

Movie Magic - Smart Movie Ticket Booking System

Project Documentation

1. INTRODUCTION:

PROJECT TITLE: Movie Magic—Smart Movie Ticket Booking System

In today's digital age, the demand for a seamless and modern movie-watching experience is rapidly growing. Traditional ticket booking methods often struggle to meet this demand due to challenges like long queues, limited availability, and inconsistent service. To overcome these issues, the development team created **Movie Magic**—a smart, cloud-based movie ticket booking system.

Movie Magic leverages modern technologies to deliver a smooth and user-friendly booking experience. Built with Flask for backend development, hosted on AWS EC2, and integrated with DynamoDB for efficient data management, the platform enables users to register, log in, and book movie tickets online with ease. The system allows users to search for movies and events by location, view real-time seat availability, and confirm bookings in just a few clicks. Additionally, AWS SNS integration provides instant email notifications upon booking, enhancing user engagement and building trust.

This cloud-native solution streamlines the entire movie ticketing process, offering a fast, scalable, and convenient alternative to traditional methods

2. PROJECT OVERVIEW:

Movie Magic is a cloud-based movie ticket booking system designed to deliver a fast, convenient, and modern user experience. The project aims to eliminate the common limitations of traditional ticket booking—such as long queues, manual operations, and inconsistent availability—by offering a fully digital platform.

The system is built using **Flask** for backend development, ensuring lightweight and efficient server-side operations. It is hosted on **AWS EC2**, which provides scalable and reliable cloud infrastructure, while **Amazon DynamoDB** is used for dynamic and highly available data storage, supporting operations like movie listings, user accounts, and seat management.

Key features of the platform include:

- ❖ **User Registration and Login:** Secure user authentication and personalized access.
- ❖ **Movie/Event Search by Location:** Users can browse and filter showtimes based on their preferred city or theater.
- ❖ **Real-Time Seat Availability:** Dynamically updates seat maps to reflect current availability.
- ❖ **Online Ticket Booking:** Streamlined booking process completed within a few steps.
- ❖ **Instant Booking Confirmation:** Integration with **AWS SNS** allows automated email notifications with booking details, enhancing reliability and user satisfaction.

- ❖ Overall, Movie Magic demonstrates a scalable, efficient, and user-centric approach to digital movie ticketing, leveraging cloud technologies to provide a modern alternative to traditional systems.

❖ PURPOSE:

The primary purpose of the **Movie Magic** system is to provide users with a fast, reliable, and modern movie ticket booking experience by utilizing a cloud-native architecture. Designed to overcome the inefficiencies of traditional ticketing methods, Movie Magic ensures users can easily search, book, and confirm tickets online, all in real-time.

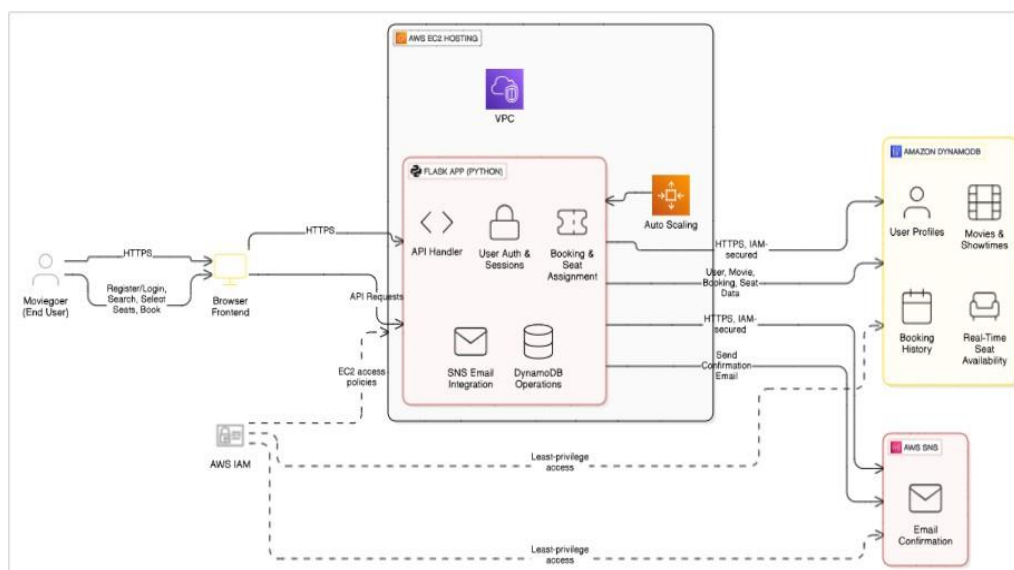
Through the integration of **AWS EC2**, the platform guarantees high availability and scalability, ensuring smooth operation even during peak traffic. The **Flask** backend facilitates efficient data handling and seamless navigation, allowing users to select movies, view showtimes, choose seats interactively, and finalize bookings without delay.

The system also enhances post-booking engagement by using **AWS SNS** to deliver **instant email notifications**, ensuring users receive timely confirmation with detailed ticket information. Furthermore, with **DynamoDB** managing real-time data storage and retrieval, both users and administrators can reliably track bookings and event details.

Ultimately, Movie Magic aims to revolutionize the movie and event booking process by delivering a **user-friendly, responsive, and trustworthy platform** that meets the expectations of today's digitally connected audience.

