Movie Magic–Smart Movie Ticket Booking System

Project Description:

With the increasing demand for a seamless and modern movie-watching experience, traditional ticket booking methods often fall short due to long queues, limited availability, and inconsistent service. To address this, the development team introduced Movie Magic—a smart, cloud-based movie ticket booking system. Built using Flask for backend development, hosted on AWS EC2, and integrated with DynamoDB for dynamic data management, the platform allows users to register, log in, and book movie tickets online with ease. Users can search for movies and events based on location, view real-time seat availability, and complete their bookings in just a few clicks. Upon booking, AWS SNS sends instant email notifications confirming ticket details, enhancing user engagement and trust. This cloud-native solution streamlines the entire movie ticketing process, ensuring fast, scalable, and user-friendly access to entertainment for all.

Scenarios :

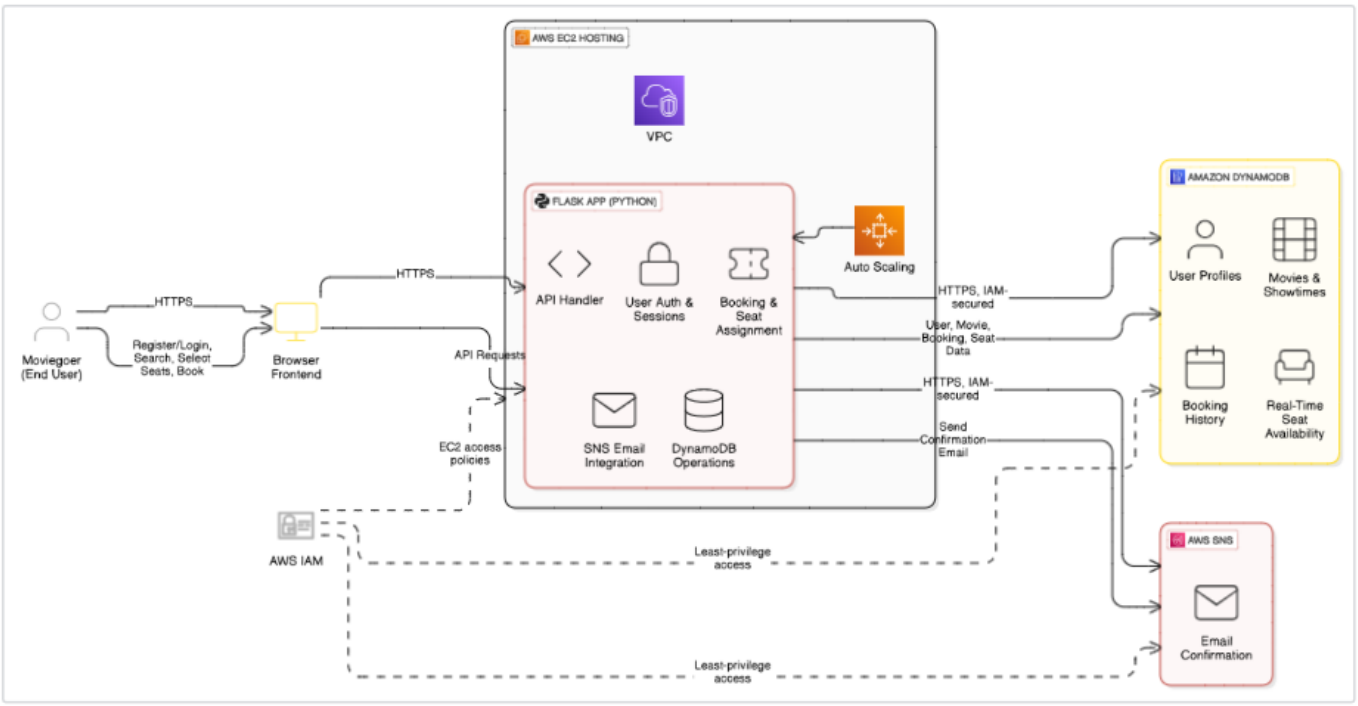
Scenario 1: Efficient Ticket Booking System for Users
In the Movie Magic System, AWS EC2 provides a reliable infrastructure capable of handling multiple users accessing the platform simultaneously. For example, a user can log in, navigate to the movie selection page, and seamlessly browse available shows and events in their city. They can then select a showtime, pick their preferred seats using an interactive layout, and confirm the booking—all in real-time. Flask manages backend processes, ensuring smooth data flow and quick response times even during high-traffic periods such as weekends or blockbuster releases.

Scenario 2: Seamless Booking Confirmation Notifications
When a user completes a ticket booking, the Movie Magic System leverages AWS SNS to send instant email notifications to confirm the booking. For instance, once the booking is submitted, Flask processes the transaction, and SNS sends a customized email to the user with all ticket details, including movie name, date, time, and seat numbers. This real-time notification system enhances the customer experience and reduces uncertainty, while DynamoDB securely stores the booking records for both users and admins to manage and track.

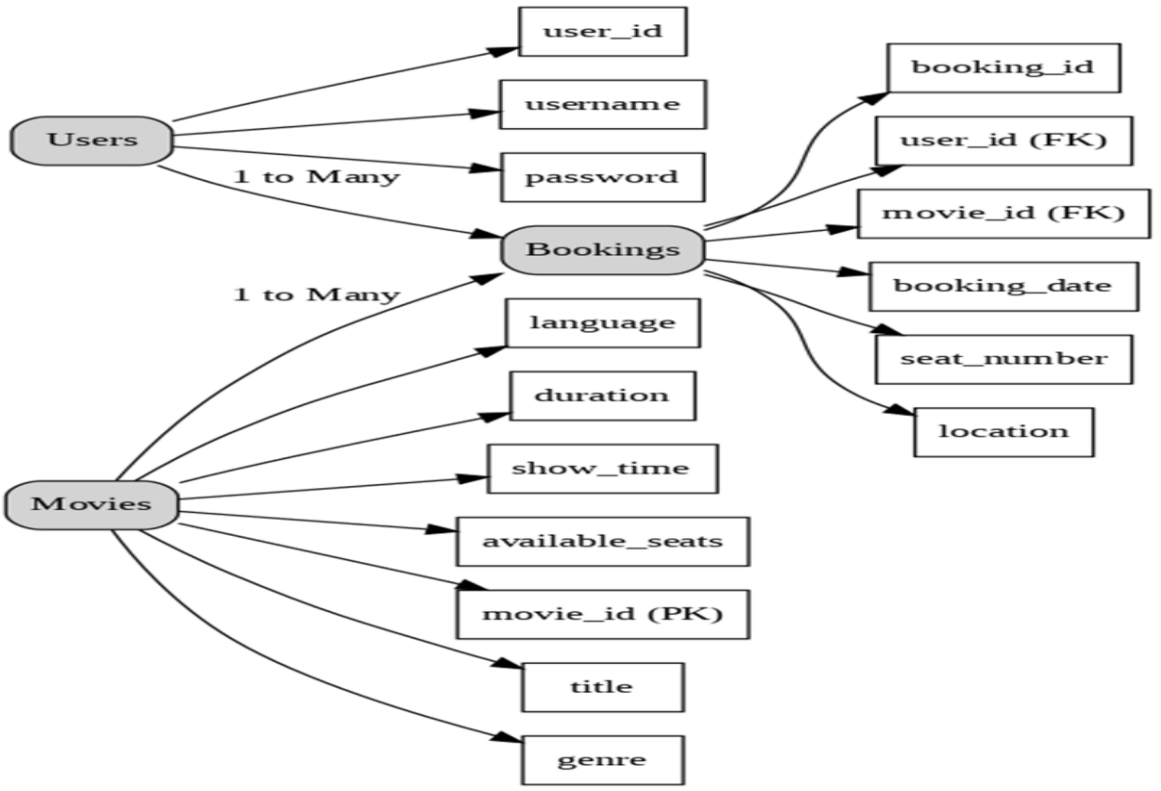
Scenario 3: Easy Access to Movies and Events
The Movie Magic platform offers users a seamless interface to explore currently running movies and upcoming live events. After logging in, a user can search by location or genre and instantly view listings with showtimes, ratings, and available seats. Flask dynamically fetches this data from DynamoDB, ensuring real-time updates on seat availability and event information. Meanwhile, the EC2-hosted application remains stable and responsive, even during traffic spikes, providing users with an uninterrupted and enjoyable booking experience.

AWS ARCHITECTURE

This AWS-based architecture powers a scalable and secure web application using Amazon EC2 for hosting the backend, with a lightweight framework like Flask handling core logic. Application data is stored in Amazon DynamoDB, ensuring fast, reliable access, while user access is managed through AWS IAM for secure authentication and control. Real-time alerts and system notifications are enabled via Amazon SNS, enhancing communication and user engagement.



Entity Relationship (ER)Diagram:



Pre-requisites

- **AWS Account Setup :** https://docs.aws.amazon.com/accounts/latest/reference/getting-started.html
- **AWS IAM (Identity and Access Management) :** https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
- **AWS EC2 (Elastic Compute Cloud) :** https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- **AWS DynamoDB :** https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
- **Amazon SNS :** https://docs.aws.amazon.com/sns/latest/dg/welcome.html
- **Git Documentation :** https://git-scm.com/doc
- **VS Code Installation : (download the VS Code using the below link or you can get that in Microsoft store)** https://code.visualstudio.com/download

Project Flow

Milestone 1. Backend Development and Application Setup

- **Develop the Backend Using Flask.**
- **Integrate AWS Services Using boto3.**

Milestone 2. AWS Account Setup and Login

- **Set up an AWS account if not already done.**
- **Log in to the AWS Management Console**

Milestone 3. DynamoDB Database Creation and Setup

- **Create a DynamoDB Table.**
- **Configure Attributes for User Data and Book Requests.**

Milestone 4. SNS Notification Setup

- **Create SNS topics for book request notifications.**
- **Subscribe users and library staff to SNS email notifications.**

Milestone 5. IAM Role Setup

- **Create IAM Role**
- **Attach Policies**

Milestone 6. EC2 Instance Setup

- **Launch an EC2 instance to host the Flask application.**
- **Configure security groups for HTTP, and SSH access.**

Milestone 7. Deployment on EC2

- **Upload Flask Files**
- **Run the Flask App**

Milestone 8. Testing and Deployment

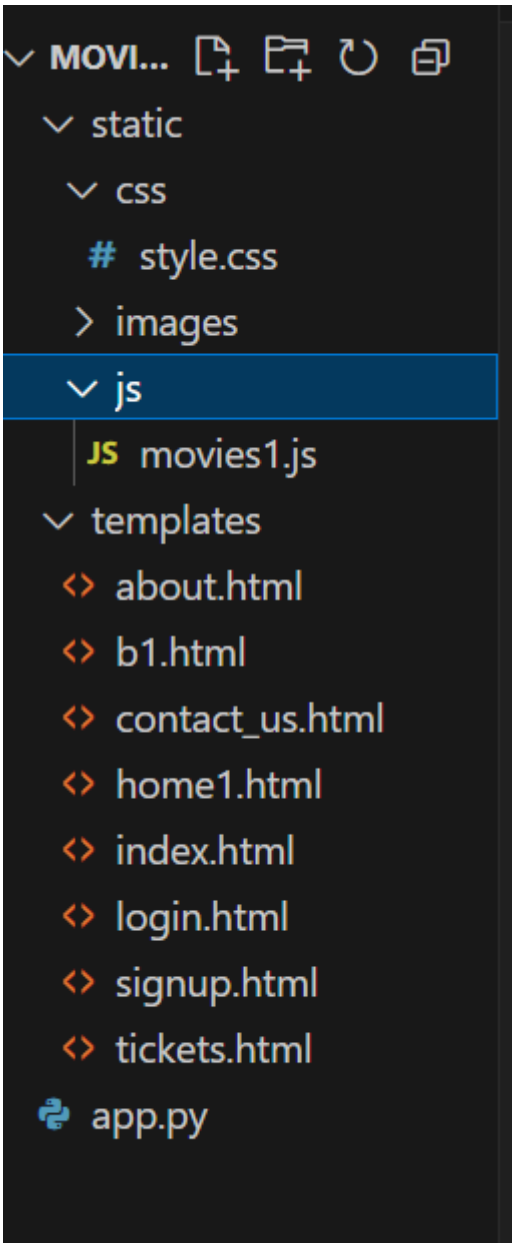
- **Conduct functional testing to verify user registration, login, book requests, and notifications.**
-

Milestone 1 : Web Application Development And Setup

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

LOCAL DEPLOYMENT

- File Explorer Structure



Description of the code :

- **Flask App Initialization**

  Imports and Configuration:

```
from flask import Flask, render_template, request, redirect, url_for, session, flash
from werkzeug.security import generate_password_hash, check_password_hash
from datetime import datetime
import boto3
import uuid
import json
import os
from botocore.exceptions import ClientError
```

Description: This project uses Flask for routing, session management, and user authentication with secure password hashing. It integrates AWS services via Boto3 for handling data storage, notifications, and unique user operations.

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

Description:A new Flask application instance is initialized, and a secret key is set to securely manage user sessions and protect against cookie tampering.

- **Dynamodb and SNS Setup:**

```
# Use a static secret key
app.secret_key = 'your_static_secret_key_here'  # Replace with your own secret string

# AWS Configuration - read from environment variables for security
AWS_REGION = os.environ.get('AWS_REGION', 'ap-south-1')

# Fix the SNS_TOPIC_ARN assignment - this was the main issue
# Instead of using os.environ.get with the ARN as the key, set it directly
SNS_TOPIC_ARN = 'arn:aws:sns:ap-south-1:605134430972:MovieTicketNotifications'

# Initialize AWS services with proper credentials handling
# On EC2, this will use the instance profile/role automatically
dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)
sns = boto3.client('sns', region_name=AWS_REGION)

# DynamoDB tables
USERS_TABLE_NAME = os.environ.get('USERS_TABLE_NAME', 'MovieMagic_Users')
BOOKINGS_TABLE_NAME = os.environ.get('BOOKINGS_TABLE_NAME', 'MovieMagic_Bookings')

users_table = dynamodb.Table(USERS_TABLE_NAME)
bookings_table = dynamodb.Table(BOOKINGS_TABLE_NAME)
```

Description: Use **boto3** to connect to **DynamoDB** for handling user registration, movie bookings database operations and also mention region_name where Dynamodb tables are created.

- **SNS Connection**
  - **Description:** Configure **SNS** to send notifications when a movie ticket is booked. 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.

