

## **Movie Magic—Smart Movie Ticket Booking System**

### **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.

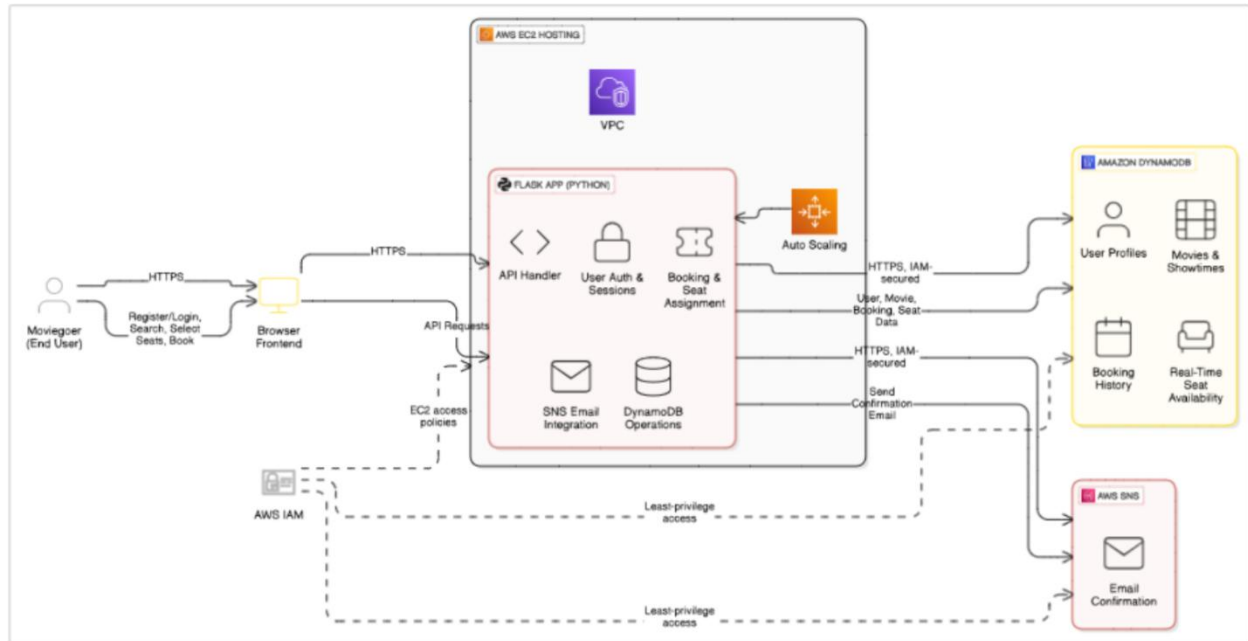
#### **Scenario 2: Seamless Booking Confirmation Notifications**

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.

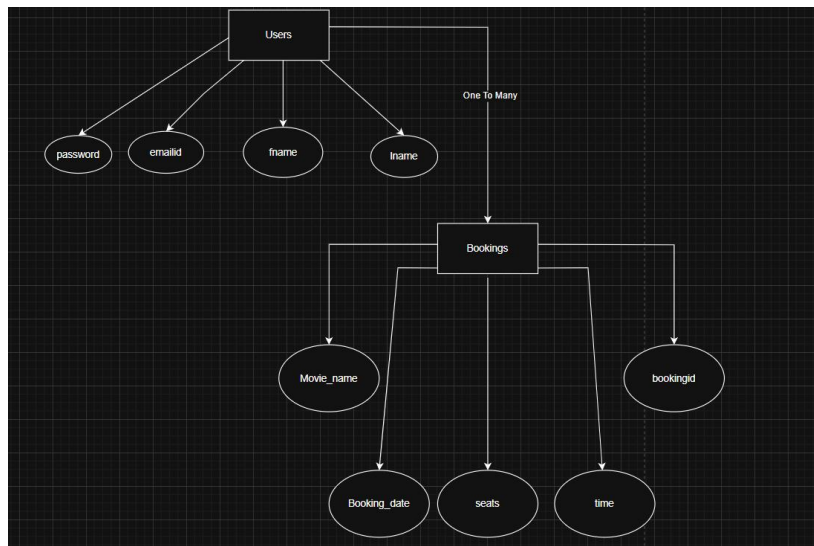
#### **Scenario 3: Easy Access to Movies and Events**

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



## ER Diagram



## Pre-requisites:

- AWS Account Setup : For More Details [Click Here](#)
- AWS IAM (Identity and Access Management) : For More Details [Click Here](#)
- AWS EC2 (Elastic Compute Cloud) : For More Details [Click Here](#)

- AWS DynamoDB : For More Details [Click Here](#)
- Amazon SNS : For More Details [Click Here](#)

## **Project WorkFlow**

### **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**

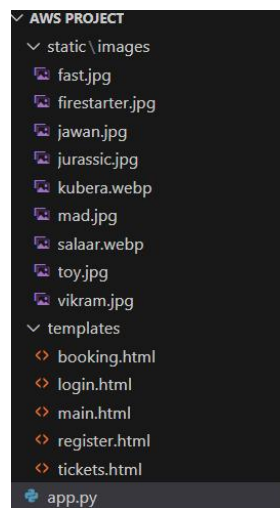
- Upload Flask Files
- Run the Flask App

### **Milestone 8. Testing and Deployment**

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

## MileStone 1: Backend Development and Application SetUp

- Firstly develop the backend structure using Flask



In this create a project folder named “AWS Project”. In that create a folder named templates in which we can store all the frontend html pages like login.html, register.html, booking.html, and other pages. And then create a folder named static->images which is used to store all the .jpg and other media files which you want to use in the project.

## Description of Code

### Flask App Initialization

```
from flask import Flask, request, render_template, redirect, url_for, session, flash
import boto3
import uuid
import os
```

Firstly, import all the essential packages which are required to run the flask application, boto3 for dynamodb operations, uuid for generating the unique booking id's and os for accessing the environment variables like the secret keys etc

Initialize the flask application using the Flask(\_\_name\_\_)

```
# Flask setup
app = Flask(__name__)
app.secret_key = "b7127b5f7f99f682c84375c6e5cf4fa80734d4d790372f4ae3926a6b6f8c3c6f"
```

After that setup the dynamodb configuration

```
# AWS setup
REGION = 'us-east-1'
dynamodb = boto3.resource('dynamodb', region_name=REGION)
users_table = dynamodb.Table('userdata')
bookings_table = dynamodb.Table('Bookingdata')
```

AWS DynamoDB integration using the boto3 library. It specifies the AWS region as us-east-1. The dynamodb object is created to interact with DynamoDB services. Two tables are accessed: userdata (for storing user details) and Bookingdata (for storing booking information). These tables will be used throughout the application to perform database operations.