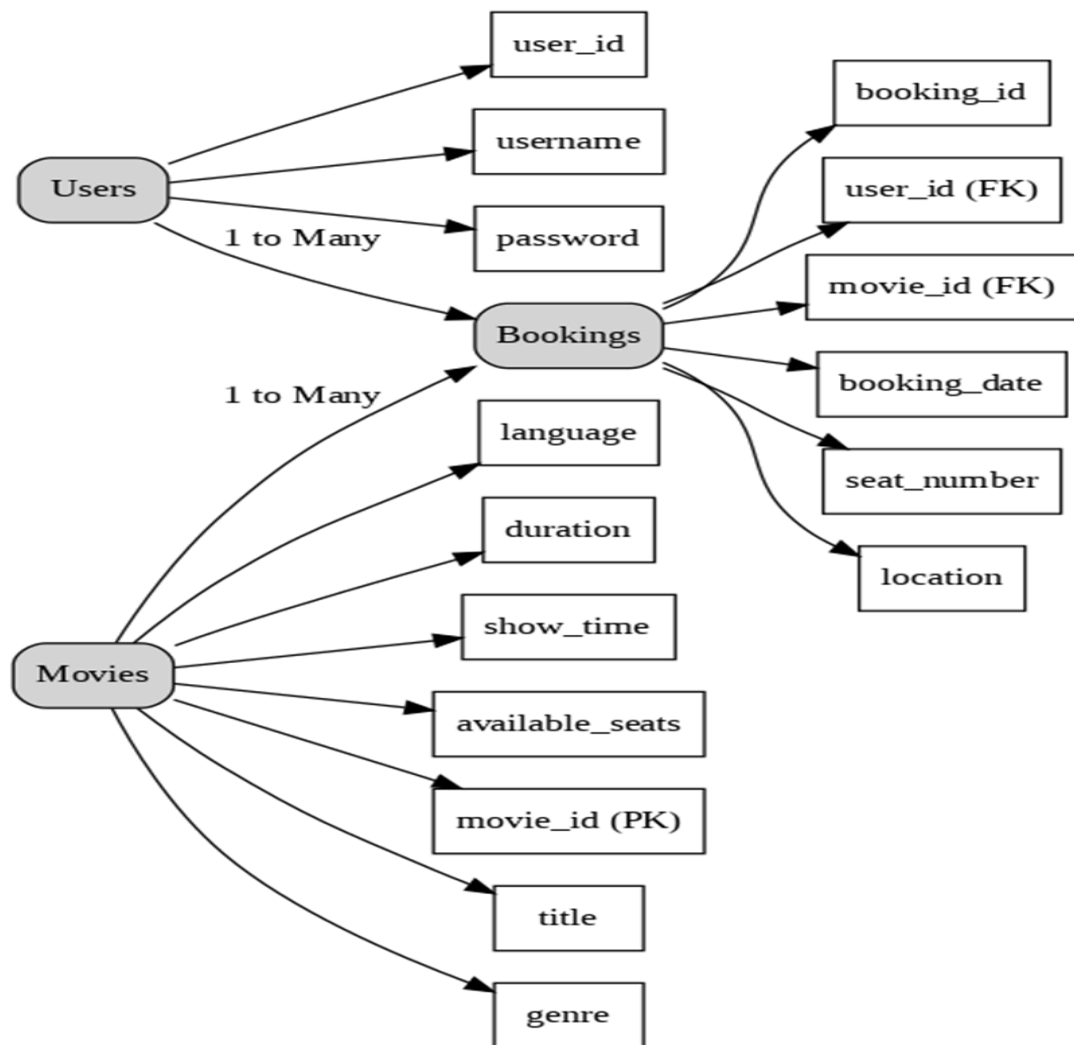
3. ARCHITECTURE:

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.



❖ ER DIAGRAM:

An ER (Entity-Relationship) diagram visually represents the logical structure of a database by defining entities, their attributes, and the relationships between them. It helps organize data efficiently by illustrating how different components of the system interact and relate. This structured approach supports effective database normalization, data integrity, and simplified query design.

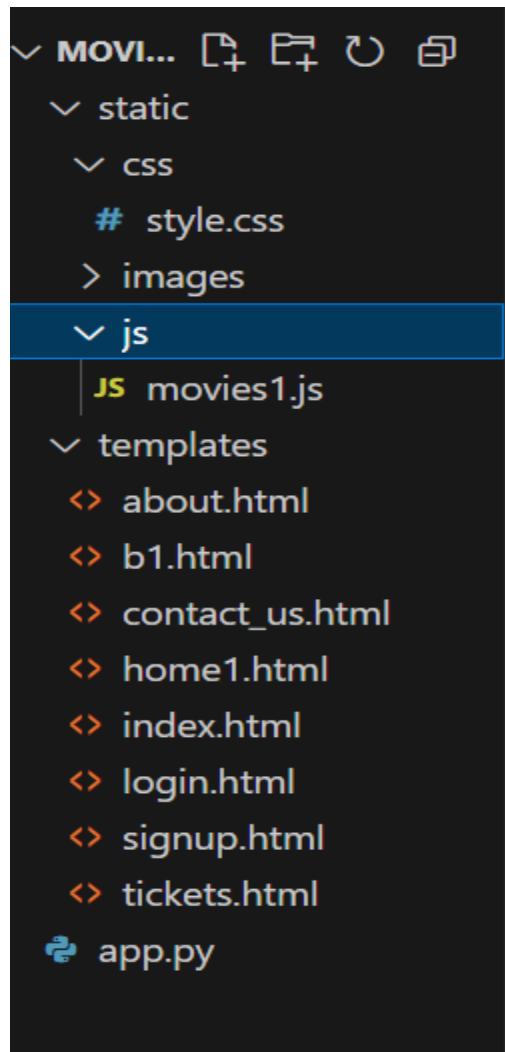


❖ PRE-REQUISTIC:

- 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>

4. FOLDER STRUCTURE:



5. RUNNING THE APPLICATION:

❖ Backend Development

1. Project Setup :

Start by creating a new project directory for your application. Set up a virtual environment and initialize the Python project. Install the necessary libraries including Flask (for the backend), Boto3 (for AWS services), and any others needed for handling requests, security, and session.

2. Database Configuration :

Use AWS DynamoDB as your NoSQL database. Create two tables:

Users: Stores user data such as email, hashed password, and session details.