- **Function to send the Notifications:**

Description: This function sends a booking confirmation email using AWS SNS. It formats the booking details into a message and publishes it to a specified SNS topic, notifying the user via email about their successful movie ticket booking.

```
def send_booking_confirmation(booking):
    """Send booking confirmation email using SNS"""
    # Check if SNS_TOPIC_ARN is set
    if not SNS_TOPIC_ARN:
        print("SNS_TOPIC_ARN is not set. Unable to send notification.")
        return False

    try:
        # Format email content
        email_subject = f"MovieMagic Booking Confirmation - {booking['booking_id']}"

        email_message = f"""
        Hello {booking['user_name']},

        Your movie ticket booking is confirmed!

        Booking Details:
        ---------------
        Booking ID: {booking['booking_id']}
        Movie: {booking['movie_name']}
        Date: {booking['date']}
        Time: {booking['time']}
        Theater: {booking['theater']}
        Location: {booking['address']}
        Seats: {booking['seats']}
        Amount Paid: ₹{booking['amount_paid']}

        Please show this confirmation at the theater to collect your tickets.
```

```
    Thank you for choosing MovieMagic!
    """

    # User email
    user_email = booking['booked_by']

    print(f"Attempting to send notification to {user_email} via SNS topic {SNS_TOPIC_ARN}")

    # Send directly to the email using SNS
    response = sns.publish(
        TopicArn=SNS_TOPIC_ARN,
        Subject=email_subject,
        Message=email_message,
        MessageAttributes={
            'email': {
                'DataType': 'String',
                'StringValue': user_email
            }
        }
    )

    print(f"SNS publish response: {response}")
    print(f"Booking confirmation sent to {user_email}")
    return True

except Exception as e:
    print(f"Error sending booking confirmation: {str(e)}")
    return False
```

- **Routes for Web Pages**

- **Register User:** Collecting registration data, hashes the password, and stores user details in the database.

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

        try:
            # Check if user already exists
            response = users_table.get_item(Key={'email': email})
            if 'Item' in response:
                flash('Email already registered!', 'danger')
                return redirect(url_for('signup'))

            # Create new user in DynamoDB
            user_id = str(uuid.uuid4())
            users_table.put_item(
                Item={
                    'id': user_id,
                    'name': name,
                    'email': email,
                    'password': password,
                    'created_at': datetime.now().isoformat()
                }
            )

            flash('Registration successful! Please login.', 'success')
            return redirect(url_for('login'))
                return redirect(url_for('login'))

            except ClientError as e:
                print(f"Error accessing DynamoDB: {e.response['Error']['Message']}")
                flash('An error occurred during registration. Please try again.', 'danger')

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

- **login Route (GET/POST)**: Verifies user credentials, increments login count, and redirects to the dashboard on success.

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

        try:
            # Get user from DynamoDB
            response = users_table.get_item(Key={'email': email})

            if 'Item' in response:
                user = response['Item']
                if check_password_hash(user['password'], password):
                    session['user'] = {
                        'id': user['id'],
                        'name': user['name'],
                        'email': user['email']
                    }
                    return redirect(url_for('home1'))

            flash('Invalid email or password', 'danger')
        except ClientError as e:
            print(f"Error accessing DynamoDB: {e.response['Error']['Message']}")
            flash('An error occurred. Please try again later.', 'danger')

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

- These Flask routes handle key navigation in the app: /logout logs out the user by clearing the session and showing a flash message; /home1 is a protected route accessible only to logged-in users; /about and /contact_us render static pages with information about the app and ways to get in touch.

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

# Application Routes
@app.route('/home1')
def home1():
    if 'user' not in session:
        return redirect(url_for('login'))
    return render_template('home1.html')

@app.route('/about')
def about():
    return render_template('about.html')

@app.route('/contact_us')
def contact():
    return render_template('contact_us.html')
```

- **Booking Page Route:**

Description: This route displays the booking page (b1.html) with movie, theater, address, and price details passed as query parameters. It ensures only logged-in users can access the page.

```python
# Booking page route
@app.route('/b1', methods=['GET'], endpoint='b1')  # Add explicit endpoint
def booking_page():
    if 'user' not in session:
        return redirect(url_for('login'))

    return render_template('b1.html',
        movie=request.args.get('movie'),
        theater=request.args.get('theater'),
        address=request.args.get('address'),
        price=request.args.get('price')
    )
```

- **Tickets Page Route:**

Description: This route processes movie ticket bookings by collecting form data, generating a unique booking ID, storing details in DynamoDB, and sending a confirmation email via AWS SNS. It then displays the booking details on the tickets page.

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

    try:
        # Extract booking details from form
        movie_name = request.form.get('movie')
        booking_date = request.form.get('date')
        show_time = request.form.get('time')
        theater_name = request.form.get('theater')
        theater_address = request.form.get('address')
        selected_seats = request.form.get('seats')  # Changed from selected_seats
        amount_paid = request.form.get('amount')    # Changed from total_price

        # Generate a unique booking ID
        booking_id = f"MVM-{datetime.now().strftime('%Y%m%d')}-{str(uuid.uuid4())[:8]}"

        # Store booking in DynamoDB
        booking_item = {
            'booking_id': booking_id,
            'movie_name': movie_name,
            'date': booking_date,
            'time': show_time,
            'theater': theater_name,
            'address': theater_address,
            'booked_by': session['user']['email'],
            'user_name': session['user']['name'],
            'seats': selected_seats,
```

```
            'user_name': session['user']['name'],
            'seats': selected_seats,
            'amount_paid': amount_paid,
            'booking_time': datetime.now().isoformat()
        }

        bookings_table.put_item(Item=booking_item)

        # Send email notification via SNS
        notification_sent = send_booking_confirmation(booking_item)
        if notification_sent:
            flash('Booking confirmation has been sent to your email!', 'success')

        # Pass the booking details to the tickets template
        return render_template('tickets.html', booking=booking_item)

    except Exception as e:
        print(f"Error processing booking: {str(e)}")
        flash('Error processing booking', 'danger')
        return redirect(url_for('home1'))
```

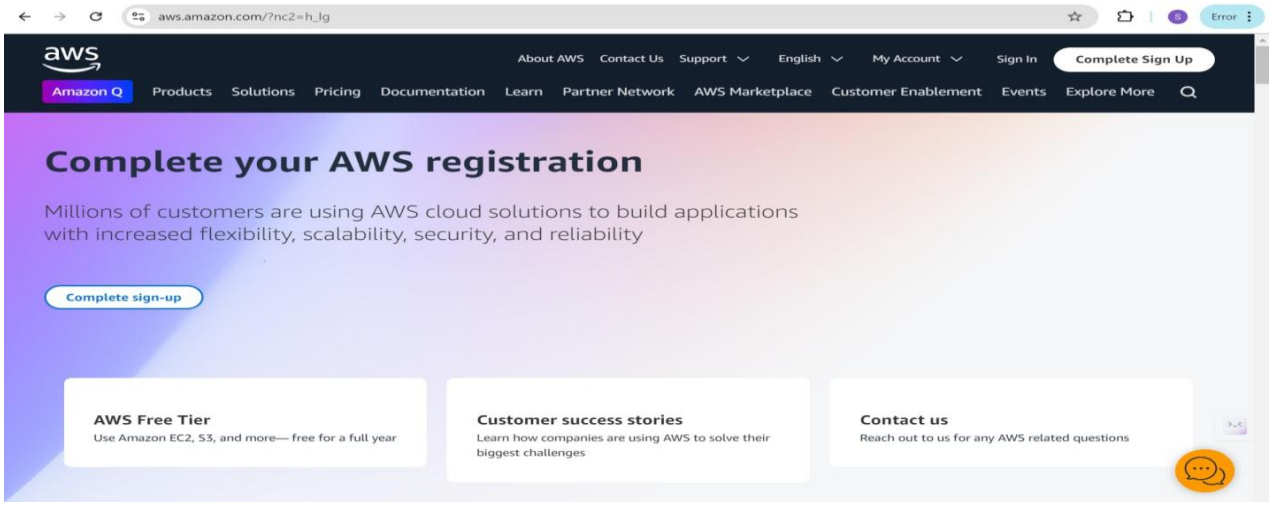
Application Entry point:

```
if __name__ == '__main__':
    # Using Flask's built-in server as requested
    port = int(os.environ.get('PORT', 5000))
    # You can set debug=False in production
    app.run(host='0.0.0.0', port=port, debug=True)
```

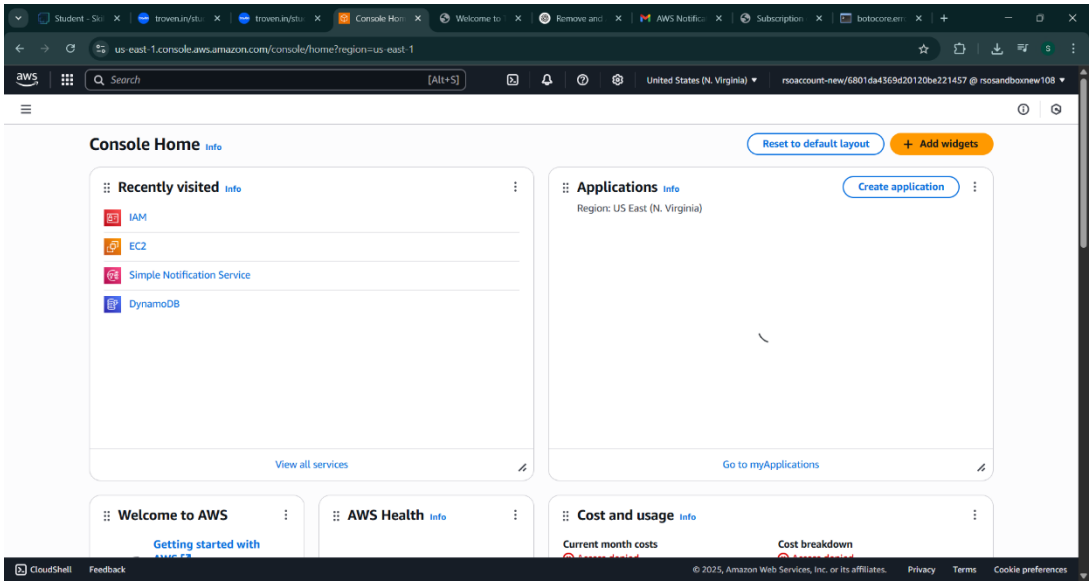
- **Description**: This block starts the Flask application using the built-in development server, setting the host, port, and enabling debug mode for easier development and testing.

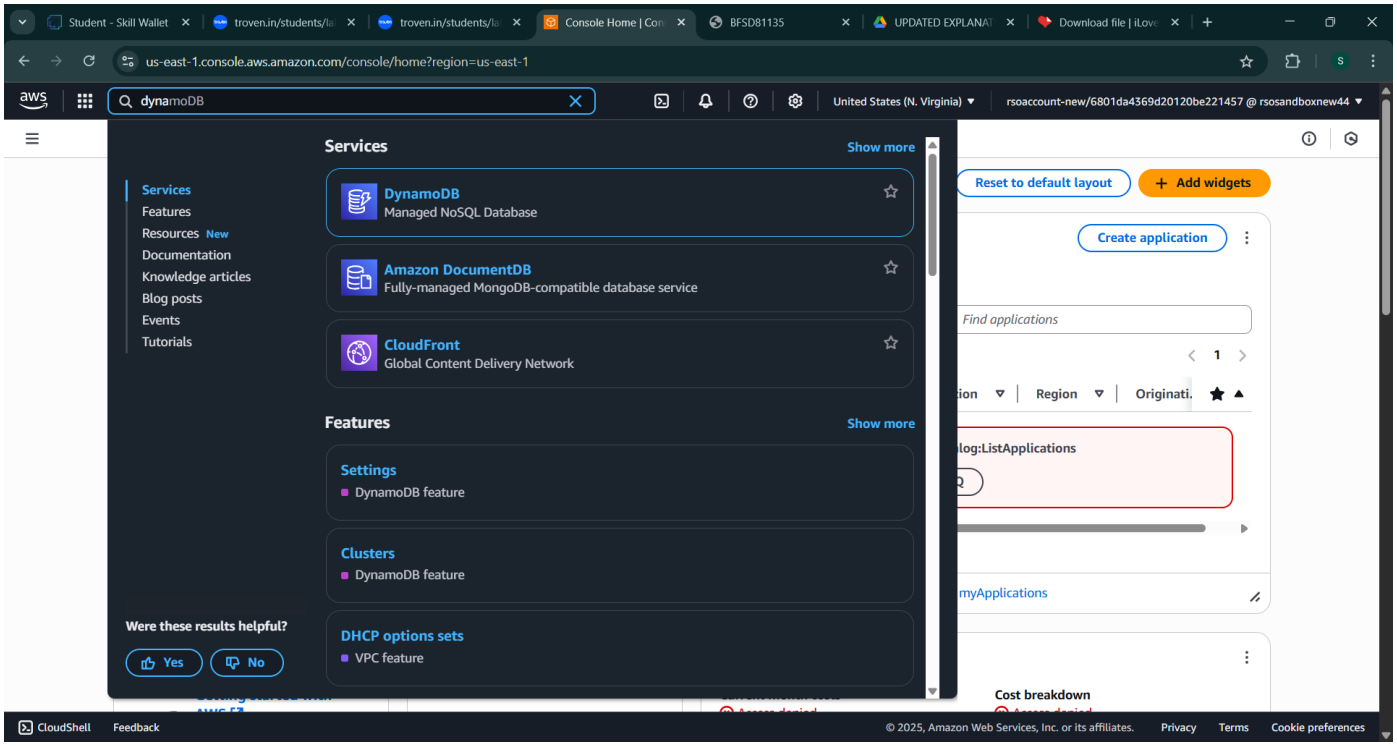
  -
  -
  -
  - .