## SNS Configuration

```
sns = boto3.client('sns', region_name=REGION)
sns_topic_arn = 'arn:aws:sns:us-east-1:195275652542:BookingRequestNotifications'

# Send email via AWS SNS
def send_booking_email(email, movie, date, time, seat, booking_id):
    message = f"""
    📌 Booking Confirmed!

    Movie: {movie}
    Date: {date}
    Time: {time}
    Seat(s): {seat}
    Booking ID: {booking_id}

    Thank you for booking with us!
    """
    try:
        sns.publish(
            TopicArn=sns_topic_arn,
            Message=message,
            Subject="Your Movie Ticket Booking Confirmation"
        )
        return True
    except Exception as e:
        print(f"Error sending email via SNS: {e}")
        return False
```

This function uses AWS SNS to send a movie ticket booking confirmation email. It takes in details like email, movie, date, time, seat, and booking ID, and formats them into a user-friendly message. The message is then published to an SNS topic (`BookingRequestNotifications`) using the `boto3` client. The email subject is set as "Your Movie Ticket Booking Confirmation". If the message is successfully published, the function returns `True`. If there's an error, it catches the exception, prints the error message, and returns `False`. The email parameter assumes the recipient is already subscribed to the SNS topic.

## Web Page Routing

### ◆ Home Route

```
# Routes
@app.route('/')
def index():
    return render_template('login.html')
```

**Desc :** This Flask route defines the root URL `'/'` of the web application. When a user accesses the homepage, the `index()` function is called. It renders and returns the `login.html` template. This means users will be directed to the login page by default. The `@app.route` decorator binds the URL path to the function.

### ◆ Register Route

```

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        fname = request.form.get('fname')
        lname = request.form.get('lname')
        email = request.form.get('email')
        password = request.form.get('setPwd')
        confirm_password = request.form.get('confirmPwd')
        if not all([fname, lname, email, password, confirm_password]):
            return "All fields are required.", 400

        if password != confirm_password:
            return "Passwords do not match.", 400
        try:
            existing = users_table.get_item(Key={"email": email}).get("Item")
            if existing:
                return "User already registered.", 400
        except Exception as e:
            print("Error checking user:", e)
            return "Database error", 500
        try:
            users_table.put_item(Item={
                "first_name": fname,
                "last_name": lname,
                "email": email,
                "password": password
            })
            return redirect(url_for('index'))
        except Exception as e:
            print("Error inserting user:", e)
            return "Registration failed", 500

    return render_template('register.html')

```

**Desc:** This Flask route handles user registration via GET and POST methods. On POST, it collects form data: first name, last name, email, password, and confirm password. It checks that all fields are filled, and that the passwords match. Then it queries the DynamoDB `users_table` to check if the email already exists. If not, it inserts the new user into the table and redirects to the login page. If any errors occur during the process, appropriate error messages and status codes are returned. On GET, it simply renders the `register.html` template.

## ◆ Login Route

```

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

        try:
            user = users_table.get_item(Key={"email": email}).get("Item")
            if user and user['password'] == password:
                session['user'] = email
                return redirect(url_for('main'))
        except Exception as e:
            print("Login error:", e)

        flash("Invalid credentials")
        return redirect(url_for('login'))

    return render_template('login.html')

```

**Desc :** This Flask route handles user login using GET and POST methods. On a POST request, it retrieves the email and password from the login form. It then checks the DynamoDB `users_table` for a matching user record. If the user exists and the password matches, the email is stored in the session and the user is redirected to the main page. If the login fails or an error occurs, it prints the error and redirects back to the login page with a flash message. On GET requests, it simply renders the `login.html` template.

## ◆ Main Block and Booking block

```
@app.route('/main')
def main():
    if 'user' not in session:
        return redirect(url_for('login'))
    return render_template('main.html')
@app.route('/booking')
def booking_page():
    if 'user' not in session:
        return redirect(url_for('login'))
    movie = request.args.get('movie')
    booked = bookings_table.scan(
        FilterExpression=boto3.dynamodb.conditions.Attr("Movie").eq(movie)
    ).get("Items", [])
    booked_seats = []
    for b in booked:
        if "Seat" in b:
            seats = b["Seat"].split(", ")
            booked_seats.extend(seats)
    return render_template('booking.html', movie=movie, booked_seats=booked_seats)
```

**Desc :** The `/main` route renders the main page if the user is logged in; otherwise, it redirects to the login page. The `/booking` route shows the booking page for a selected movie. It fetches booked seats from DynamoDB for that movie and passes them to the `booking.html` template. Unauthenticated users are redirected to login.

## ◆ Booking Confirmation Block

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

    data = {
        'email': session['user'], # FIXED
        'Movie': request.form['movie'],
        'Date': request.form['date'],
        'Time': request.form['time'],
        'Seat': request.form['seat'],
        'BookingID': str(uuid.uuid4())
    }

    try:
        bookings_table.put_item(Item=data)

        # Send SNS email notification
        send_booking_email(
            data['email'], data['Movie'], data['Date'], data['Time'], data['Seat'], data['BookingID']
        )

        return render_template('tickets.html', booking=data)
    except Exception as e:
        print("Booking error:", e)
        flash("Booking failed. Please try again.")
        return redirect(url_for('main'))
```



**Desc :** This route handles movie ticket booking via POST. It checks user login, collects form data, and generates a unique booking ID. The data is stored in the DynamoDB `bookings_table`. Then, a booking confirmation email is sent using AWS SNS. If successful, it renders `tickets.html`; otherwise, it flashes an error.

### ◆ Deployment Block

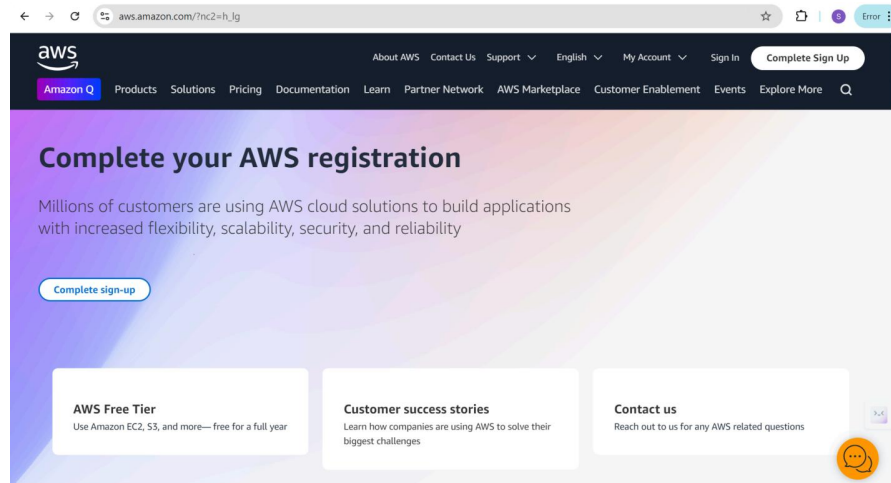
```
# Run the app
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