Bookings: Stores each movie ticket booking including movie name, showtime, number of seats, seat numbers, price, and booking ID.

3. Flask Application :

Set up a Flask server to handle HTTP requests from the frontend. Configure session handling for logged-in users. Ensure the server can respond to routes for login, signup, viewing movies, booking tickets, and downloading confirmation.

4. API Routes and Logic :

Define clear backend routes to handle all actions such as:

- User registration and login
- Listing available movies
- Selecting and booking seats
- Sending confirmation emails using AWS SNS
- Showing ticket summaries and generating downloadable ticket PDFs

5. Data Models :

Define the structure of data to be stored in DynamoDB. Each user has a unique email (primary key). Each booking has a unique booking ID (primary key). Design models in a way that links users to their bookings.

6. User Authentication

Implement secure user login and signup systems. Use password hashing to protect user credentials. Manage sessions so that users remain logged in across pages and only logged-in users can book tickets.

7. Booking System :

Allow users to:

- View a list of available movies and showtimes
- Select preferred date, time, and seats
- Confirm bookings and view a success screen
- Store all booking details in the database with a unique booking ID

8. Email Notifications

Integrate AWS SNS (Simple Notification Service) to send booking confirmation emails to users. Automatically trigger an email after a successful booking is made.

❖ Frontend Development

1. Project Setup :

Use HTML, CSS, Bootstrap, and Jinja2 templates to build the user interface. Organize files into *templates* for dynamic HTML pages and *static* for styling and assets.

2. User Experience Design :

Design a clean, responsive, and user-friendly UI. Pages include:

- Login and signup forms
- A movie list page with options to select showtime
- Seat selection and booking confirmation page
- Ticket summary and download page

3. Dynamic Content Rendering :

Use Jinja2 templating (integrated with Flask) to dynamically load content based on backend data. For example, movie listings, available seats, user names, and booking details are rendered based on server data.

4. Interactive Booking Flow :

Users can:

- Select a movie
- Choose the date and showtime
- Select the number of seats
- View pricing and proceed to book
- After booking, users receive confirmation and can download the ticket

6. API DOCUMENTATION:

Flask App Initialization :

Import 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:

```
11 # Use a static secret key
12 app.secret_key = 'your_static_secret_key_here' # Replace with your own secret string
13
14 # AWS Configuration - read from environment variables for security
15 AWS_REGION = os.environ.get('AWS_REGION', 'ap-south-1')
16
17 # Fix the SNS_TOPIC_ARN assignment - this was the main issue
18 # Instead of using os.environ.get with the ARN as the key, set it directly
19 SNS_TOPIC_ARN = 'arn:aws:sns:ap-south-1:605134430972:MovieTicketNotifications'
20
21 # Initialize AWS services with proper credentials handling
22 # On EC2, this will use the instance profile/role automatically
23 dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)
24 sns = boto3.client('sns', region_name=AWS_REGION)
25
26 # DynamoDB tables
27 USERS_TABLE_NAME = os.environ.get('USERS_TABLE_NAME', 'MovieMagic_Users')
28 BOOKINGS_TABLE_NAME = os.environ.get('BOOKINGS_TABLE_NAME', 'MovieMagic_Bookings')
29
30 users_table = dynamodb.Table(USERS_TABLE_NAME)
31 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.

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.
        """
        # Publish message to SNS topic
        sns.publish(TopicArn=SNS_TOPIC_ARN, Subject=email_subject, Message=email_message)
```



```

    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:

This route handles the user registration process. It collects user input data (such as name, email, and password) from the registration form. The password is securely hashed using a hashing algorithm (e.g., `werkzeug.security.generate_password_hash`) before storing it in the database. The user's information is then saved in the **Users** table in DynamoDB, using the email as the primary key.

```

@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'))

```



```

2         return redirect(url_for('login'))
3
4     except ClientError as e:
5         print(f"Error accessing DynamoDB: {e.response['Error']['Message']}")
6         flash('An error occurred during registration. Please try again.', 'danger')
7
8     return render_template('signup.html')
9

```

Login Route (GET/POST):

This route is responsible for handling user login functionality. It accepts both `GET` and `POST` requests. When a user submits the login form via a `POST` request, the route verifies the entered email and password against the records in the **Users** table. If the credentials are valid, the user's login count may be incremented, and a session is created. Upon successful login, the user is redirected to the dashboard or homepage. If the credentials are invalid, an error message is displayed.

```

@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 displaying a flash message confirming the logout.
- **/home1**: A protected route accessible only to logged-in users. It serves as the main dashboard or home page after successful login.
- **/about** and **/contact_us**: These routes render static pages providing information about the application and offering ways for users to get in touch or provide feedback.

```
0  @app.route('/logout')
1  def logout():
2      session.pop('user', None)
3      flash('You have been logged out!', 'info')
4      return redirect(url_for('index'))
5
6  # Application Routes
7  @app.route('/home1')
8  def home1():
9      if 'user' not in session:
10         return redirect(url_for('login'))
11         return render_template('home1.html')
12
13  @app.route('/about')
14  def about():
15      return render_template('about.html')
16
17  @app.route('/contact_us')
18  def contact():
19      return render_template('contact_us.html')
20
```

Booking page:

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.

```
# 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:

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.