Milestone 2 : AWS Account Setup



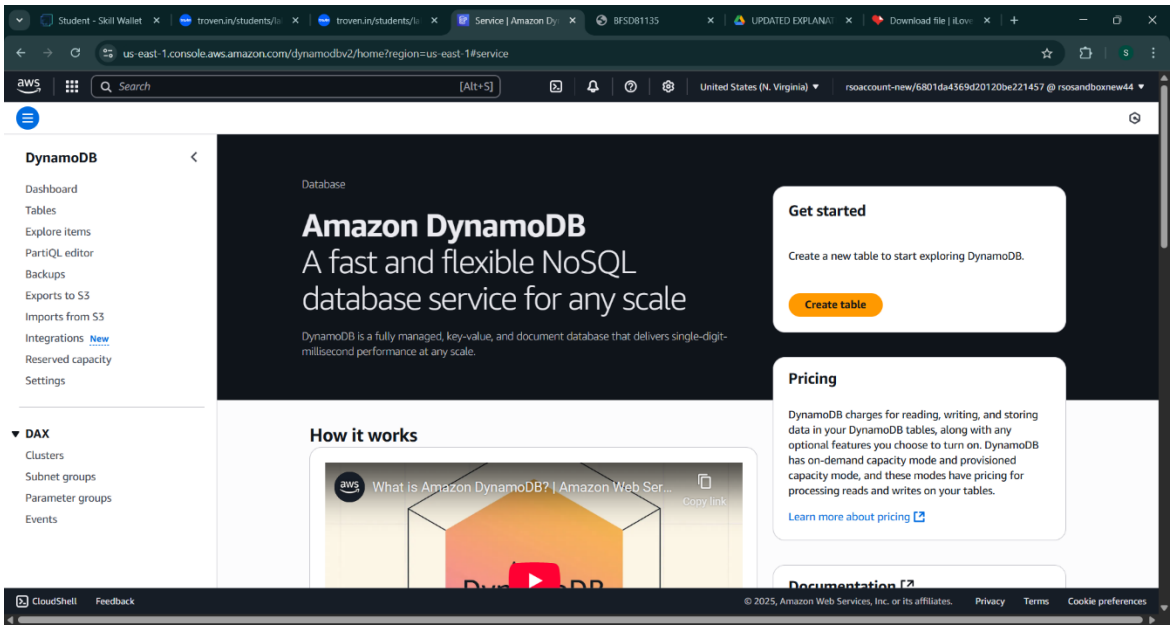
- **Activity 2.2: Log in to the AWS Management Console**
    - After setting up your account, log in to the AWS Management Console.





Milestone 3: DynamoDB Database Creation and Setup

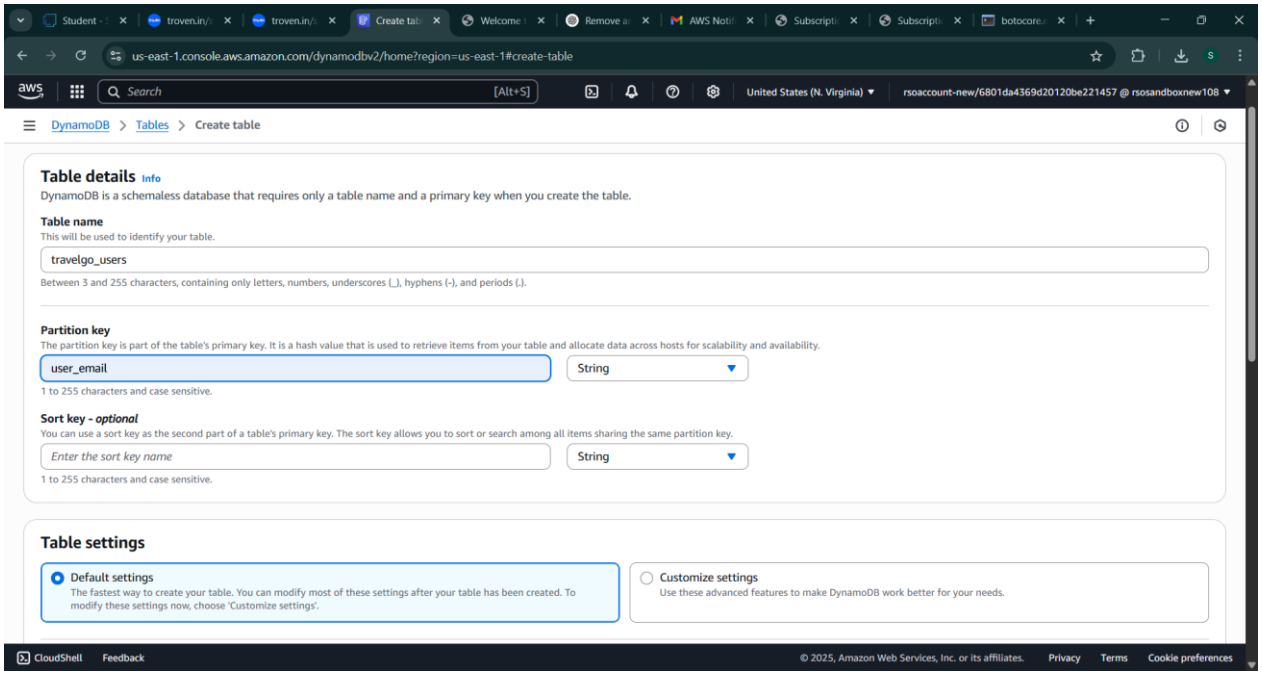
- **Activity 3.1:Navigate to the DynamoDB**
    - In the AWS Console, navigate to DynamoDB and click on create tables.



2.
3.
○

- **Activity 3.2:Create a DynamoDB table for storing registration details and book requests.**
    - Create Users table with partition key “user_email” with type String and click on create tables.

4.



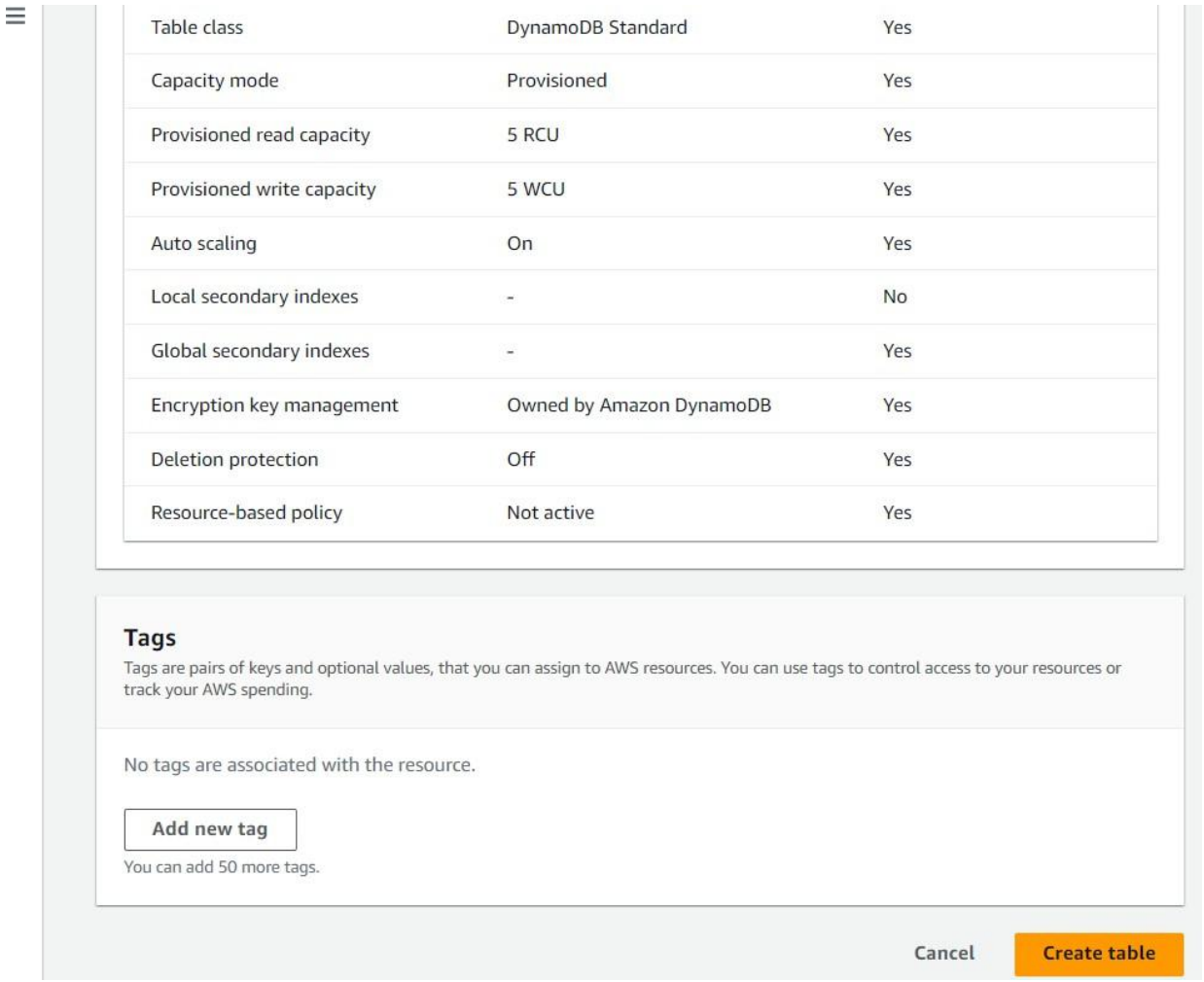
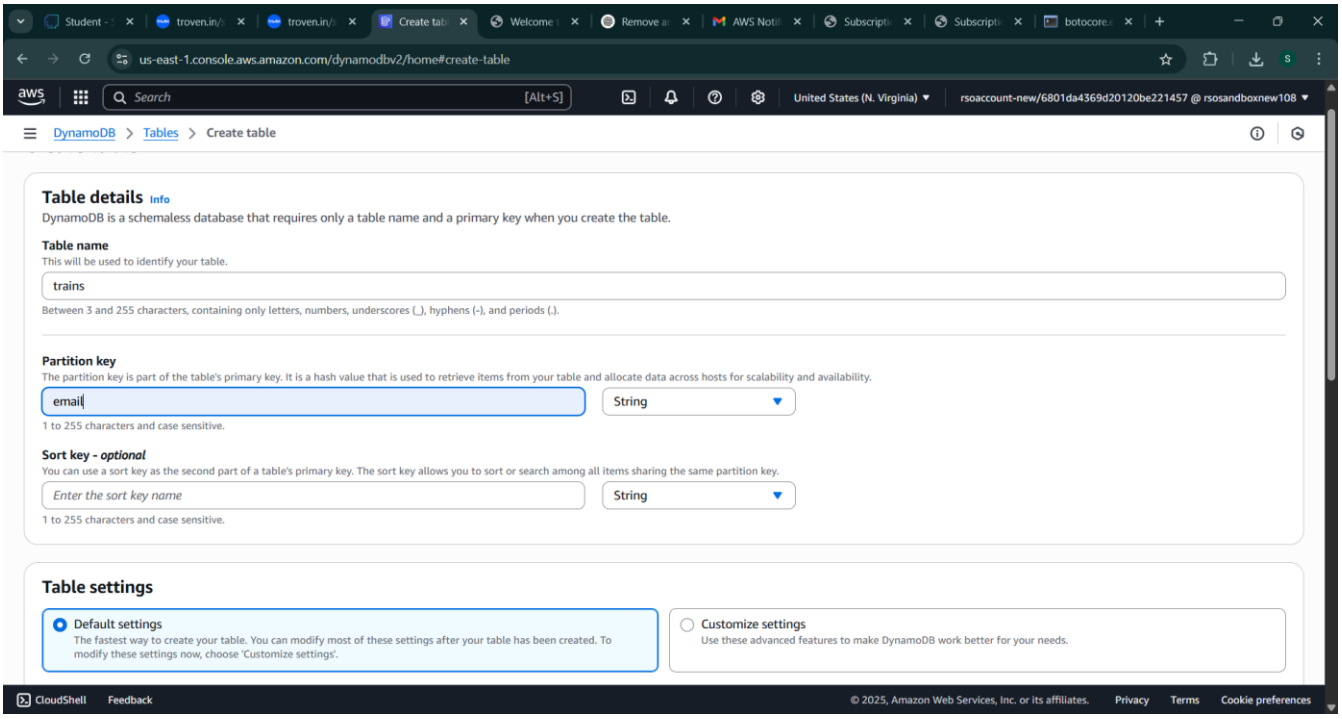
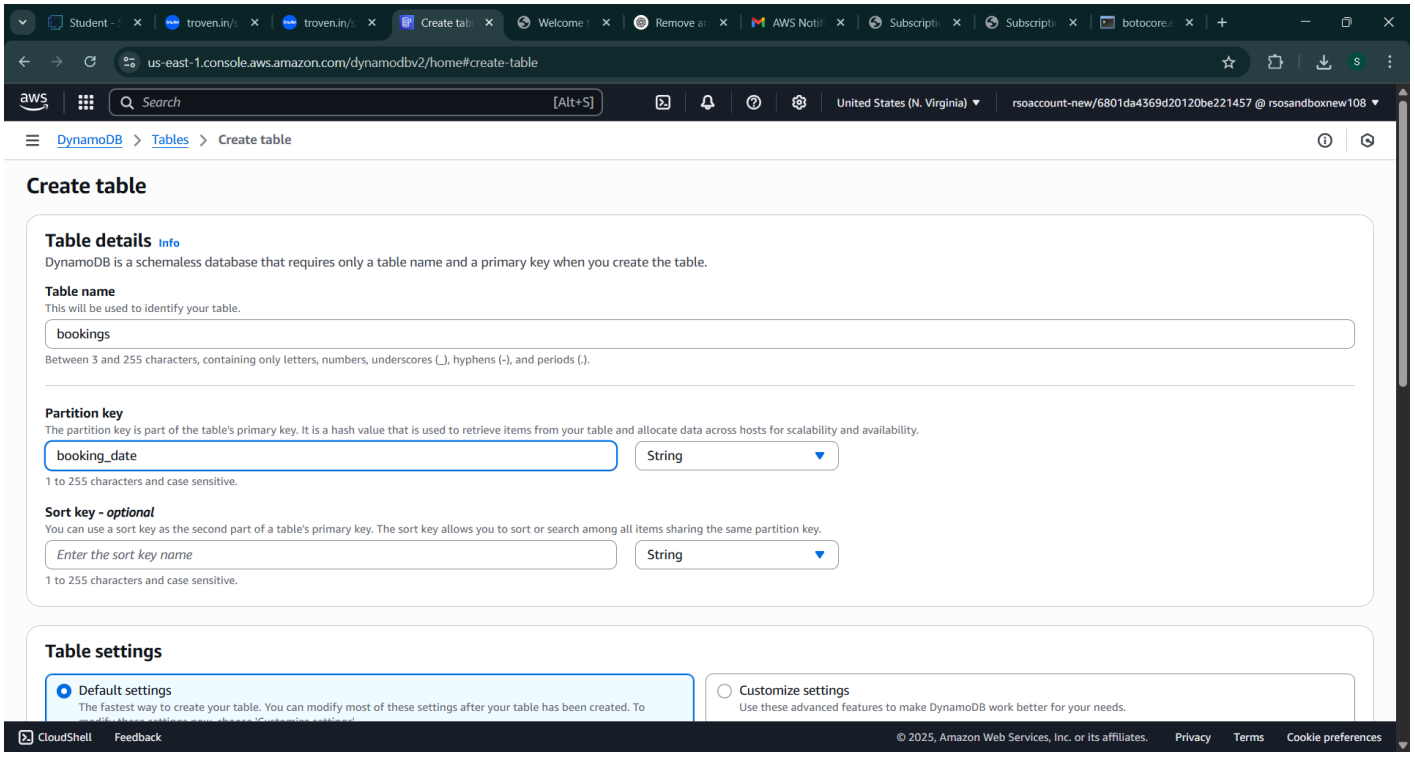
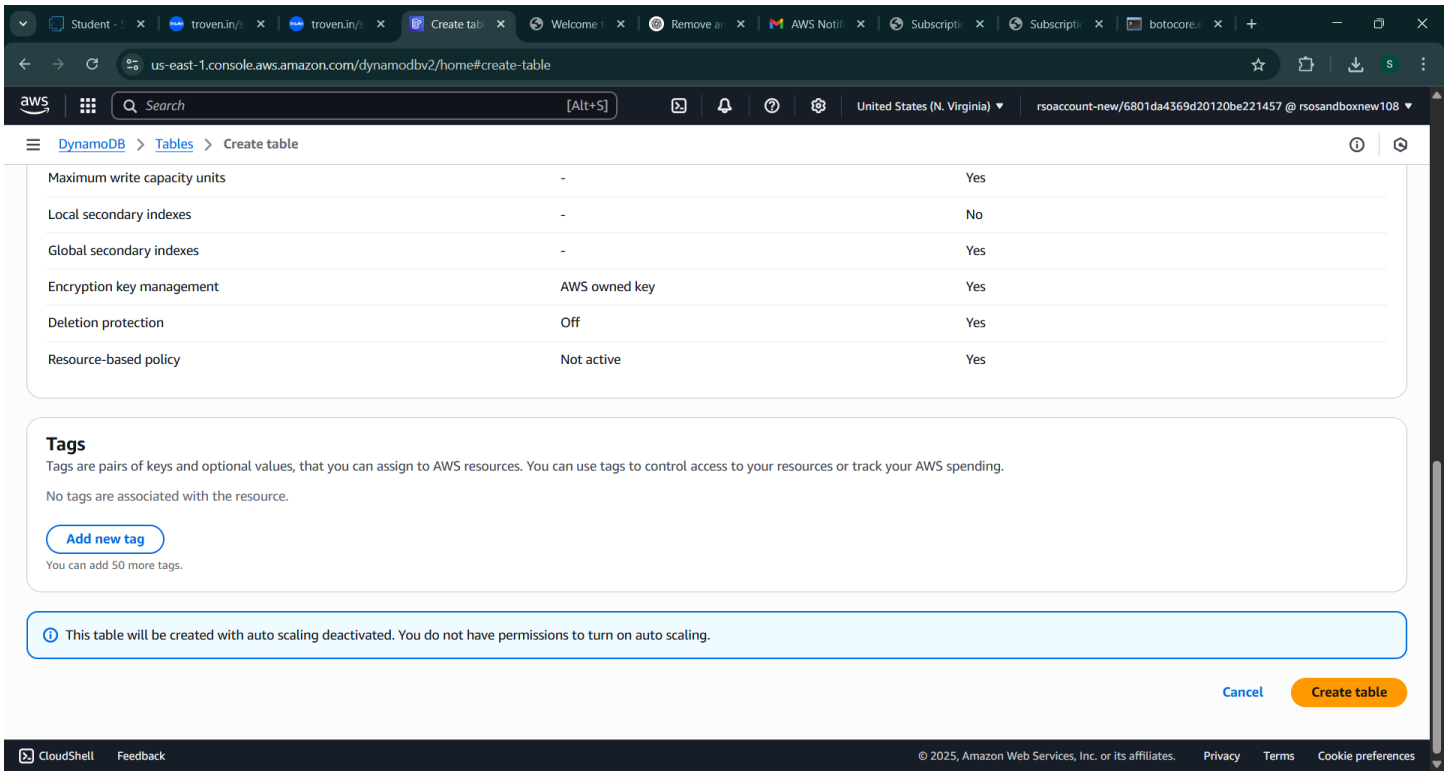


Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based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

- Follow the same steps to create a requests table with user_email as the primary key for book requests data.

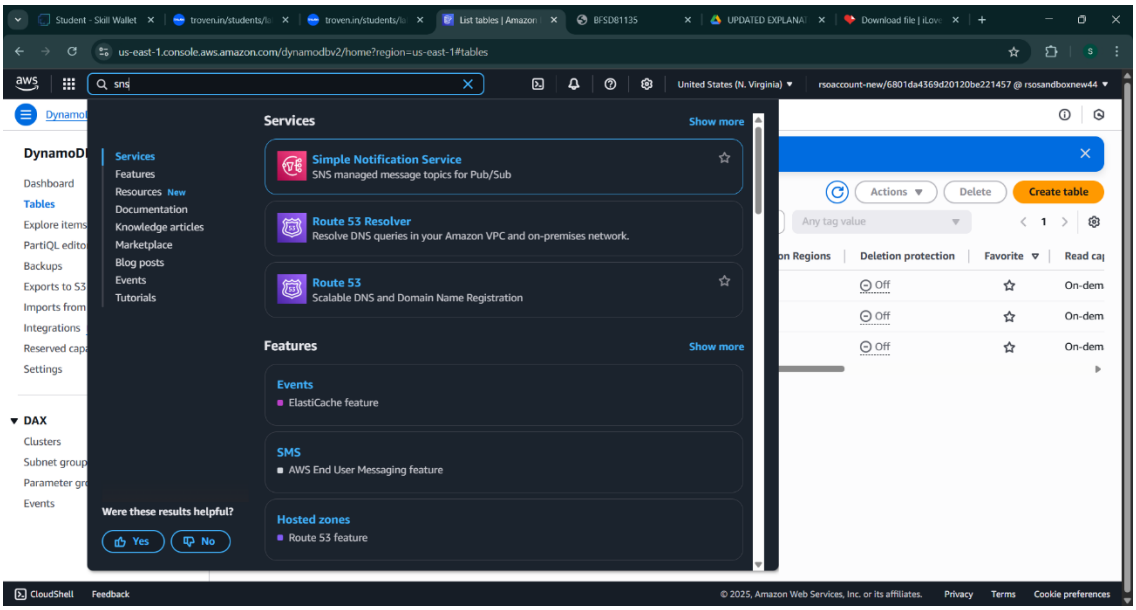

us-east-1.console.aws.amazon.com/dynamodbv2/home#create-table
DynamoDB > Tables > Create table
Table details Info
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.
Table name
This will be used to identify your table.
trains
Between 3 and 255 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
The fastest way to create your table. You can modify most of these settings after your table has been created. To modify these settings now, choose 'Customize settings'.
Customize settings
Use these advanced features to make DynamoDB work better for your needs.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/dynamodbv2/home#create-table
DynamoDB > Tables > Create table
Create table
Table details Info
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.
Table name
This will be used to identify your table.
bookings
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.
booking_date
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
The fastest way to create your table. You can modify most of these settings after your table has been created. To
Customize settings
Use these advanced features to make DynamoDB work better for your needs.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

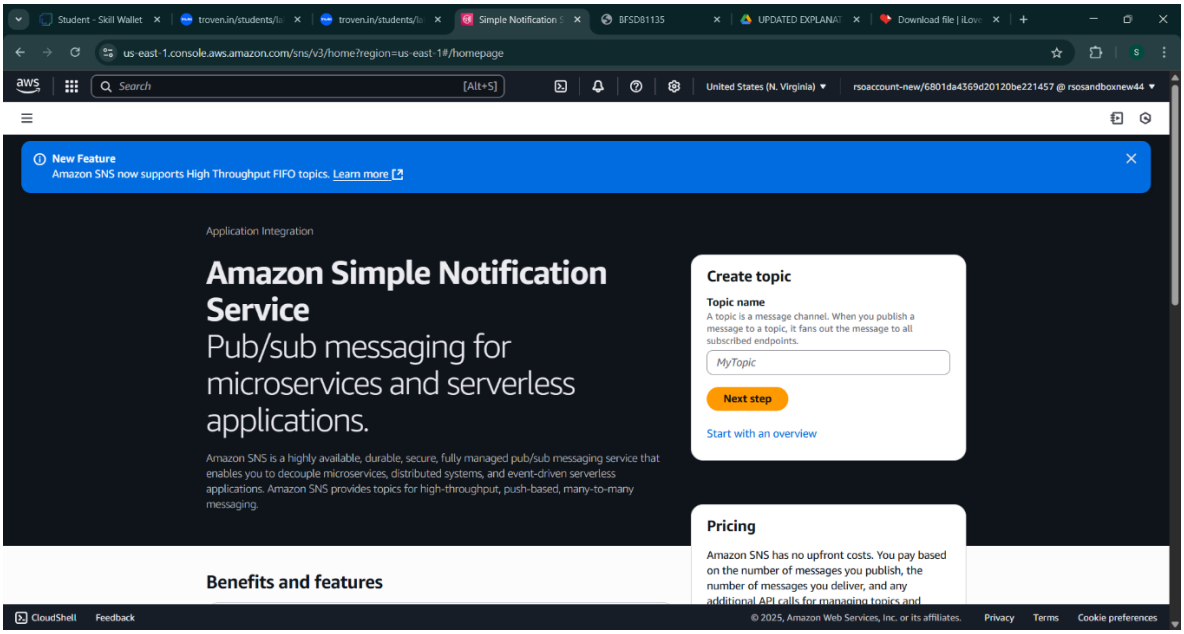


Milestone 4: SNS Notification Setup

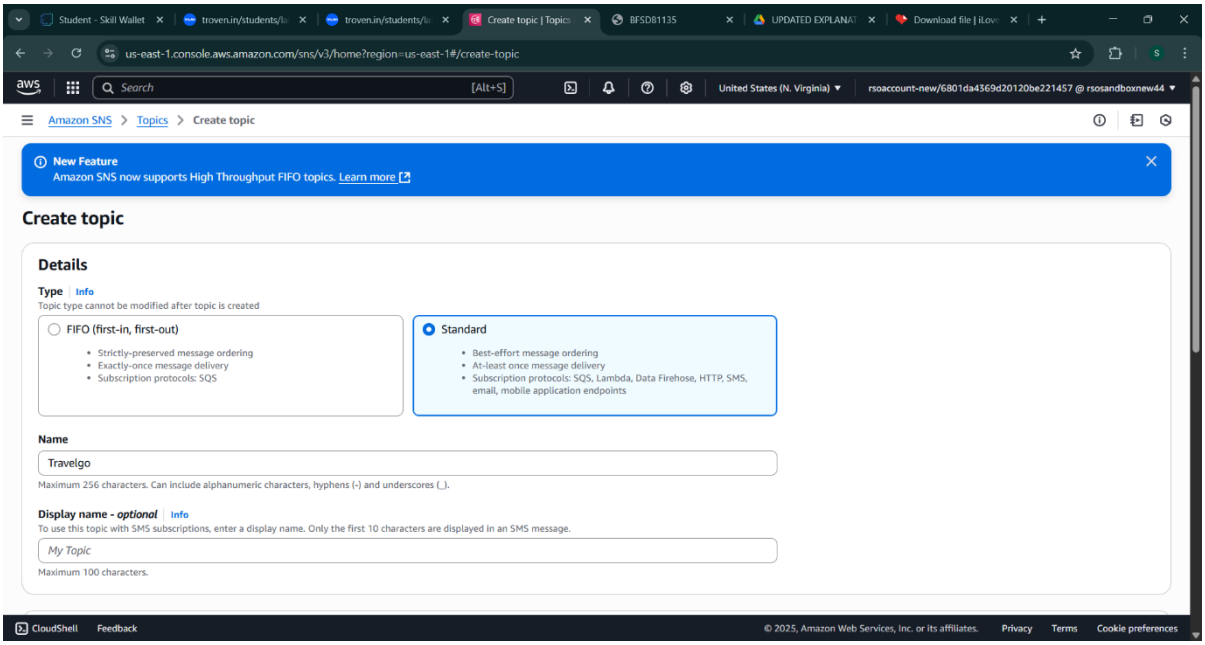
- **Activity 4.1: Create SNS topics for sending email notifications to users**



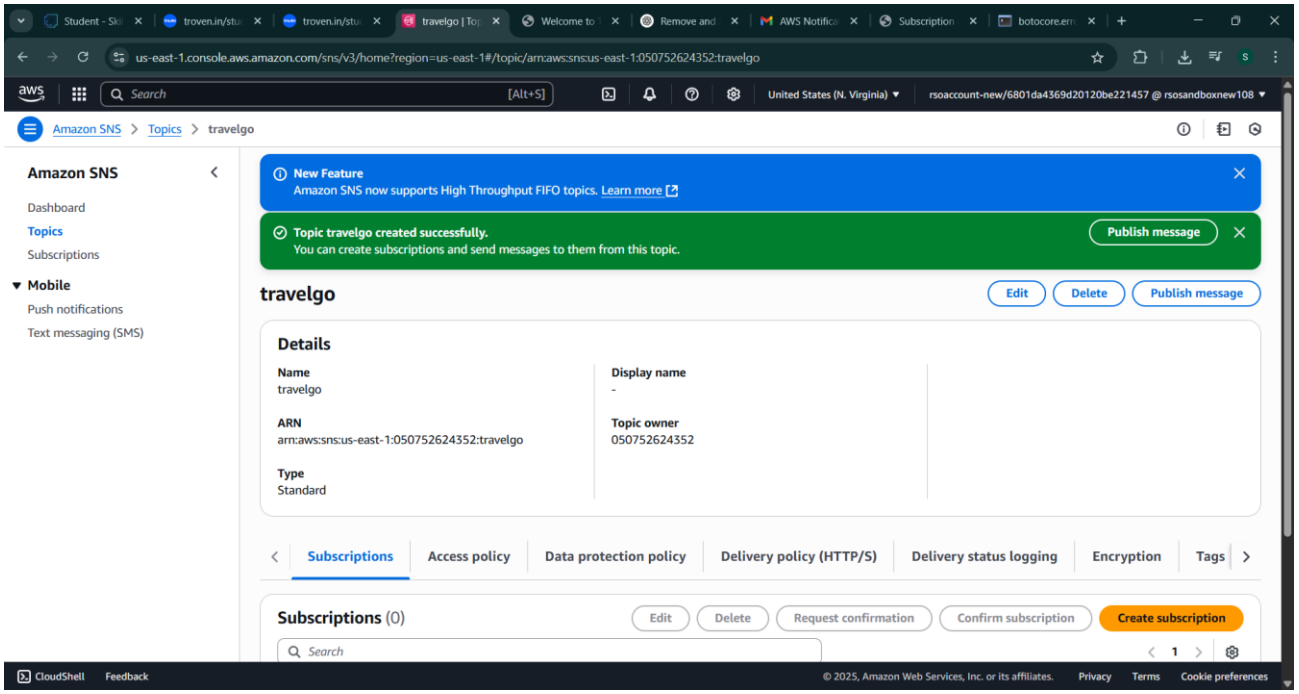
- In the AWS Console, search for SNS and navigate to the SNS Dashboard.