**Desc :** The application is configured to run on host '0.0.0.0', which allows devices across the local network to access it, making it handy for collaborative development or testing on multiple machines. It runs on port 5000, a common default for Flask apps, and the debug mode is turned on, offering developers enhanced error messages and auto-reloading during changes.

## Milestone 2: AWS Account Setup and Login

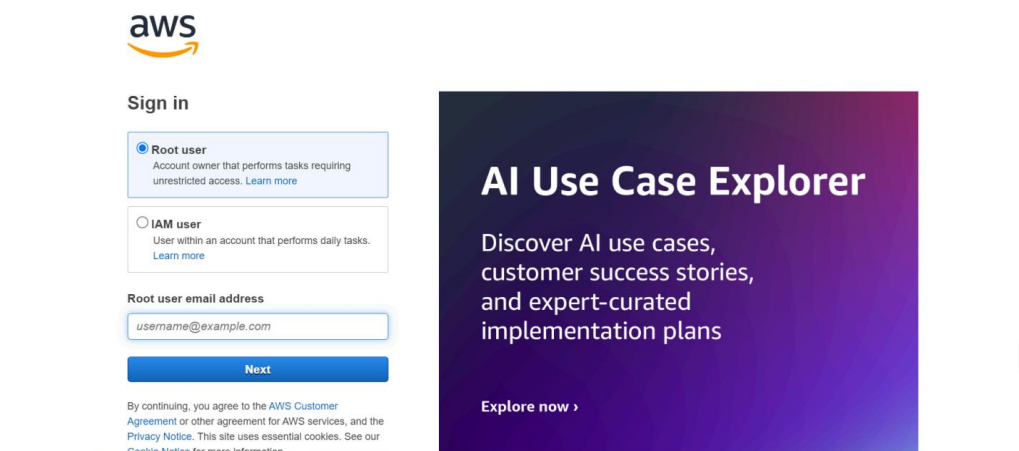
**Activity 2.1:** Set up an AWS account if not already done.

- Sign up for an AWS account and configure billing settings



## Activity 2.2: Log in to the AWS Management Console

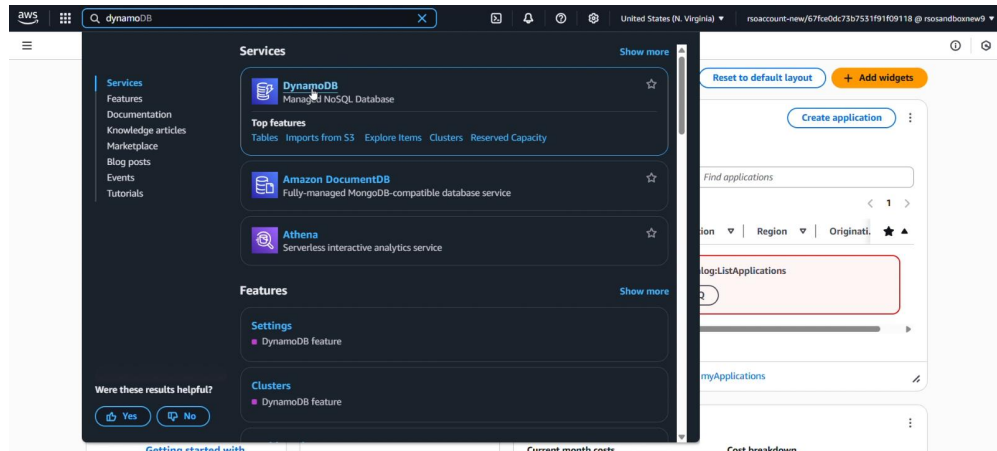
- After setting up your account, log in to the [AWS Management Console](#)



## Milestone 3. DynamoDB Database Creation and Setup

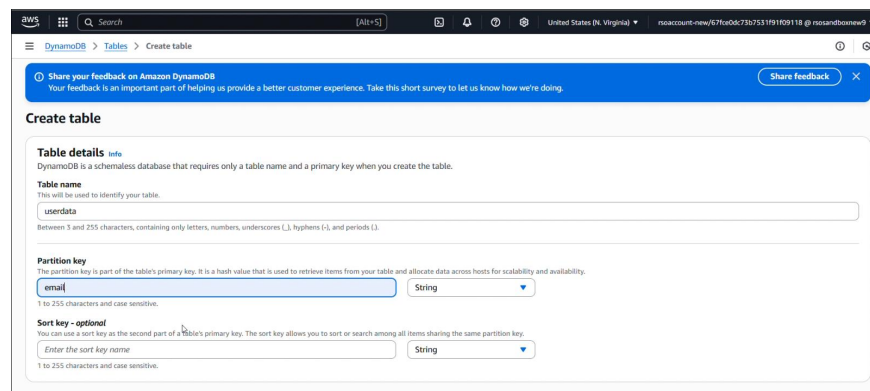
### Activity 3.1 : Creating a Dynamodb table