```
@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,
```

Application Entry point:

```
        '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'))
```



```
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

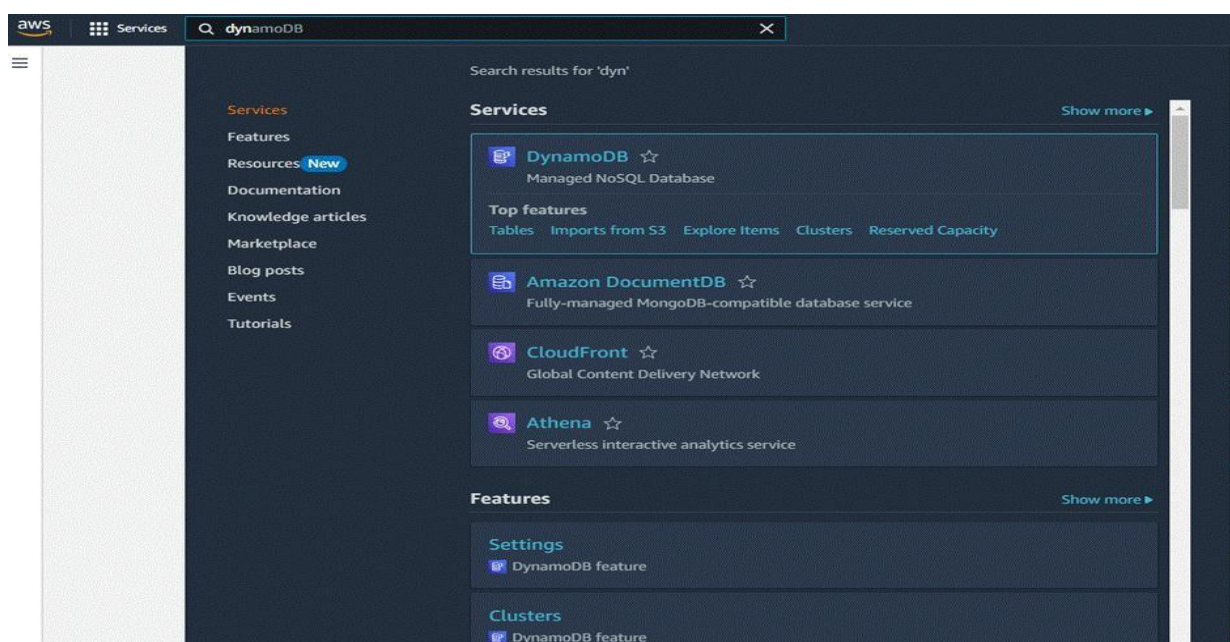
7. AUTHENTICATION:

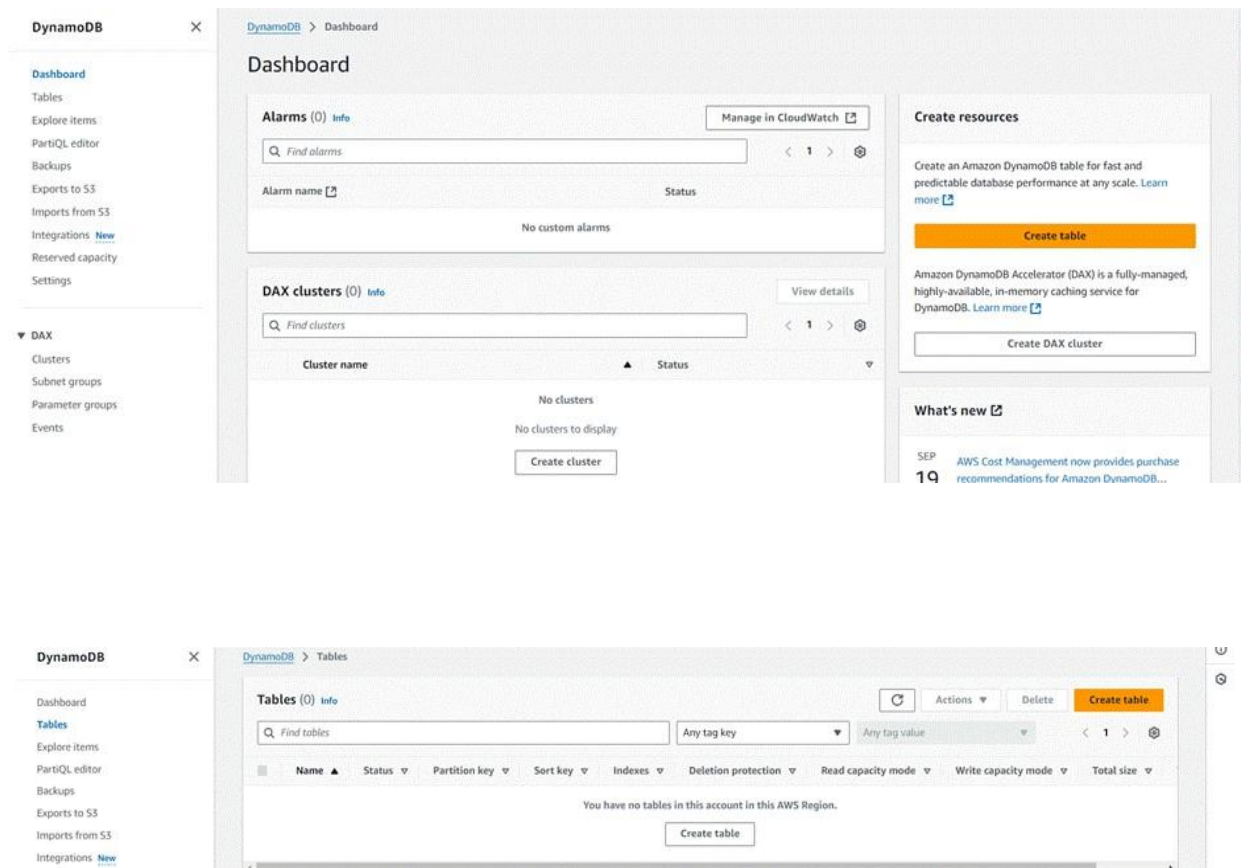
Milestone 3 : DynamoDB Database Creation and Setup

Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

Navigate to the DynamoDB

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





Create a DynamoDB table for storing data

- CreateMovieMagic_Users table with partition key “Email” with type String and click on create tables.

≡ [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.

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.

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.

1 to 255 characters and case sensitive.

Table 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

Tags

Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

No tags are associated with the resource.