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

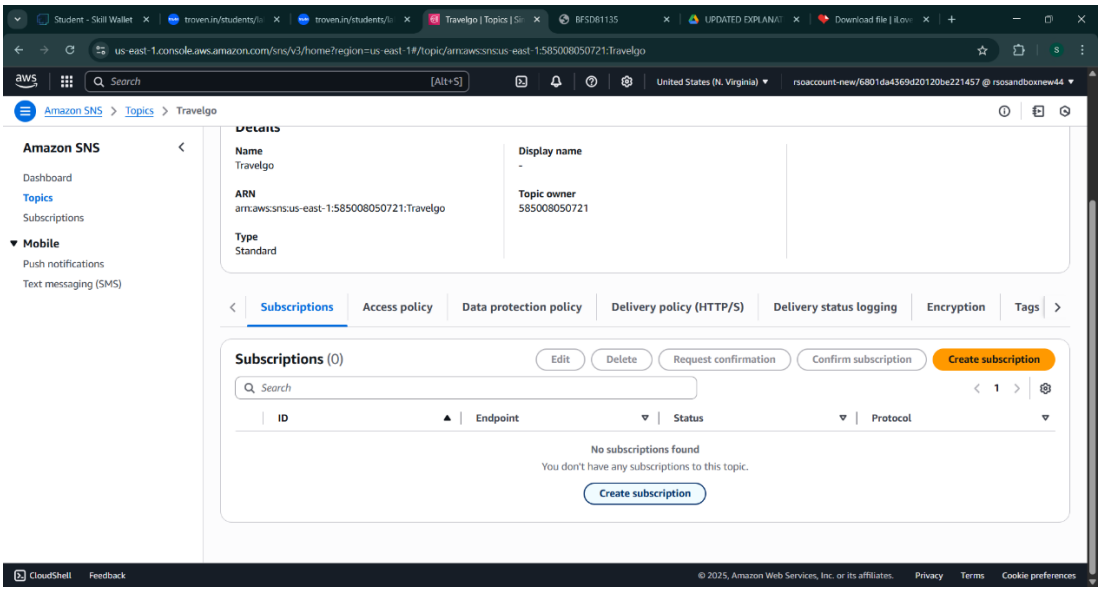


- Choose Standard type for general notification use cases and Click on Create Topic.

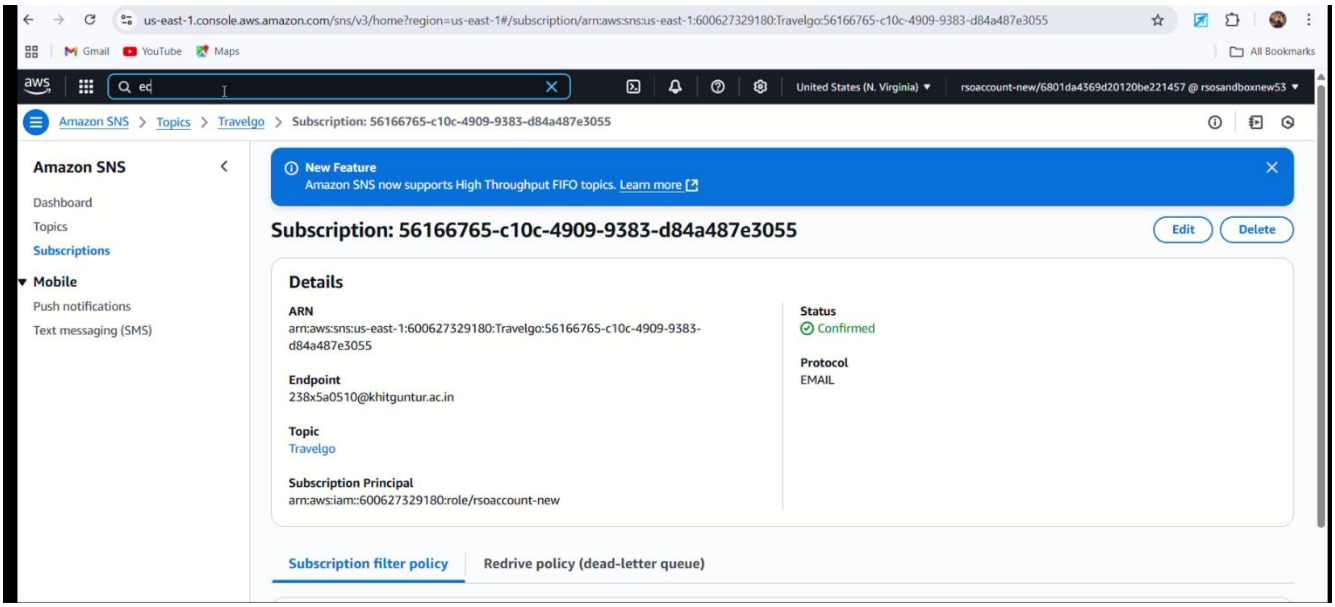


- Configure the SNS topic and note down the **Topic ARN**.

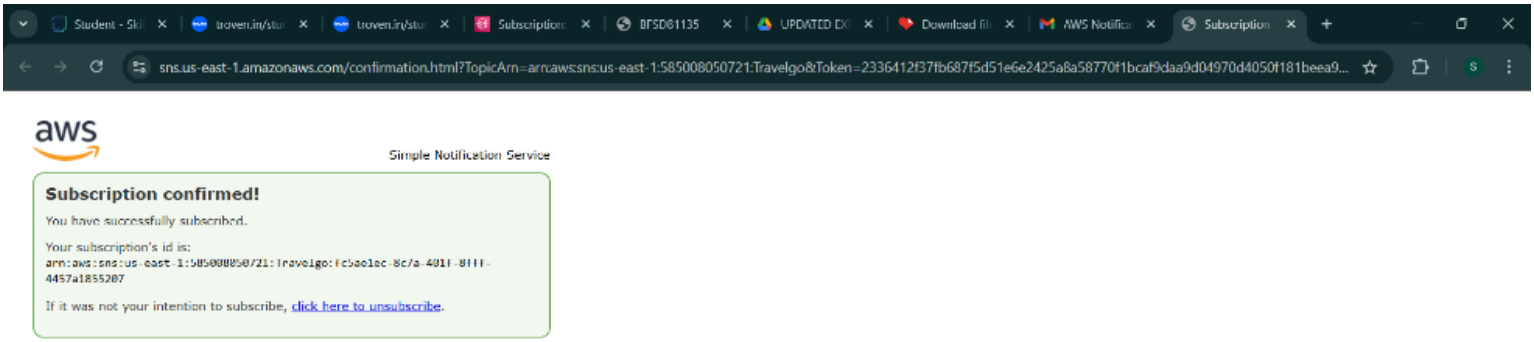
- **Activity 4.2: Subscribe users relevant SNS topics to receive real-time notifications when a book request is made.**



- Subscribe users to this topic via Email. When a book request is made, notifications will be sent to the subscribed emails.



After subscription request for the mail confirmation

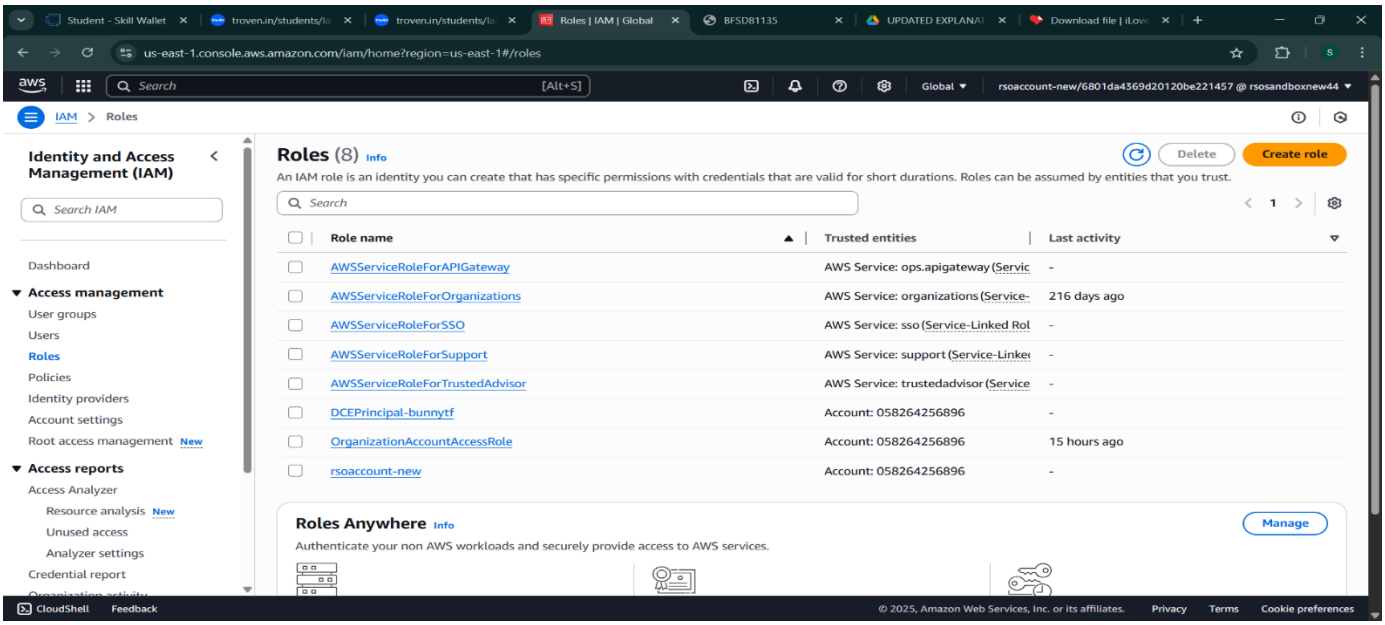


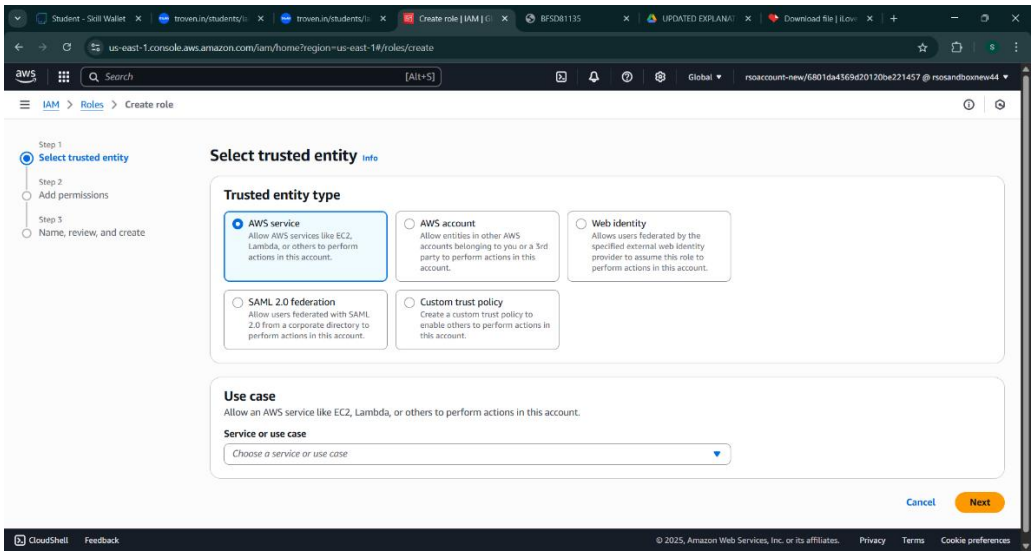
Successfully done with the SNS mail subscription and setup, now store the ARN link

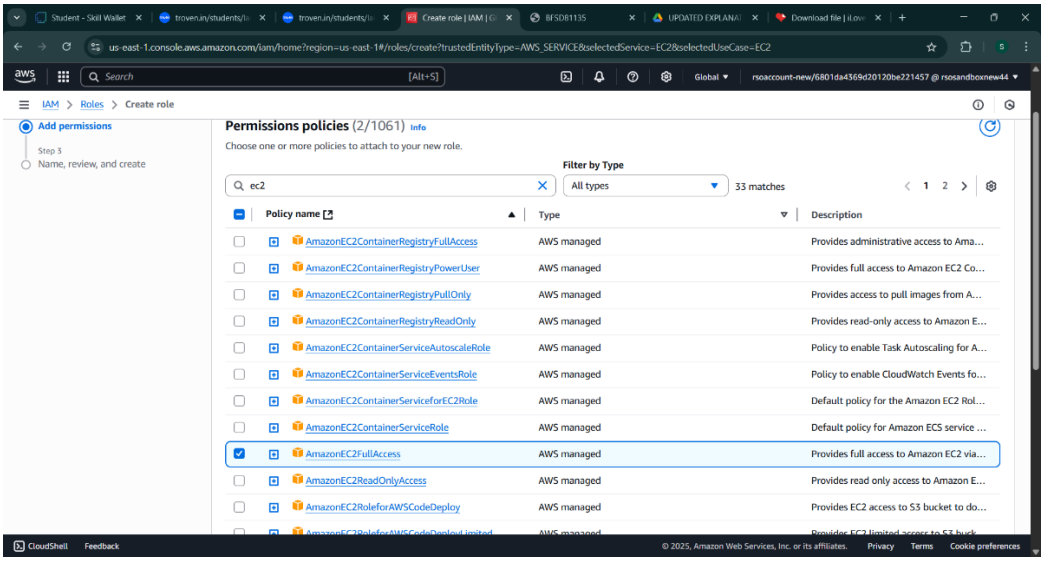
.