For creating the table open the amazon console and search for dynamodb

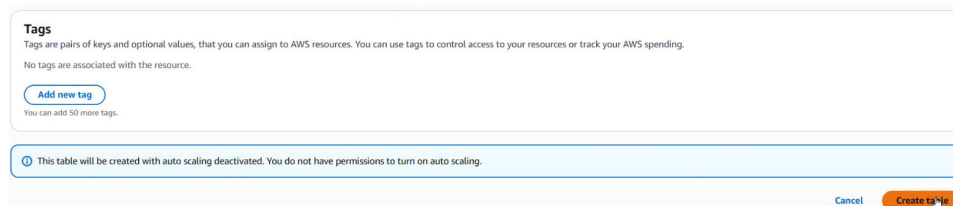


## Activity 3.2 : Creating the tables (userdata and bookingdata) for storing information

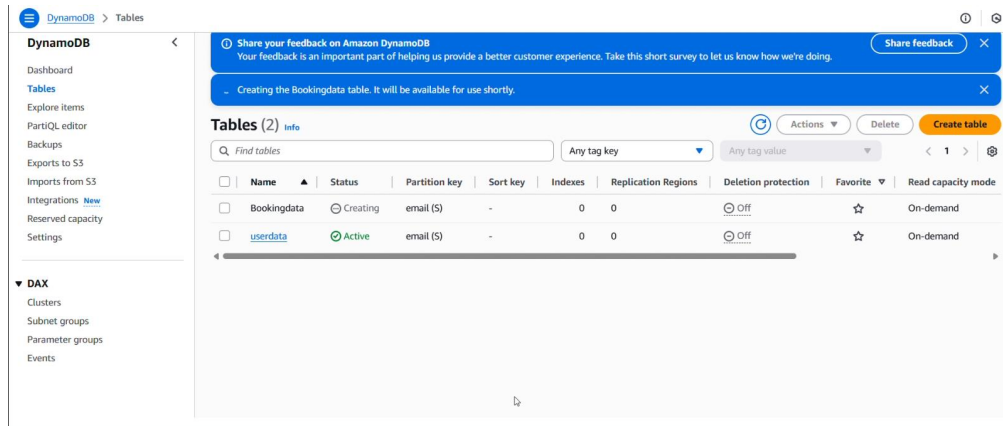
After this, select the dynamodb and click on create table option. In that give the table as the same name in which we used in the flask code for storing the userdata and their credentials and give the partition key as email.



Now click on create table option after scrolling down, then a new table will appear in the dynamodb page



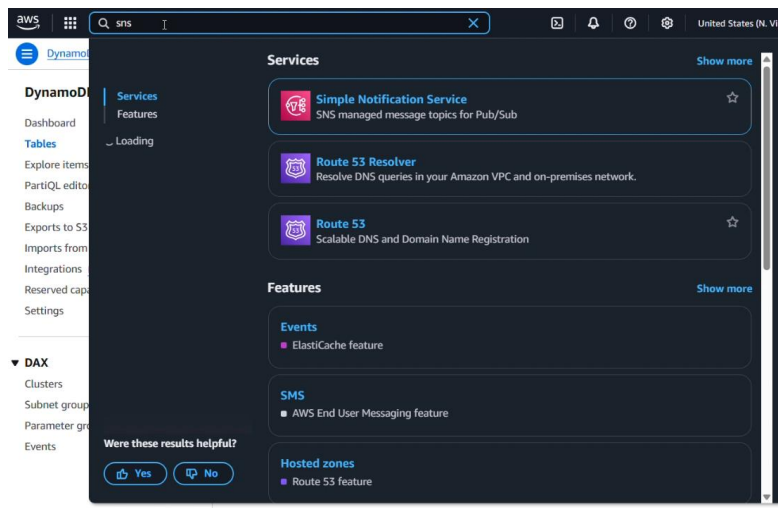
And repeat the same procedure for remaining tables present in the flask code i.e the bookingtable which is used to store the booking data



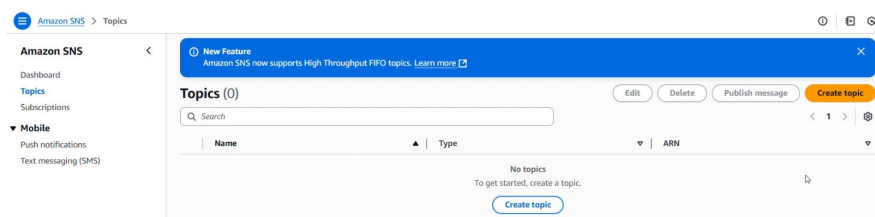
## Milestone 4 : SNS Modification Setup

### Activity 4.1 : Create SNS topic for book request notifications

Firstly, In AWS search for SNS



Select that one and in that click on create new topic



Open standard type for general notifications usecases and click on 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
- Subscription protocols: SQS

☒ Standard

- Best-effort message ordering
- At-least-once message delivery
- Subscription protocols: SQS, Lambda, Data Firehose, HTTP, SMS, email, mobile application endpoints

**Name**

BookingRequestNotifications

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.

Now a topic will be created with the name of BookingRequestNotification

**Amazon SNS**

Dashboard  
Topics  
Subscriptions

**Mobile**  
Push notifications  
Text messaging (SMS)

**BookingRequestNotifications**

**Details**

Name	BookingRequestNotifications
Display name	-
ARN	arn:aws:sns:us-east-1:195275652542:BookingRequestNotifications
Type	Standard

**Subscriptions** Access policy Delivery policy (HTTP/S) Delivery status logging Encryption Tags Integrations

Subscriptions (0)

Edit Delete Request confirmation Confirm subscription Create subscription

## Activity 4.2 : Subscribe the users and library staff to SNS notifications

After creating a topic click on create subscription. In that we need to enter the type of protocol i.e email and the your emailid for subscription confirmation