Add new tag

You can add 50 more tags.

Cancel

Create table

- Follow the same steps to create a MovieMagic_Bookings table with Booking_id as the primary key for movie bookings data.

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.

MovieMagic_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_id

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 learn more, see [DynamoDB Default Settings](#).

☐ Customize settings

Use these advanced settings to create a table that meets your specific needs. To learn more, see [DynamoDB Advanced 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

Tags
Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

No tags are associated with the resource.

[Add new tag](#)
You can add 50 more tags.

[Cancel](#) [Create table](#)

Tables (4) [Info](#)

[Actions](#) [Delete](#) [Create table](#)

<input type="checkbox"/>	Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite	Re
<input type="checkbox"/>	MovieMagic_Bookings	Active	booking_id (S)	-	0	0	Off	☆	Or
<input type="checkbox"/>	MovieMagic_Users	Active	email (S)	-	0	0	Off	☆	Or

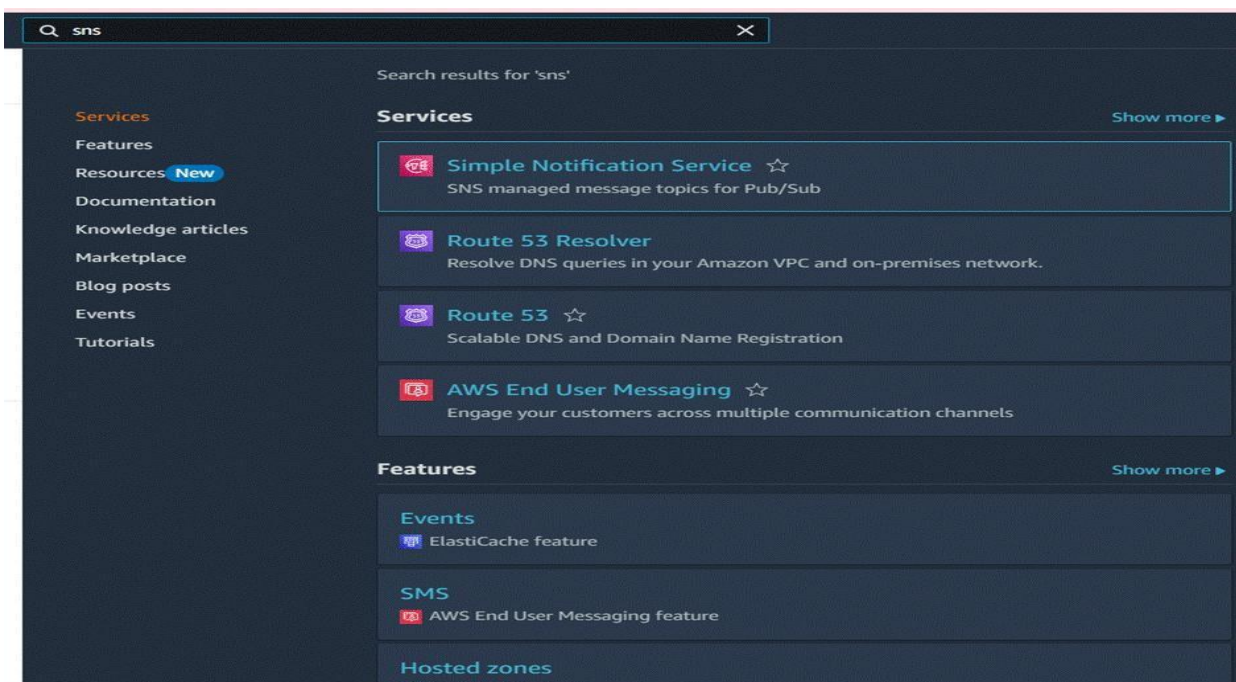
Milestone 4 : SNS Notification Setup :

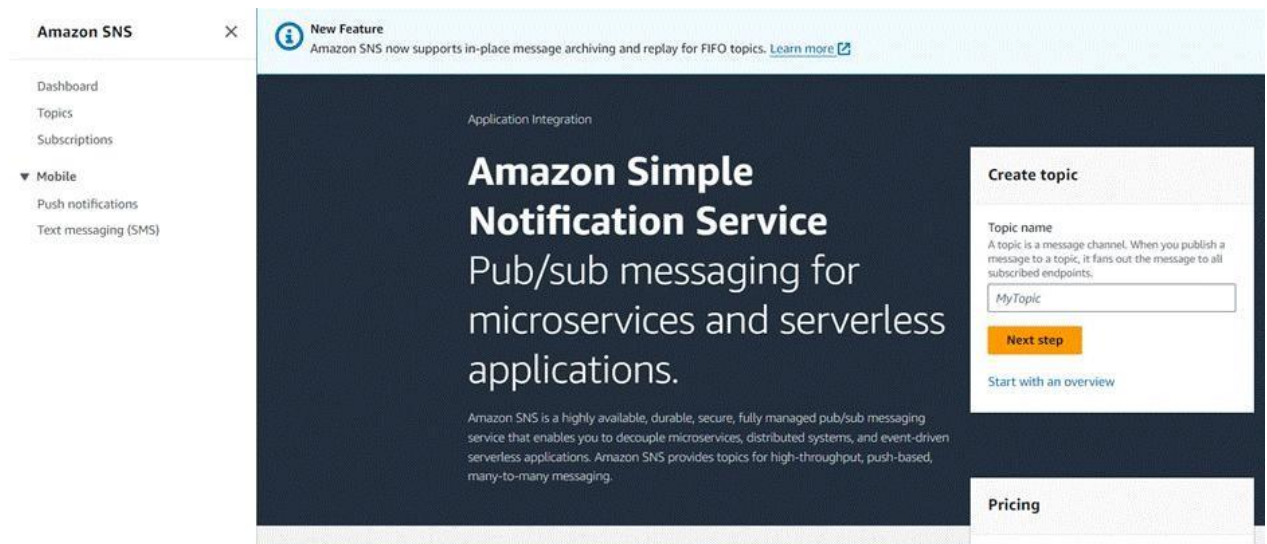
SNS Notification Setup

Amazon SNS is a fully managed messaging service that enables real-time notifications through channels like SMS, email, or app endpoints. You create topics, configure subscriptions, and integrate SNS into your app to send notifications based on specific events.