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.
-



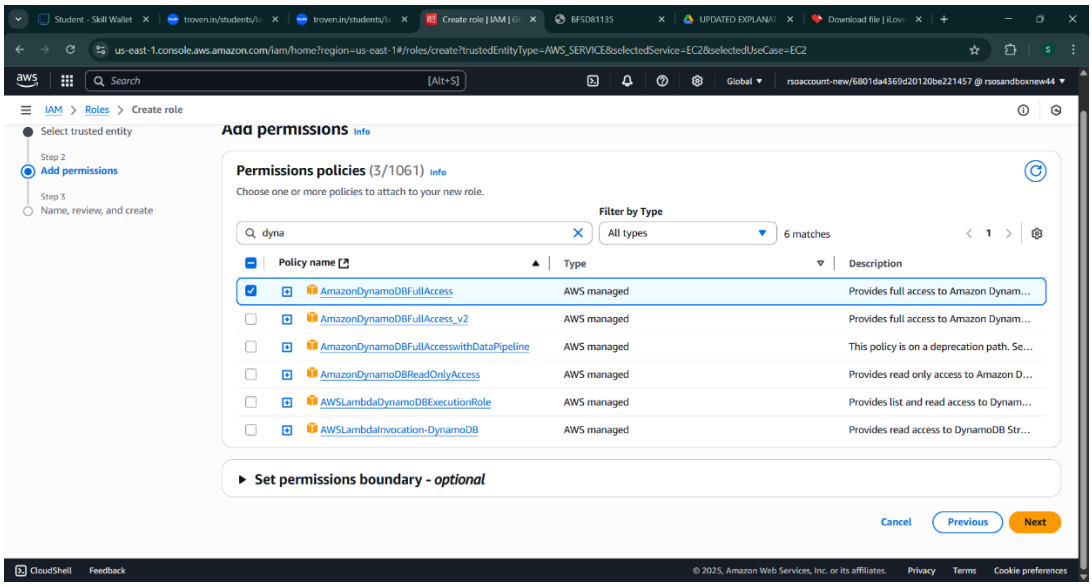


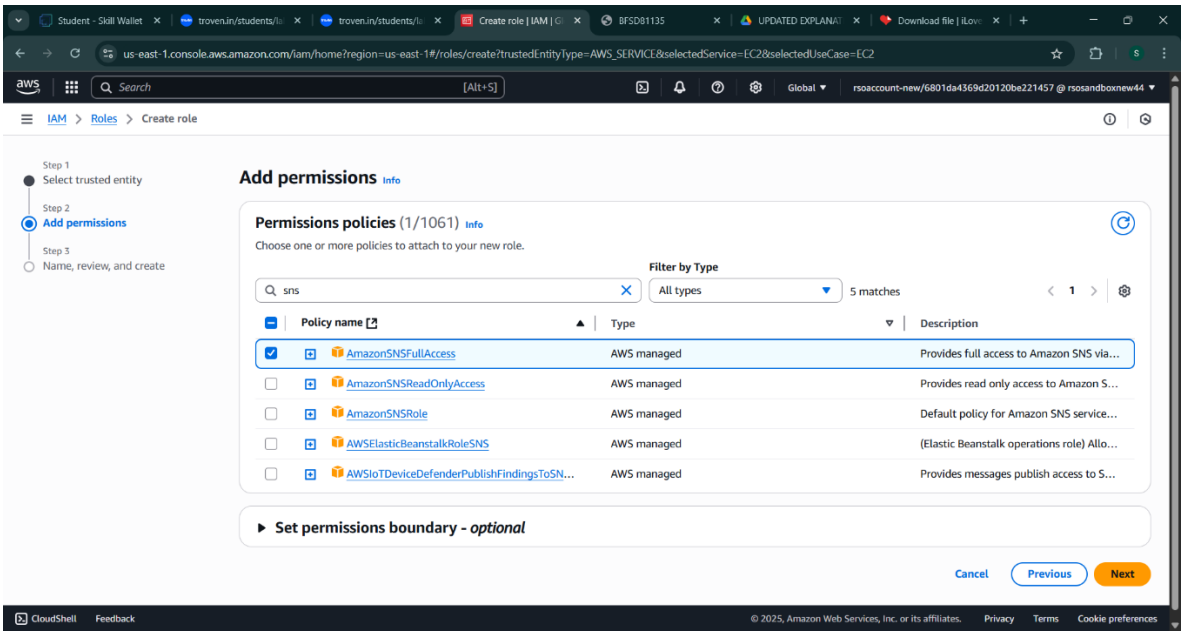


- **Activity 5.2: Attach Policies.**

  Attach the following policies to the role:

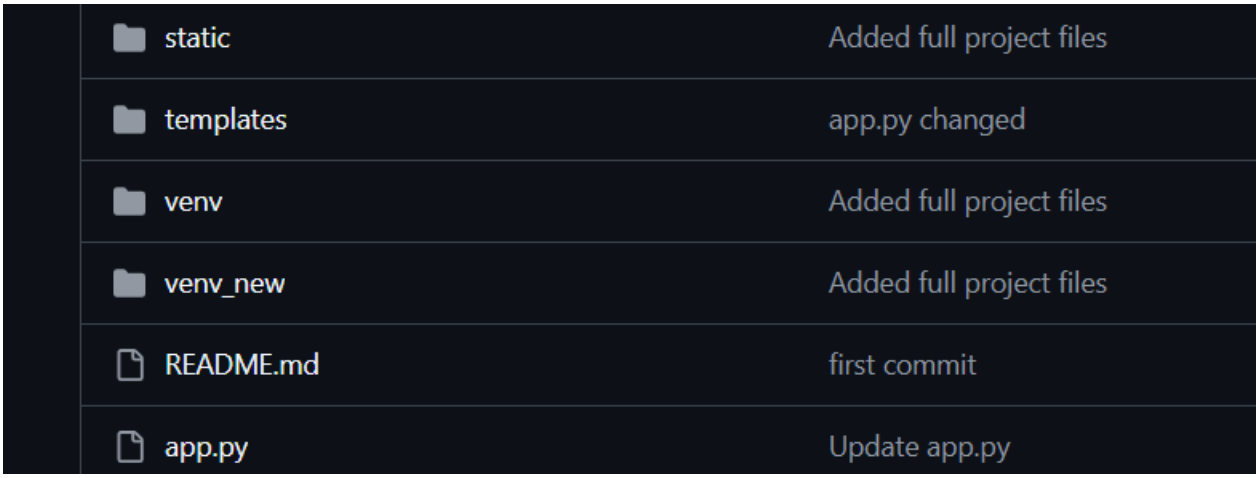
  - AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on DynamoDB.
  - AmazonSNSFullAccess: Grants EC2 the ability to send notifications via SNS.


us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/roles/create?trustedEntityType=AWS_SERVICE&selectedService=EC2&selectedUseCase=EC2
IAM > Roles > Create role
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Add permissions Info
Permissions policies (3/1061) Info
Choose one or more policies to attach to your new role.
Filter by Type
dyna
All types
6 matches
Policy name
Type
Description
AmazonDynamoDBFullAccess
AWS managed
Provides full access to Amazon Dynam…
AmazonDynamoDBFullAccess_v2
AWS managed
Provides full access to Amazon Dynam…
AmazonDynamoDBFullAccesswithDataPipeline
AWS managed
This policy is on a deprecation path. Se…
AmazonDynamoDBReadOnlyAccess
AWS managed
Provides read only access to Amazon D…
AWSLambdaDynamoDBExecutionRole
AWS managed
Provides list and read access to Dynam…
AWSLambdaInvocation-DynamoDB
AWS managed
Provides read access to DynamoDB Str…
Set permissions boundary - optional
Cancel
Previous
Next
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/roles/create?trustedEntityType=AWS_SERVICE&selectedService=EC2&selectedUseCase=EC2
IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Add permissions Info
Permissions policies (1/1061) Info
Choose one or more policies to attach to your new role.
Filter by Type
sns
All types
5 matches
Policy name
Type
Description
AmazonSNSFullAccess
AWS managed
Provides full access to Amazon SNS via…
AmazonSNSReadOnlyAccess
AWS managed
Provides read only access to Amazon S…
AmazonSNSRole
AWS managed
Default policy for Amazon SNS service…
AWSElasticBeanstalkRoleSNS
AWS managed
(Elastic Beanstalk operations role) Allo…
AWSIoTDeviceDefenderPublishFindingsToSN…
AWS managed
Provides messages publish access to S…
Set permissions boundary - optional
Cancel
Previous
Next
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

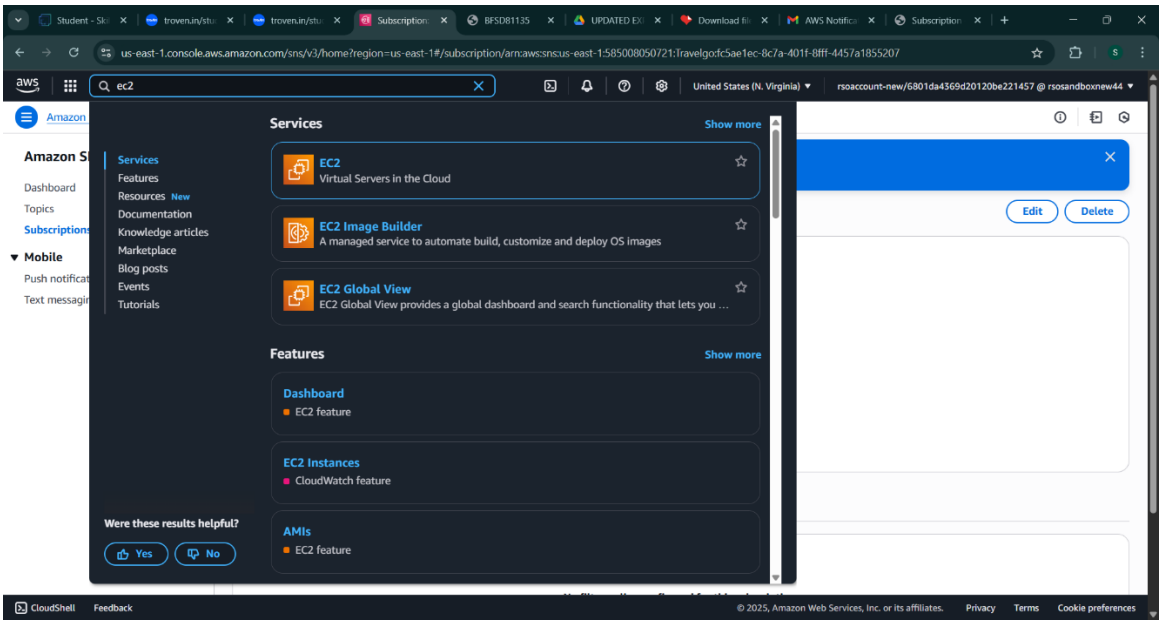
Milestone 6: EC2 Instance Setup

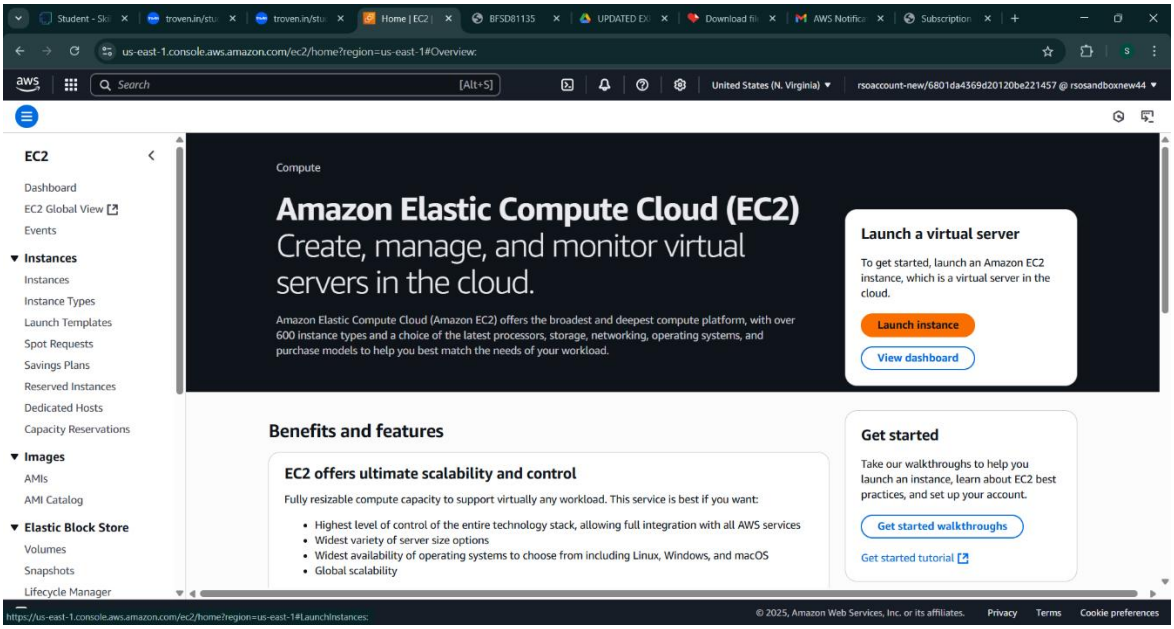
- **Note: Load your Flask app and Html files into GitHub repository.**



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

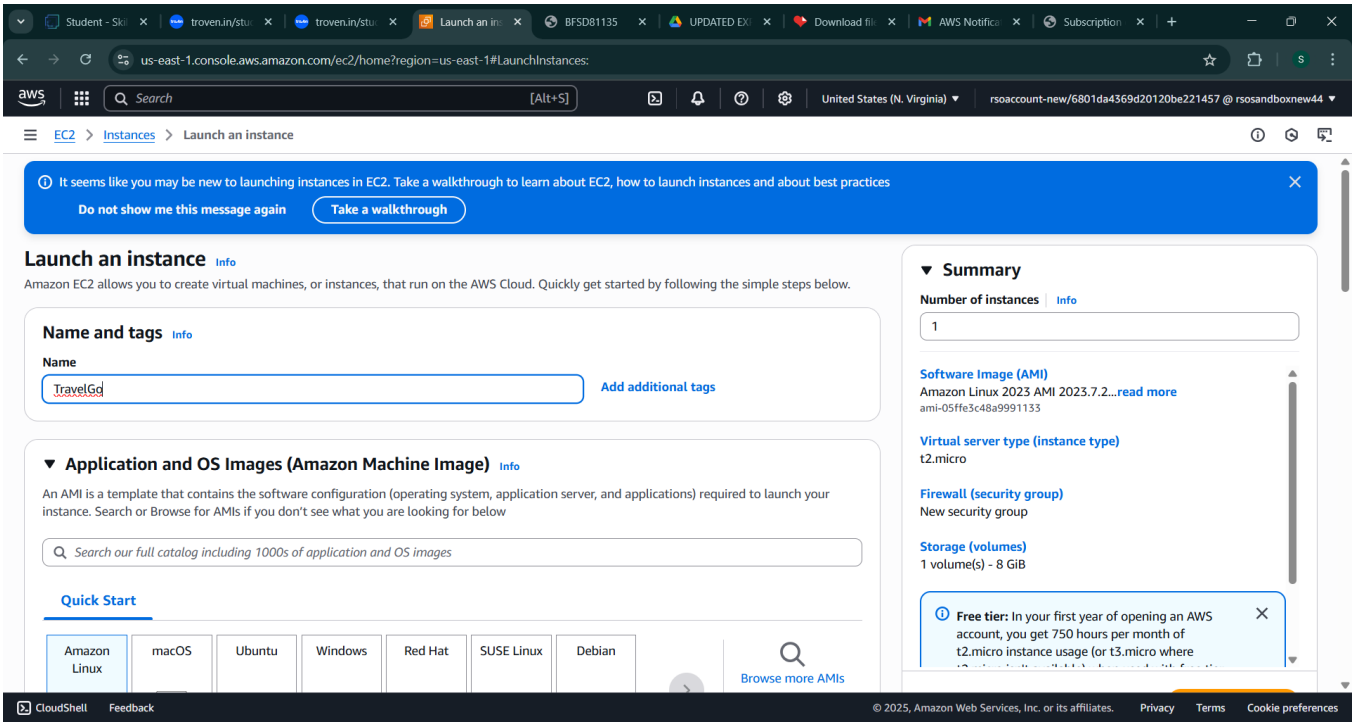
    - **Launch EC2 Instance**
        - In the AWS Console, navigate to EC2 and launch a new instance.