**Create subscription**

**Details**

Topic ARN

arn:aws:sns:us-east-1:195275652542:BookingRequestNotifications

**Protocol**

The type of endpoint to subscribe

Email

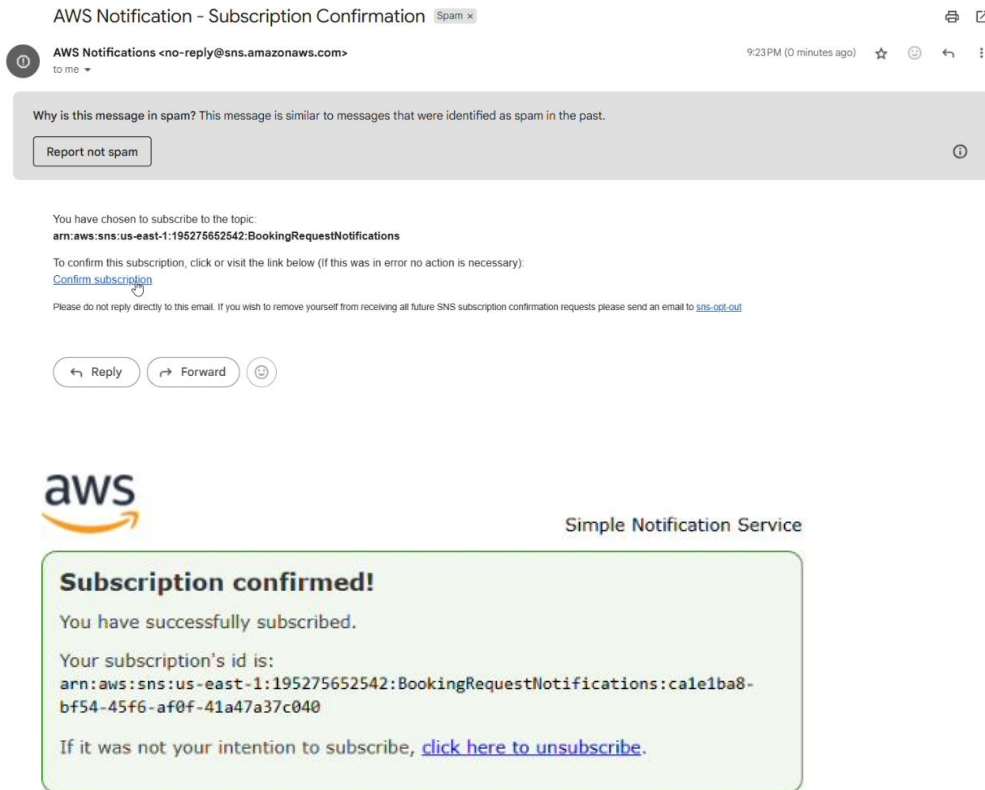
**Endpoint**

An email address that can receive notifications from Amazon SNS.

nikhilnandanavanam.123@gmail.com

After your subscription is created, you must confirm it. info

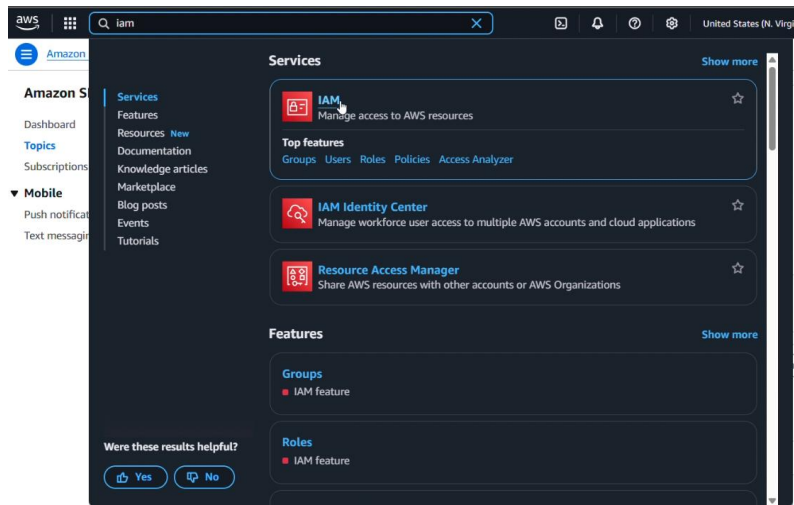
Check the mail which you entered in the subscription information. In that mail you'll see an subscription request mail like this and in that to click on "Confirm Subscription"



## Milestone 5 : IAM Role

### Activity 5.1 :Create an IAM role

Search for IAM in AWS



In that click on create role. After clicking that one you'll see a page to select the trusted entity. In trusted entity type select "AWS Service" and use case as "EC2"

**Select trusted entity**

**Trusted entity type**

- ☒ **AWS service**  
Allow AWS services like EC2, Lambda, or others to perform actions in this account.
- ☐ **AWS account**  
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.
- ☐ **Web identity**  
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.
- ☐ **SAML 2.0 federation**  
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.
- ☐ **Custom trust policy**  
Create a custom trust policy to enable others to perform actions in this account.

**Use case**  
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

**Service or use case**  
EC2

## Activity 5.2 : Attach Policies

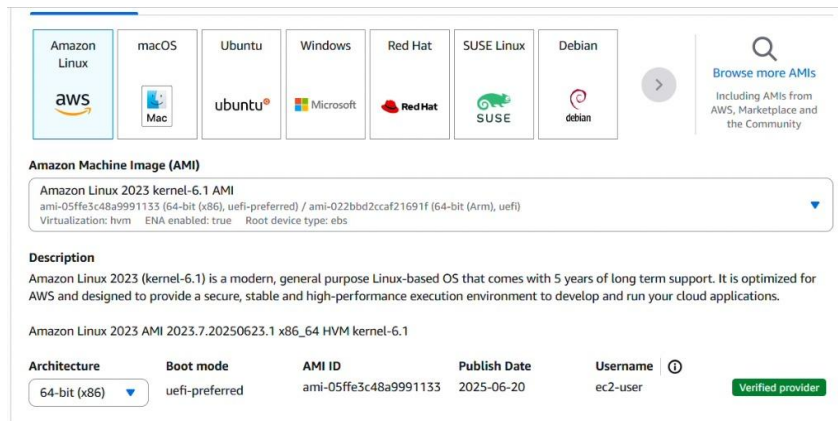
After that you need to add the permissions for EC2, SNS and dynamodb  
 AmazonEC2FullAccess, AmazonDynamoDBFullAccess and  
 AmazonSNSFullAccess

Permissions policy summary		
Policy name	Type	Attached as
<a href="#">AmazonDynamoDBFullAccess</a>	AWS managed	Permissions policy
<a href="#">AmazonEC2FullAccess</a>	AWS managed	Permissions policy
<a href="#">AmazonSNSFullAccess</a>	AWS managed	Permissions policy

Now give the IAM role and click on create role







Create a new key pair and it will download the .pem file

A new file named project.pem file will be downloaded



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

## Edit inbound rules Info

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

Security group rule ID	Type <small>Info</small>	Protocol <small>Info</small>	Port range <small>Info</small>	Source <small>Info</small>	Description - optional <small>Info</small>	
sgr-0a4e7306bebf63041	SSH	TCP	22	Custom	0.0.0.0/0	Delete
-	HTTP	TCP	80	Anywh...	0.0.0.0/0	Delete
-	Custom TCP	TCP	5000	Anywh...	0.0.0.0/0	Delete