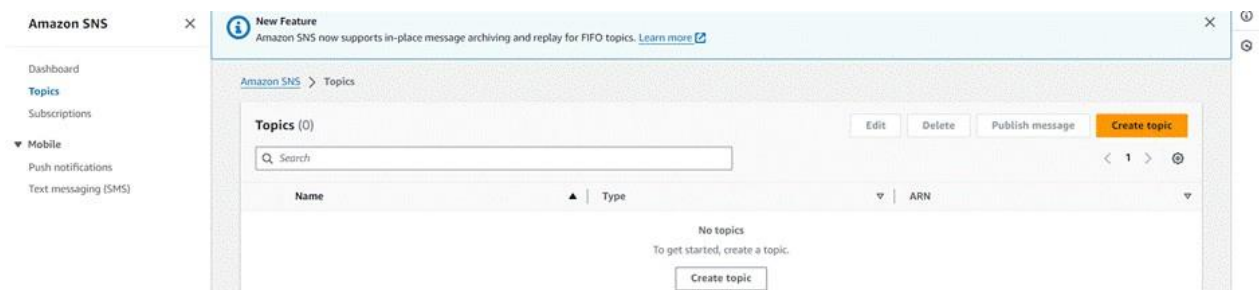
Create SNS topics for sending email notifications to users

- 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.**

Amazon SNS

Topics

Create topic

New Feature

Amazon SNS now supports High Throughput FIFO topics. [Learn more](#)

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

Maximum 256 characters. Can include alphanumeric characters, hyphens (-) and underscores (_).

Display name - optional

Info

Maximum 100 characters.

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

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

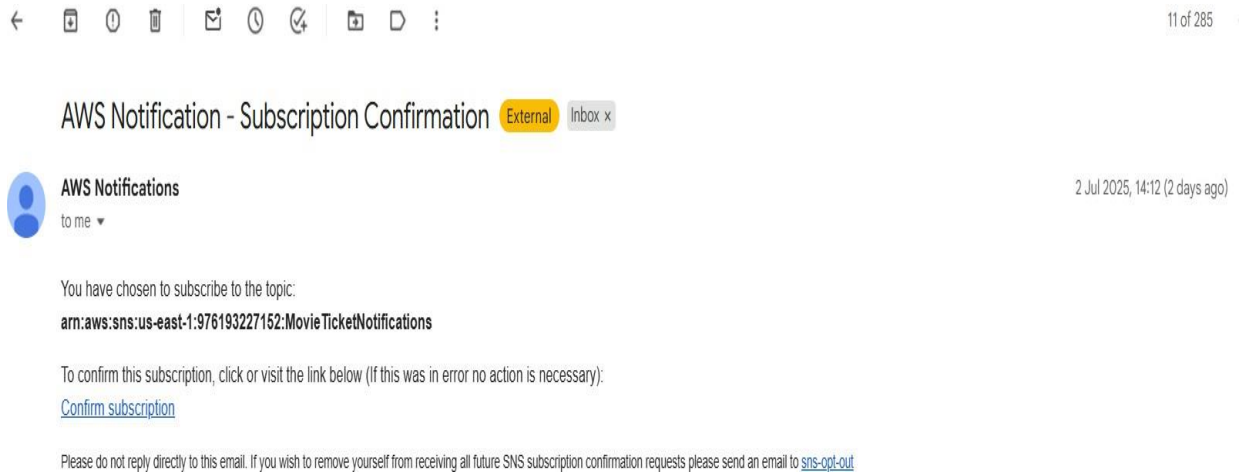
Subscribe Users And Admin

- **Subscribe users (or admin staff) to this topic via Email. When a movie ticket is booked, notifications will be sent to the user's emails.**

MovieTicketNotifications

Details	
Name MovieTicketNotifications	Display name -
ARN arn:aws:sns:ap-south-1:605134430972:MovieTicketNotifications	Topic owner 605134430972
Type Standard	

- **After subscription request for the mail confirmation**
- **Navigate to the subscribed Email account and Click on the confirm subscription in the AWS Notification- Subscription Confirmation mail.**