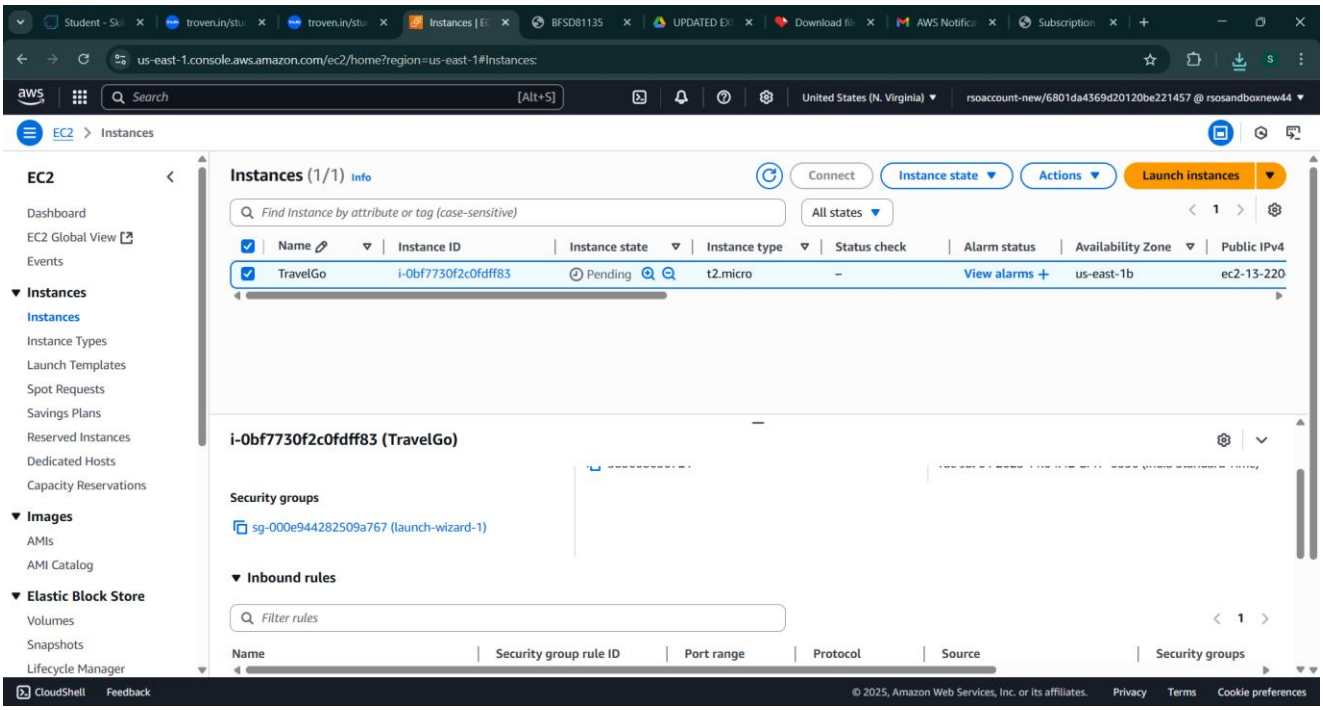


- Click on Launch instance to launch EC2 instance

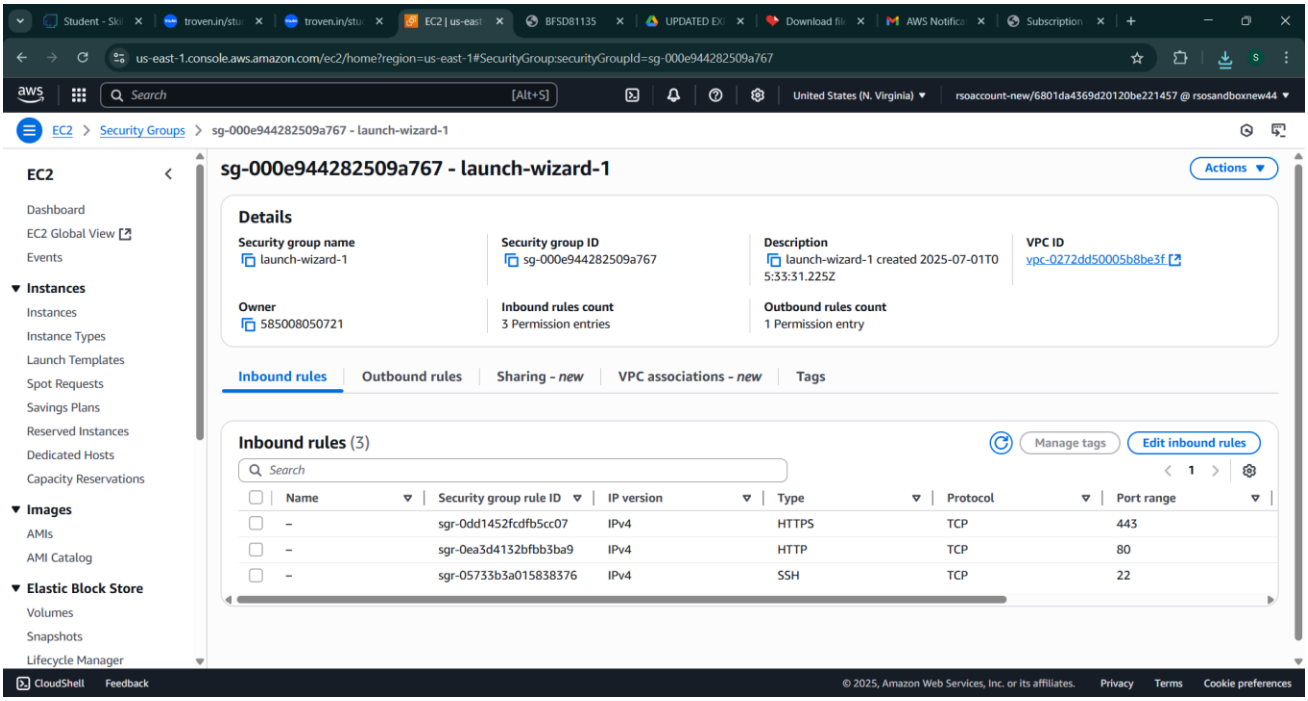
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).



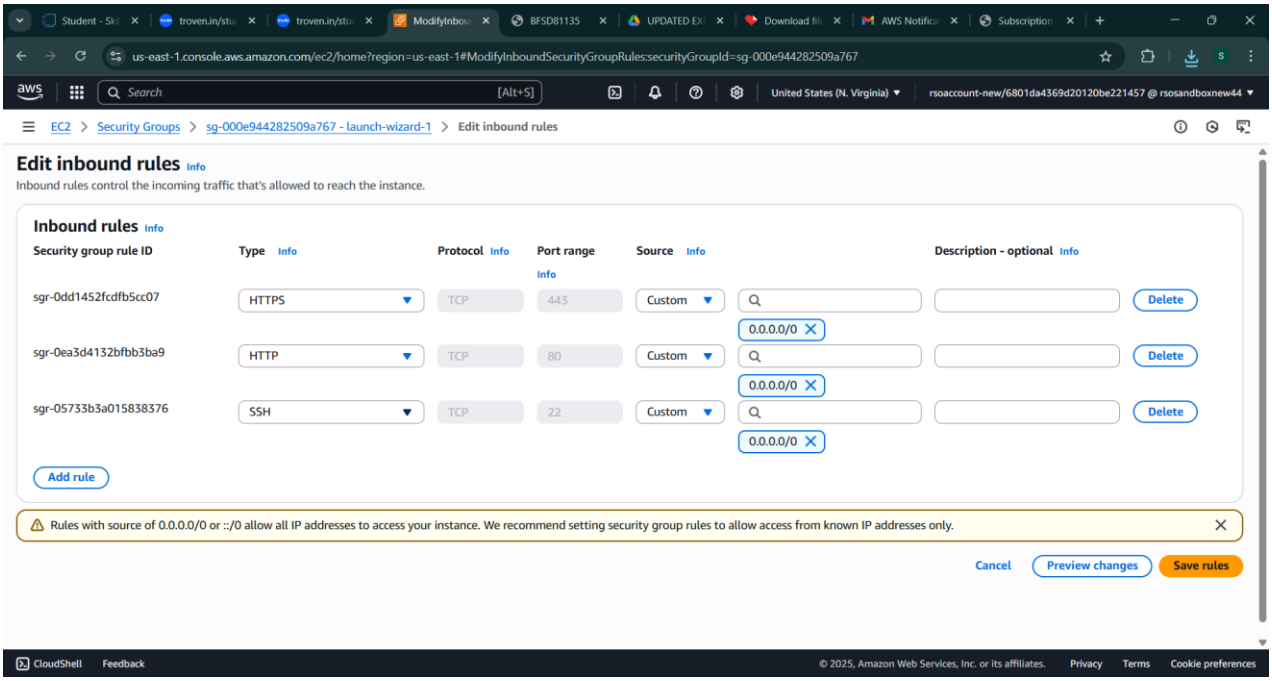
- Create and download the key pair for Server access.