[Add rule](#)

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

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

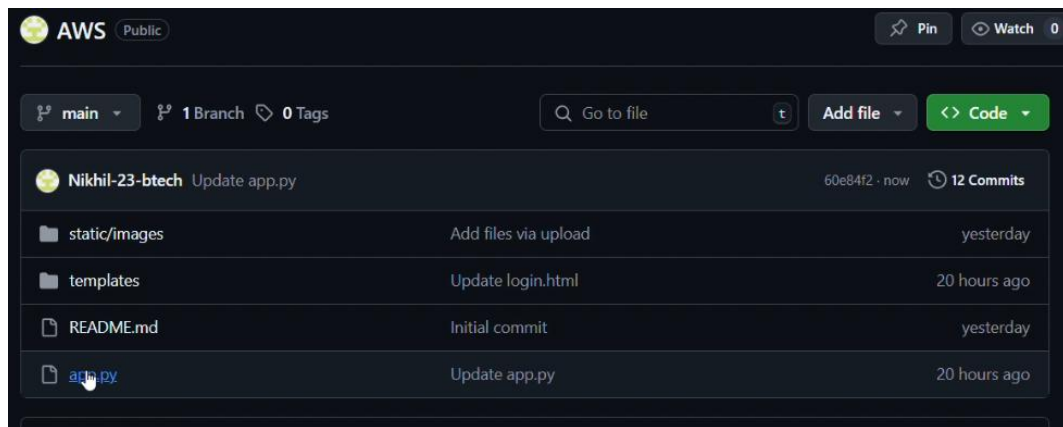
To connect to EC2 using **EC2 Instance Connect**, start by ensuring that an **IAM role** is attached to your EC2 instance. You can do this by selecting your instance, clicking on **Actions**, then navigating to **Security** and selecting **Modify IAM Role** to attach the appropriate role. After the IAM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

The screenshot shows the AWS Management Console interface for an EC2 instance. The left sidebar contains navigation links for EC2, including Dashboard, EC2 Global View, Events, Instances, Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Capacity Reservations, Images, AMIs, AMI Catalog, Elastic Block Store, Volumes, and Snapshots. The main content area displays the 'Instance summary for i-0e293a20ab7529c3a (project)'. Key details include: Instance ID (i-0e293a20ab7529c3a), IPv6 address, Hostname type (IP name: ip-172-31-20-78.ec2.internal), Answer private resource DNS name (IPv4 (A)), Auto-assigned IP address, IAM Role, IMDSv2 (Required), and Operator. The instance state is 'Stopped'. The 'Actions' menu is open, showing options like 'Connect', 'Instance diagnostics', 'Instance settings', 'Networking', 'Security', 'Image and templates', and 'Monitor and troubleshoot'. The 'Connect' button is highlighted, and a tooltip shows additional options: 'Change security groups', 'Get Windows password', and 'Modify IAM role'. The 'Connect' button is also visible in the top right corner of the instance summary card.

## Modify the IAM role

The screenshot shows the 'Modify IAM role' page in the AWS Management Console. The page title is 'Modify IAM role' with an 'Info' link. Below the title, it says 'Attach an IAM role to your instance.' The 'Instance ID' is 'i-0e293a20ab7529c3a (project)'. The 'IAM role' section has a dropdown menu with 'Choose IAM role' and a search bar. Below the search bar, there are two options: 'No IAM Role' (with a note 'Choose this option to detach an IAM role') and 'sms\_dynamodb\_role' (with a note 'arn:aws:iam::199675652542:instance-profile/sms\_dynamodb\_role'). To the right of the dropdown is a 'Create new IAM role' button. At the bottom right, there are 'Cancel' and 'Update IAM role' buttons.

After modifying the IAM role start the instance and click on connect. Before this make sure that you have uploaded all the project related info to the github by creating a repository



Then if you click on connect option present in the EC2 page then in that it consists of many contents. In that click on SSH client

The screenshot shows the 'Connect to instance' page in the AWS Management Console. The breadcrumb navigation is 'EC2 > Instances > i-0e293a20ab7529c3a > Connect to instance'. The page title is 'Connect' with an 'Info' link. Below the title, it says 'Connect to an instance using the browser-based client.' There are four tabs: 'EC2 Instance Connect', 'Session Manager', 'SSH client' (which is selected), and 'EC2 serial console'. The 'SSH client' tab shows a list of steps: 1. Open an SSH client. 2. Locate your private key file. The key used to launch this instance is project.pem. 3. Run this command, if necessary, to ensure your key is not publicly viewable. 4. Connect to your instance using its Public DNS: ec2-18-212-226-87.compute-1.amazonaws.com. Below the steps, there is an 'Example:' section with the command: `ssh -i "project.pem" ec2-user@ec2-18-212-226-87.compute-1.amazonaws.com`

Now in your system open the Windows PromptShell and run it as administrator. Copy the example link which provided in the above image and paste that one in the prompt shell. Modify the project.pem file to the location in which that file located in your system. After running this command it shows like this

```
PS C:\WINDOWS\system32> ssh -i "C:\Users\nikhi\project.pem" ec2-user@ec2-18-212-226-87.compute-1.amazonaws.com
The authenticity of host 'ec2-18-212-226-87.compute-1.amazonaws.com (18.212.226.87)' can't be established.
ED25519 key fingerprint is SHA256:mszAQuV9m4leRg7KuyZ2P5sGphKhRqrGrUIDWA3pas.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-18-212-226-87.compute-1.amazonaws.com' (ED25519) to the list of known hosts.

#
~\_ ##### Amazon Linux 2023
~~~\_ #####\
~~~\###|
~~~\#/_
~~~V~' '->
~~~~
~~~~_._/
~~~~/_/m/'
[ec2-user@ip-172-31-20-78 ~]$
```

## MileStone 7 : Deployment on EC2

## Activity 7.1 : Upload the Flask App

Before proceeding to uploading the flask app run the following commands in the prompt shell

```
sudo yum update -y
```

```
sudo yum install python3 git
```

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

```
sudo pip3 install flask boto3
```

The below commands used to verify installations

flask --version

```
git --version
```

Now upload the flask application by using the command

```
git clone your-git-repo-link
```

In place of your-git-repo-link modify or replace that one with the github link of your flask application which you recently uploaded all the project related files

## Activity 7.2 Run the flask app

After cloning the git repository, navigate to that folder in which the app.py code exists by using the cd command

After navigating to the directory run the below command for executing the app.py backend flask code