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

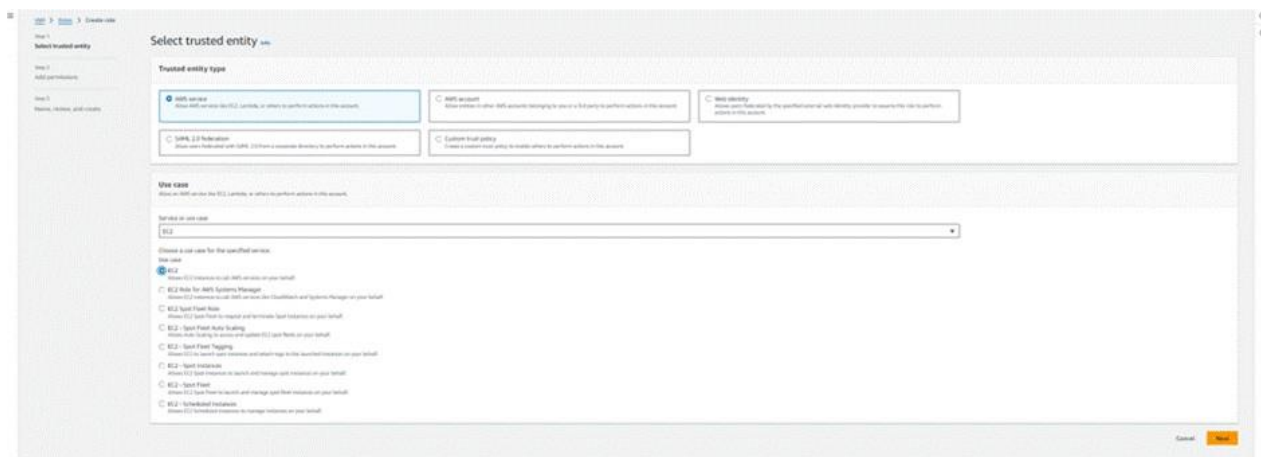
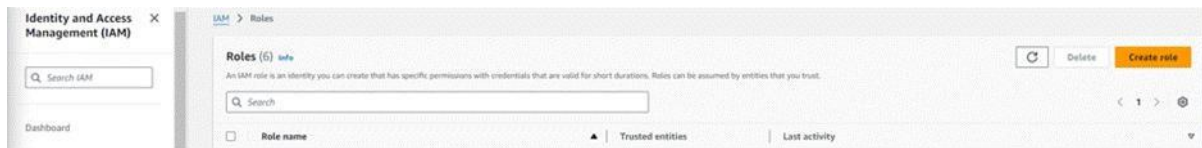
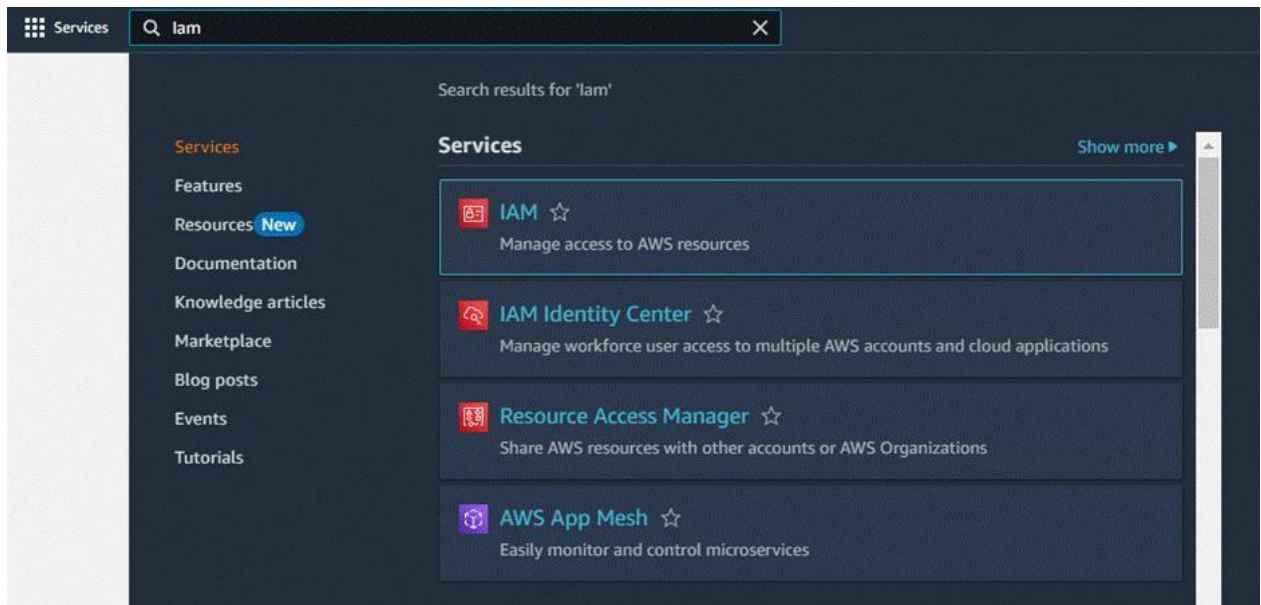
Milestone 5 : IAM Role Setup

IAM Role Setup

IAM (Identity and Access Management) role setup involves creating roles that define specific permissions for AWS services. To set it up, you create a role with the required policies, assign it to users or services, and ensure the role has appropriate access to resources like EC2, S3, or RDS. This allows controlled access and ensures security best practices in managing AWS resources.