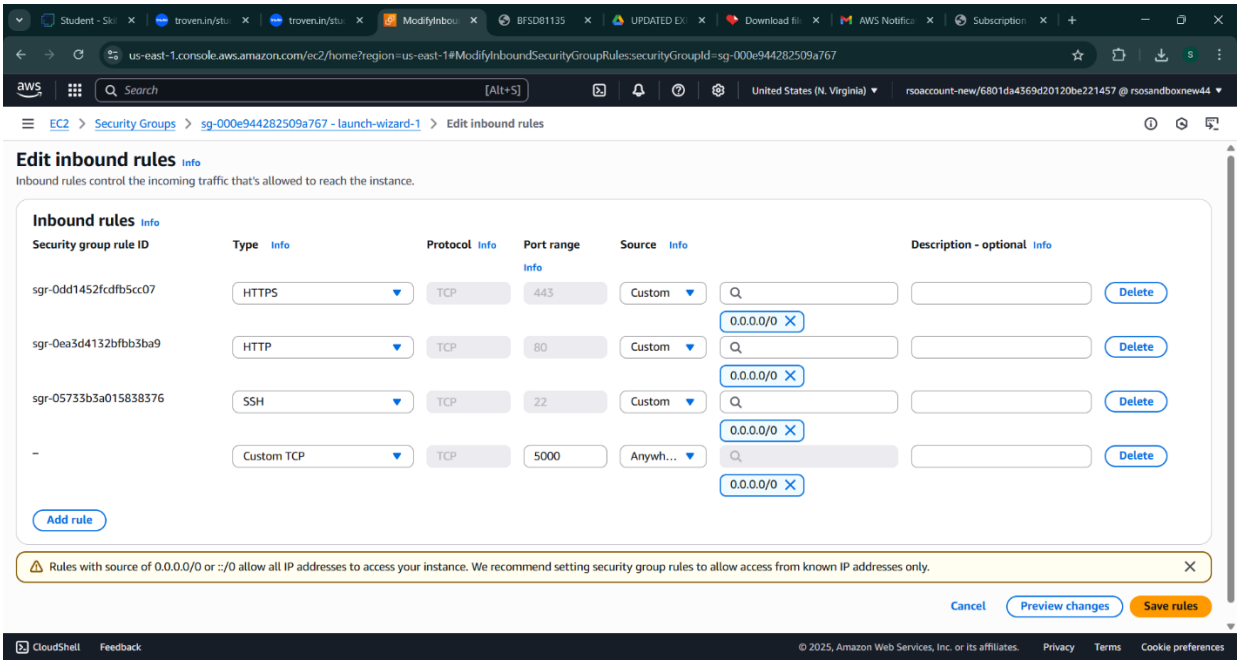


- **Activity 6.2:Configure security groups for HTTP, and SSH access.**

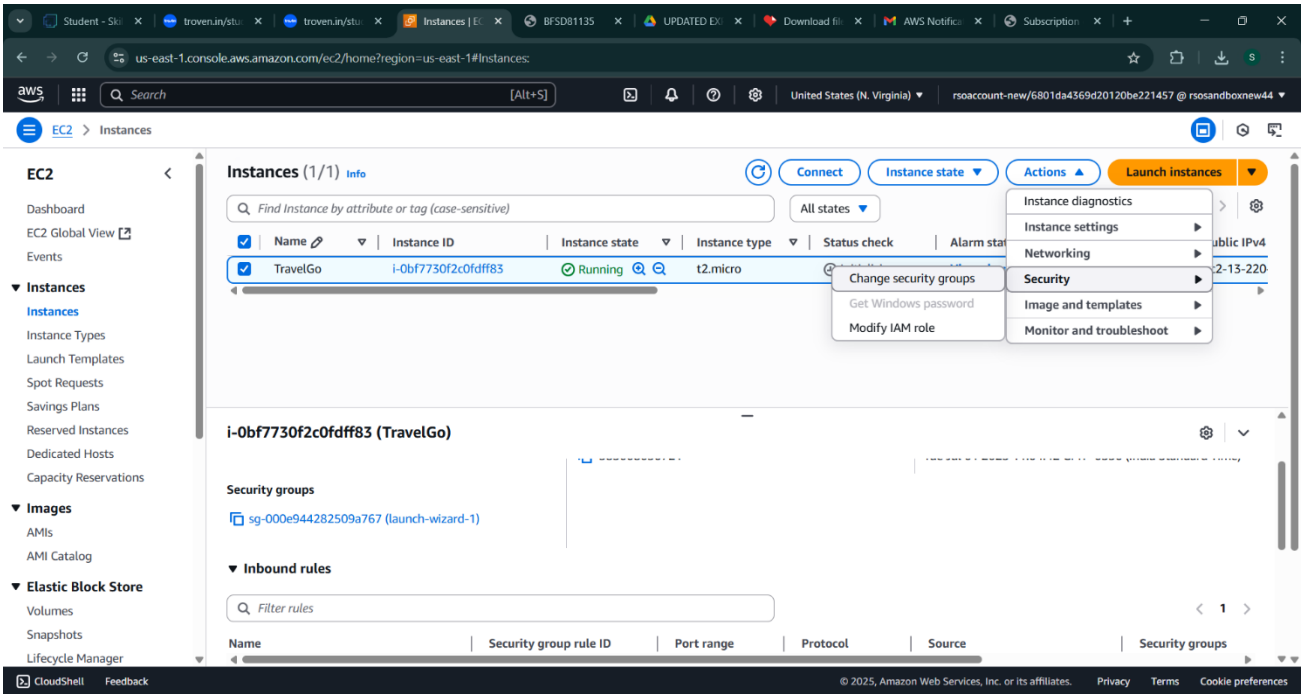


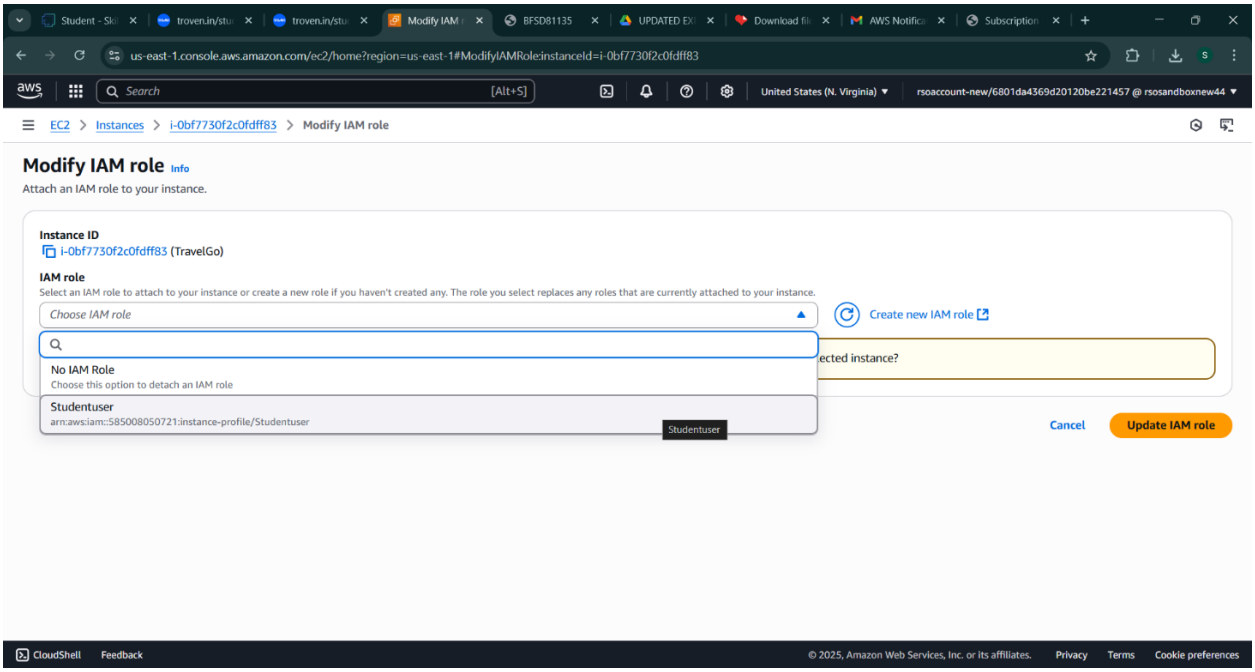
- **Activity 6.2:Configure security groups for HTTP, and SSH access.**



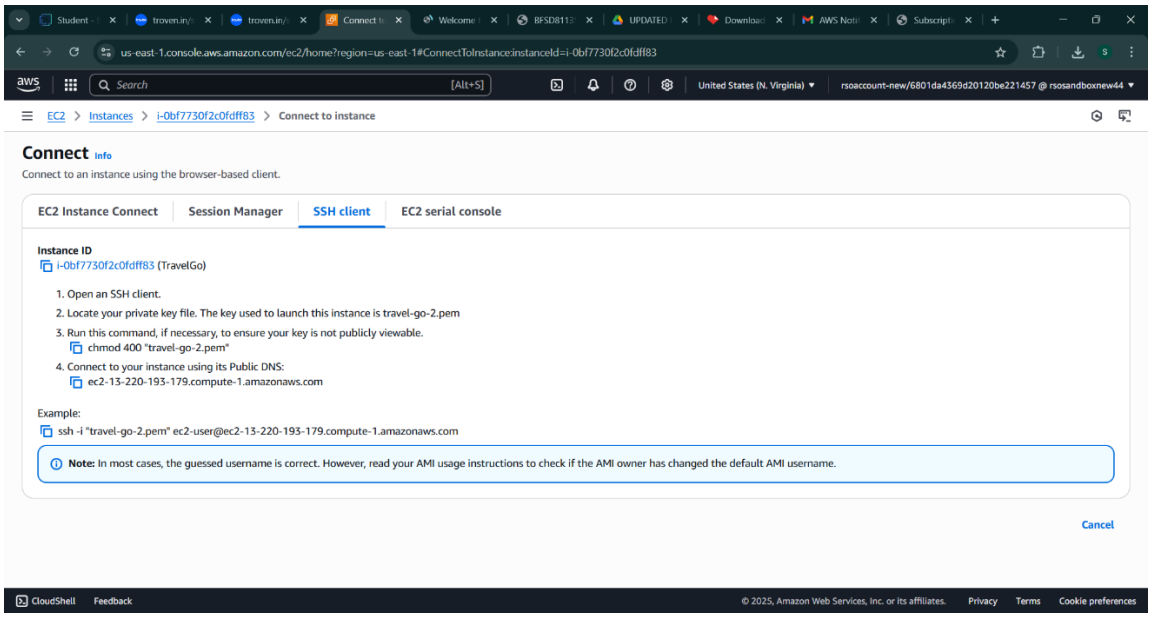


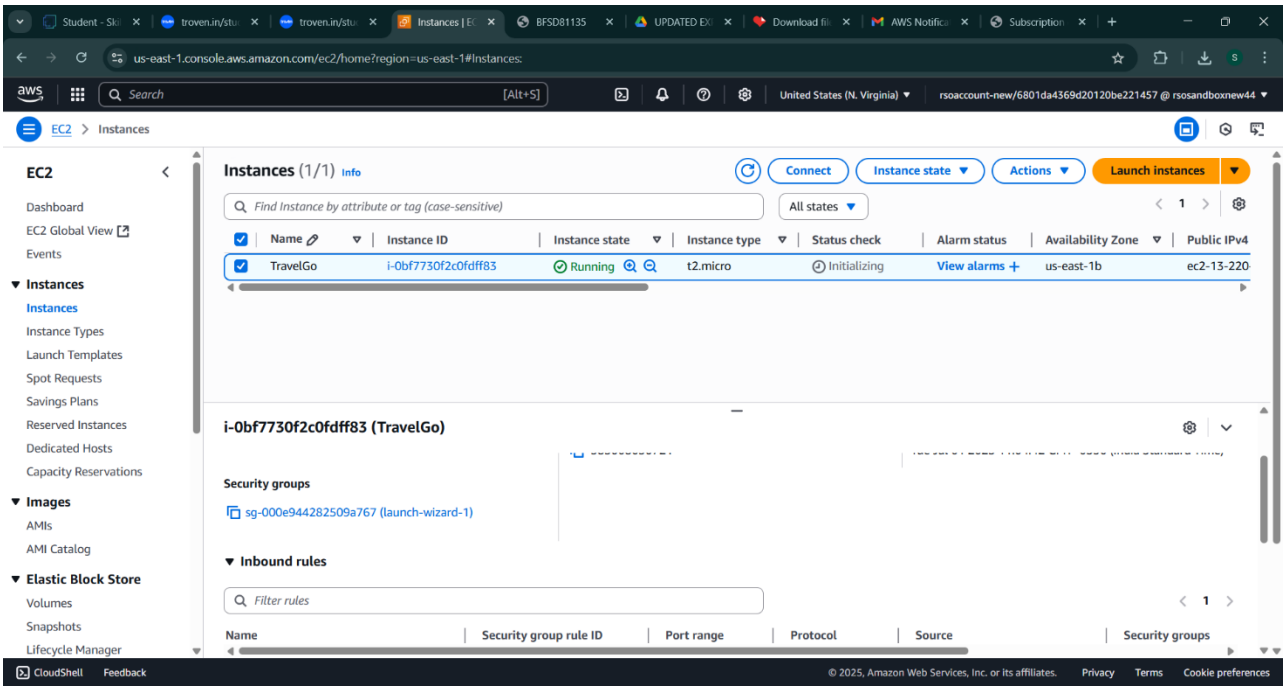
- To connect to EC2 using **EC2 Instance Connect**, start by ensuring that an **IAM role** is attached to your EC2 instance. You can do this by selecting your instance, clicking on **Actions**, then navigating to **Security** and selecting **Modify IAM Role** to attach the appropriate role. After the 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.





- Now connect the EC2 with the files





Milestone 7: Deployment on EC2

Activity 7.1: Install Software on the EC2 Instance

Install Python3, Flask, and Git:

On Amazon Linux 2:

```
sudo yum 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/your-github-username/your-repository-name.git

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

here: 'git clone https://github.com/AlekhyaPenubakula/InstantLibrary.git'

- This will download your project to the EC2 instance.

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

```
cd InstantLibrary
```

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

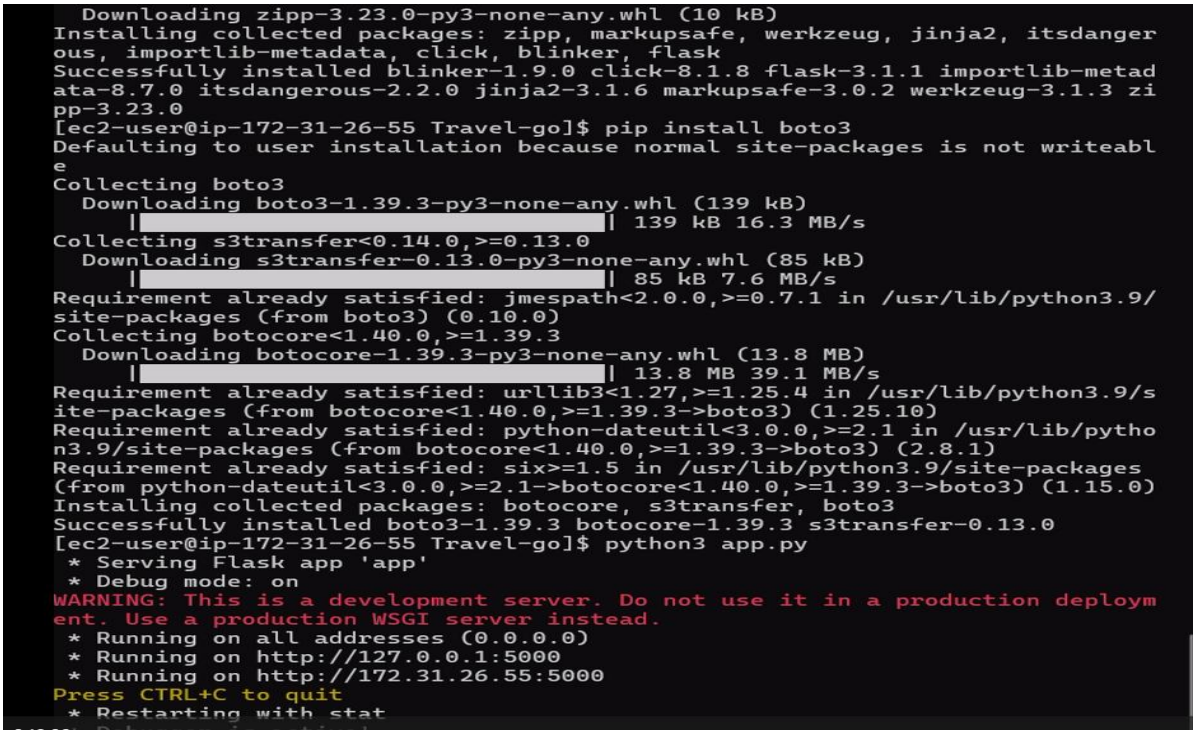
Run the Flask Application

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

Verify the Flask app is running:

http://your-ec2-public-ip

- Run the Flask app on the EC2 instance



Conclusion

The Movie Magic website has been successfully developed and deployed using a robust, cloud-native architecture. By leveraging key AWS services such as EC2 for hosting, DynamoDB for real-time data management, and SNS for instant booking confirmations, the platform delivers a scalable and user-friendly movie ticketing experience. This solution addresses the limitations of traditional ticket booking systems by enabling users to seamlessly search for movies, select seats, and receive digital tickets—all from a single platform.

The cloud-based infrastructure ensures that the system can handle a high volume of users and transactions, especially during peak movie release times, without compromising speed or reliability. The integration of Flask with AWS services enables smooth backend operations, including user authentication, movie/event browsing, seat availability checks, and secure ticket booking.

Comprehensive testing has confirmed that all core functionalities—from user registration and login to booking confirmation via email—operate seamlessly. The platform's modern interface and responsive design further enhance the user experience, making movie booking quick, efficient, and enjoyable.

In conclusion, Movie Magic stands as a powerful demonstration of how cloud technologies can modernize traditional services. It provides an efficient, scalable, and intuitive solution for movie ticket booking, showcasing the potential of full-stack cloud applications in solving real-world user experience challenges in the entertainment industry.

SMARTBRIDGE
Let's Bridge the Gap

Smart Internz

THANK YOU