```
sudo python3 app.py
```

```
Cloning into 'AWS'...
remote: Enumerating objects: 51, done.
remote: Counting objects: 100% (51/51), done.
remote: Compressing objects: 100% (48/48), done.
remote: Total 51 (delta 13), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (51/51), 629.29 KiB | 25.17 MiB/s, done.
Resolving deltas: 100% (13/13), done.
[ec2-user@ip-172-31-20-78 ~]$ cd AWS
[ec2-user@ip-172-31-20-78 AWS]$ ls
README.md  app.py  static  templates
[ec2-user@ip-172-31-20-78 AWS]$ sudo 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://172.31.20.78:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 902-036-453
```

Now copy the public ipv4 address of the EC2 instance. After copying the ipv4 address open a new tab in the browser. In that type as the below format

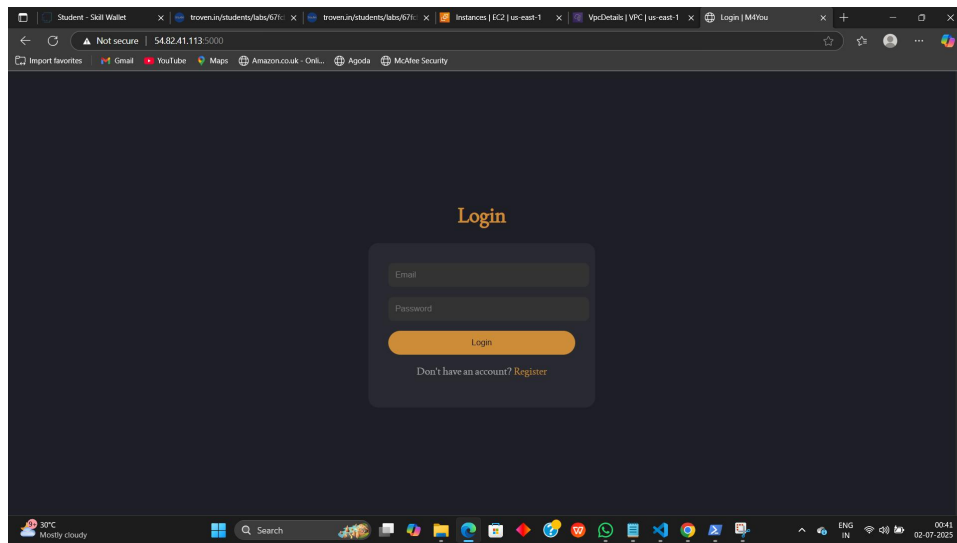
[http://public-ipv4-address:port\\_number](http://public-ipv4-address:port_number)

Modify the public ipv4 address to the ipv4 address of the EC2 instance and port number to the port which you used in the flask code (ex: 5000)

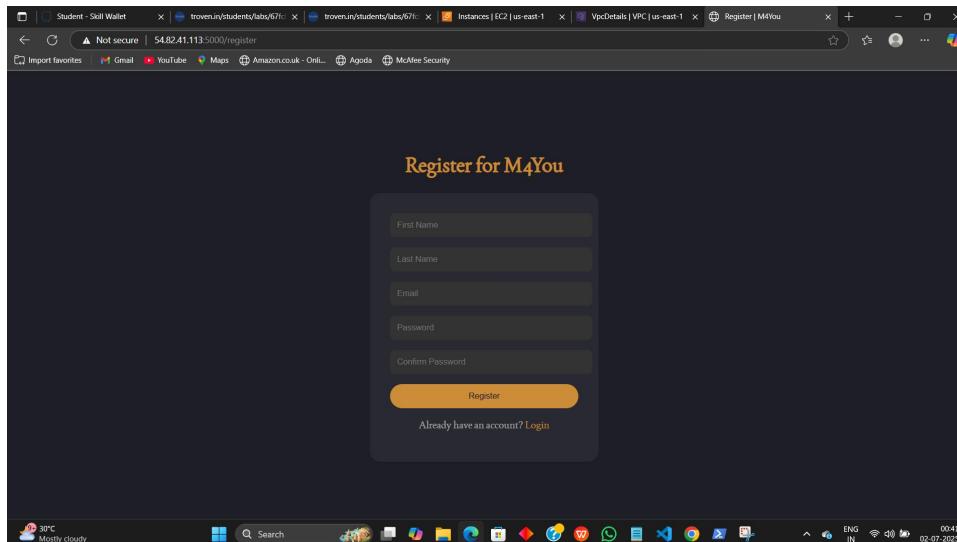
## MileStone - 8 : Testing and Deployment

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

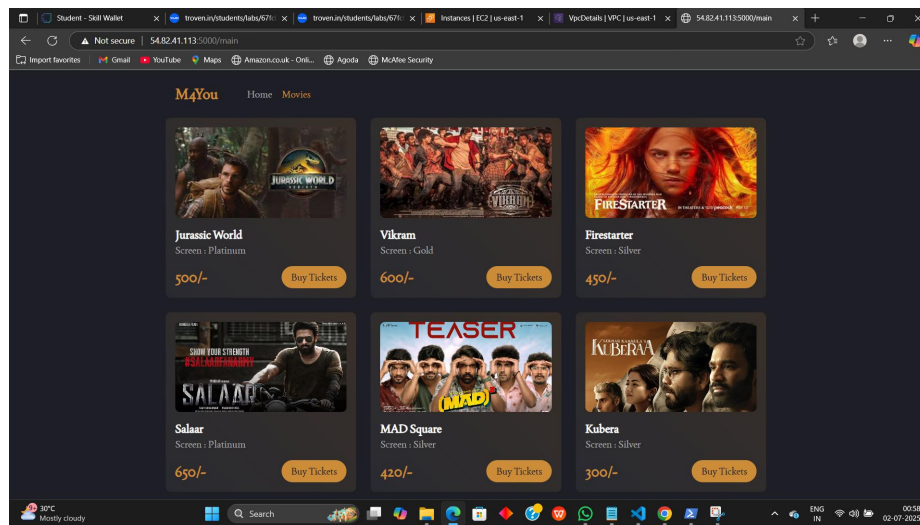
### Login Page



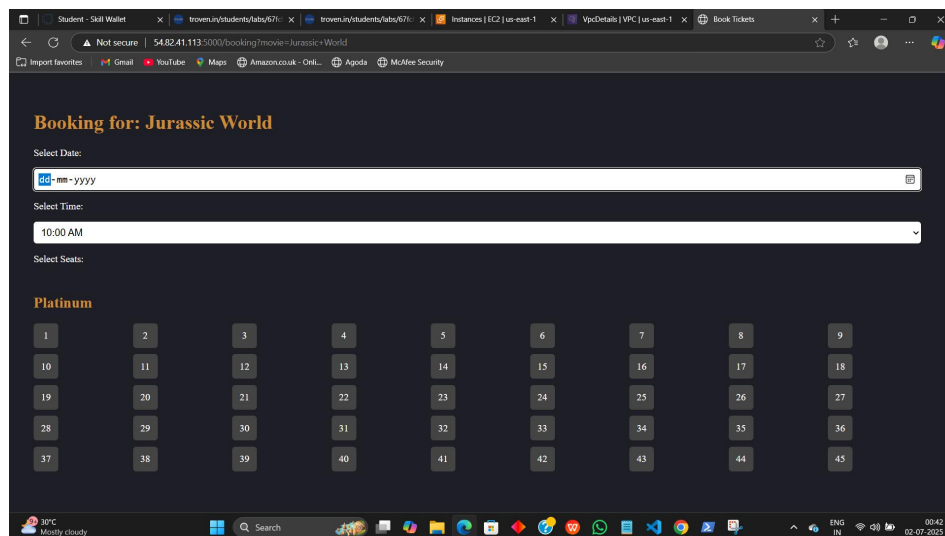
### Register Page



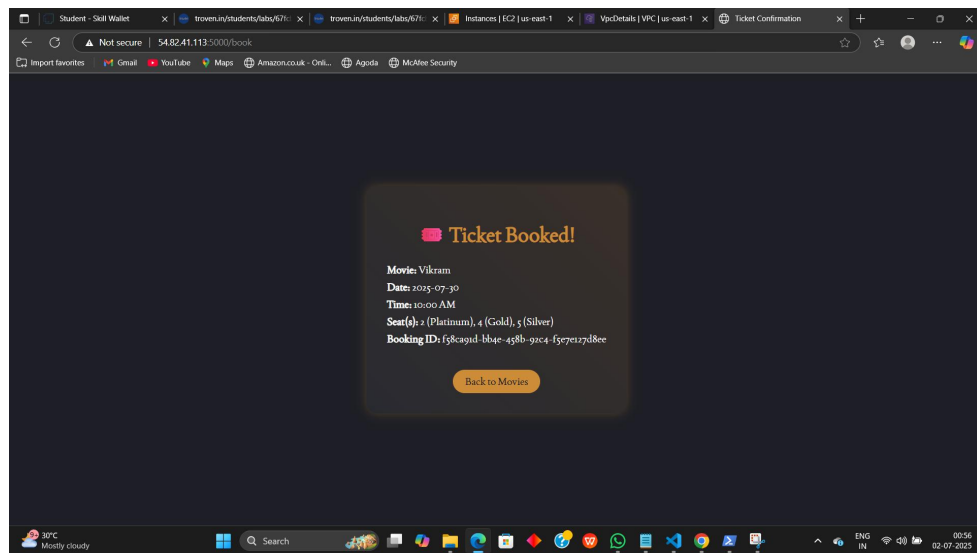
## Main Page



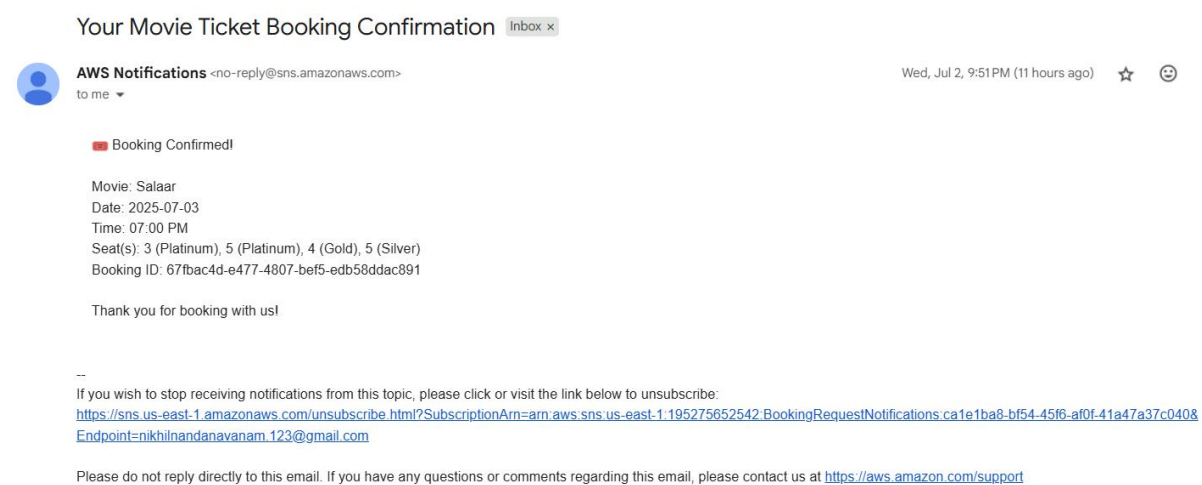
## Ticket booking page



## Ticket Confirmation Page



## Booking confirmation through email





## Conclusion

The provided project is a web-based **Smart Movie Ticket Booking System** built using the Flask framework and integrated with AWS services. It allows users to register, log in, and book movie tickets through a simple and responsive user interface. The application stores user data and booking details in **AWS DynamoDB**, with two main tables: `userdata` for user credentials and profile information, and `Bookingdata` for storing movie booking details like movie name, date, time, and seat number. Unique booking IDs are generated using the `uuid` module.

Upon successful booking, the application uses **AWS SNS (Simple Notification Service)** to send a confirmation email to the user, enhancing the reliability and user experience. The system uses session handling to maintain login status and ensures that only registered users can make bookings. It includes multiple HTML templates (like login, register, main, booking, and ticket views) styled consistently for a seamless user experience.

The app also includes a static directory with movie poster images to enrich the interface visually. It demonstrates the effective use of cloud-based, serverless architecture by leveraging AWS for both data persistence and communication services. This project serves as a great example of integrating cloud tools with traditional web frameworks, making it suitable for small-scale theater management systems or educational use cases that teach cloud application deployment.