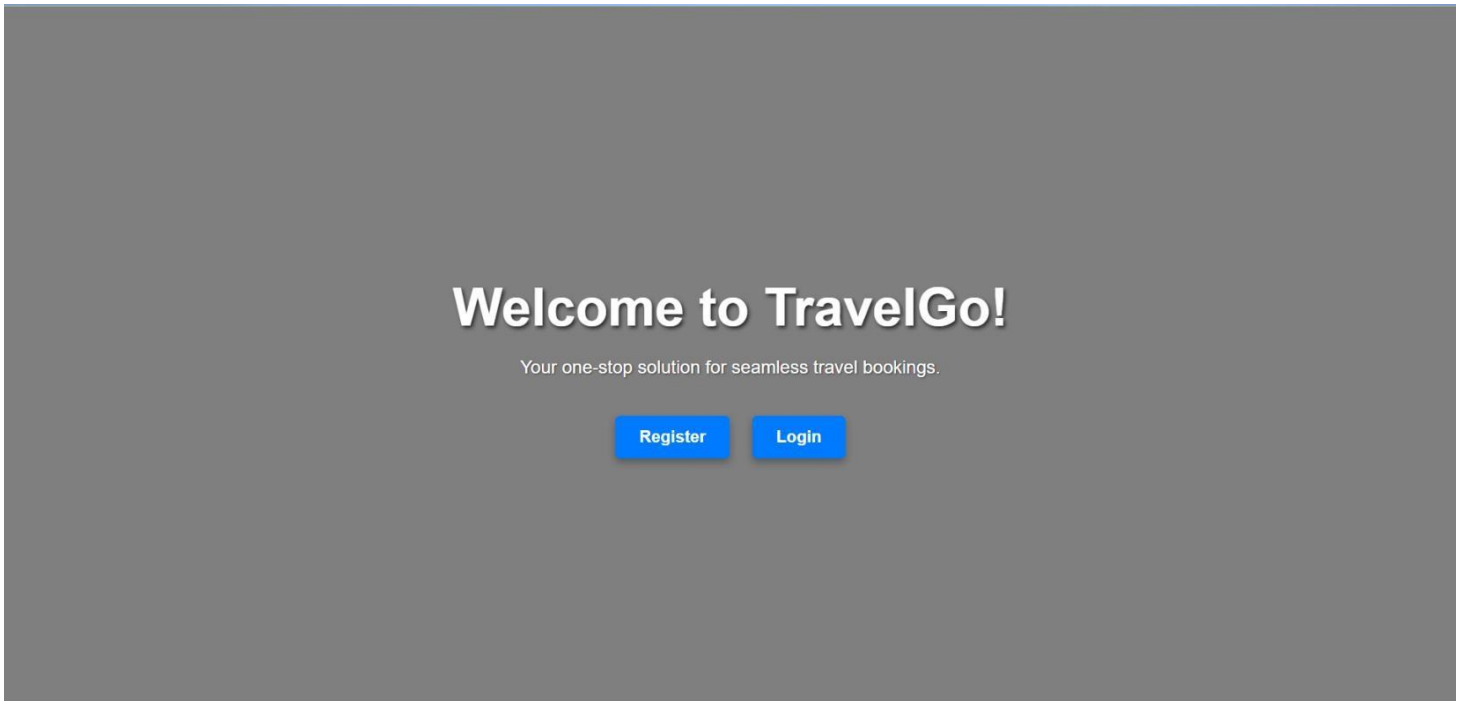
  Login Page:



Registration Page:



Front Page:-



Confirm Booking Page:

Conclusion:

The Greenfield University Travel Booking System has been successfully developed and deployed using a robust and scalable cloud architecture. By integrating Flask, AWS EC2, DynamoDB, and SNS, the platform offers a seamless experience for students and faculty to manage academic travel requests and bookings.

This cloud-based solution overcomes the limitations of traditional, manual travel management by automating booking processes, ensuring real-time communication, and enabling efficient tracking of travel data. The use of DynamoDB guarantees fast and secure storage, while AWS SNS keeps all stakeholders informed with instant email notifications.

The system's responsive interface, backed by scalable AWS services, ensures smooth operation even under high usage. From registration and login to travel request submissions and status tracking, all functionalities have been thoroughly tested and optimized.

In summary, this platform significantly improves the efficiency and transparency of academic travel planning at Greenfield University, demonstrating the power of modern cloud technologies in solving real-world administrative challenges.

Experience Gained :-

Working on the Greenfield University Travel Booking System project provided valuable hands-on experience across multiple areas of cloud computing, backend development, and full-stack deployment. The following skills and insights were gained:

- **Cloud Services Integration**

  Gained practical experience in setting up and integrating AWS services like **EC2**, **DynamoDB**, and **SNS**, learning how these services interact to support real-world web applications.

- **Flask Web Development**

  Built a scalable backend using **Flask**, implementing routing, user authentication, and session management while connecting it to a cloud-based NoSQL database.

- **DynamoDB Database Management**

  Designed and managed DynamoDB tables, learned how to perform CRUD operations using **boto3**, and understood best practices for data modeling in NoSQL environments.

- **Real-time Notification System**

  Used **AWS SNS** to send email alerts, improving user interaction and system responsiveness. This reinforced the importance of real-time communication in modern applications.

- **Deployment on AWS EC2**

  Learned how to launch, configure, and secure an **EC2 instance**, and successfully deployed a Flask app to a live production environment.

- **IAM Roles and Security Best Practices**

  Understood the importance of **IAM policies and permissions** by configuring roles for EC2 to securely interact with other AWS services.

- **Version Control and GitHub Integration**

Managed project code using **Git**, enabling better team collaboration and version tracking through GitHub.

- **Problem Solving and Debugging**

  Encountered and resolved various technical challenges during development and deployment, enhancing debugging and troubleshooting skills.

- **Project Management and Documentation**

  Documented the entire workflow, from architecture to deployment, ensuring the project can be maintained, scaled, or enhanced in the future.