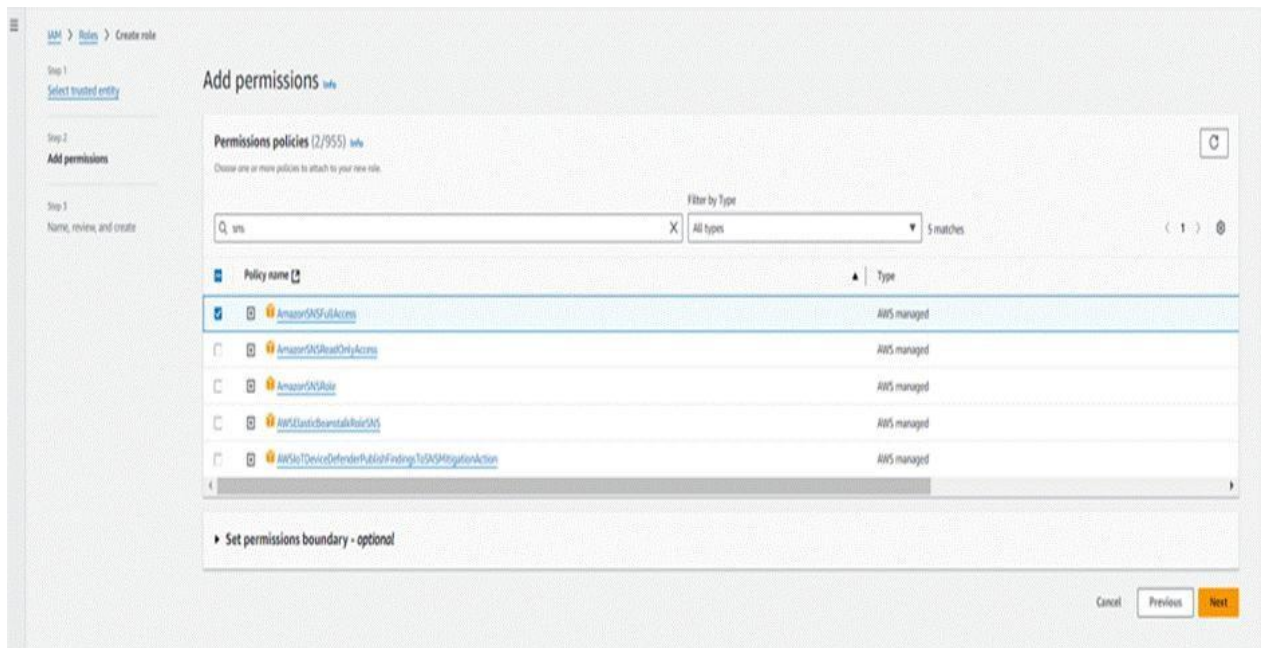
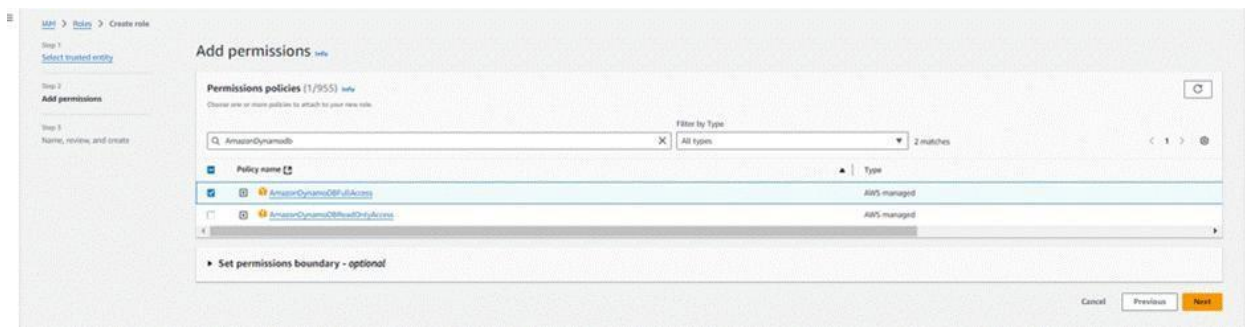
Create IAM Role

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



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.



Name, review, and create

Role details

Role name

Enter a meaningful name to identify this role.

Maximum 64 characters. Use alphanumeric and '+, @, -, _' characters.

Description

Add a short explanation for this role.

Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: _ + = , @ - / [()] ! # \$ % ^ & * { } ; ' " .

Step 1: Select trusted entities

Trust policy

```

1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Action": [
7         "sts:AssumeRole"
8       ],
9       "Principal": {
10        "Service": [
11          "ec2.amazonaws.com"
12        ]
13      }
14    ]
15  }
16 }
```


Milestone 6 : EC2 Instance setup

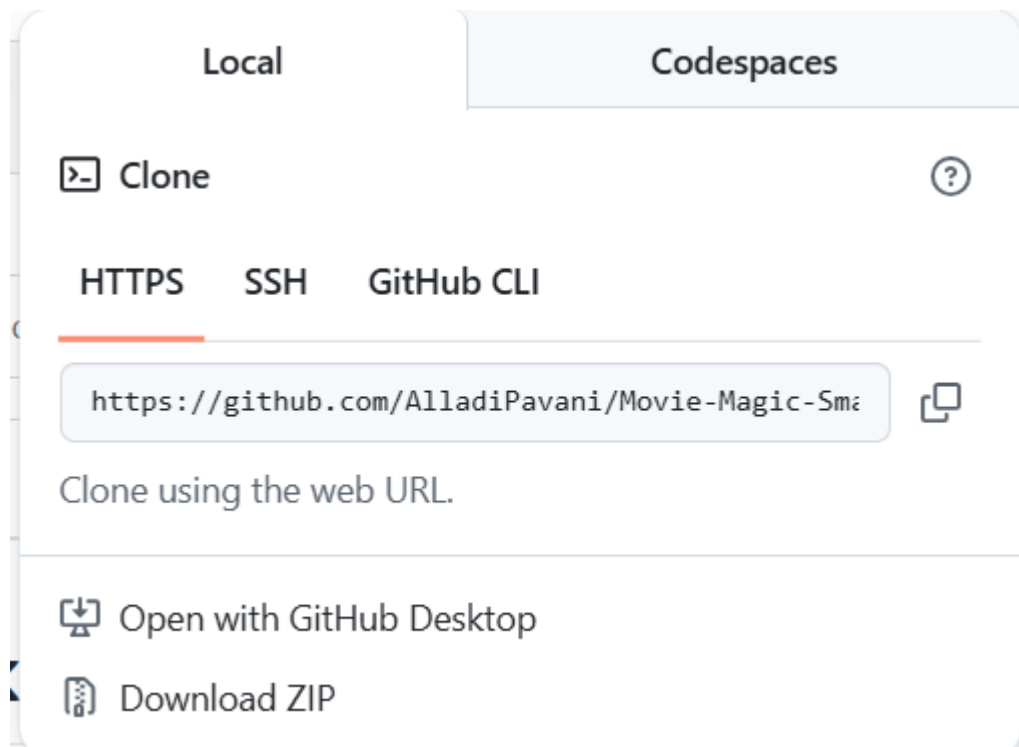
EC2 Instance setup

To set up a public EC2 instance, choose an appropriate Amazon Machine Image (AMI) and instance type. Ensure the security group allows inbound traffic on necessary ports (e.g., HTTP/HTTPS for web applications). After launching the instance, associate it with an Elastic IP for consistent public access, and configure your application or services to be publicly accessible.

Load your Project Files to GitHub

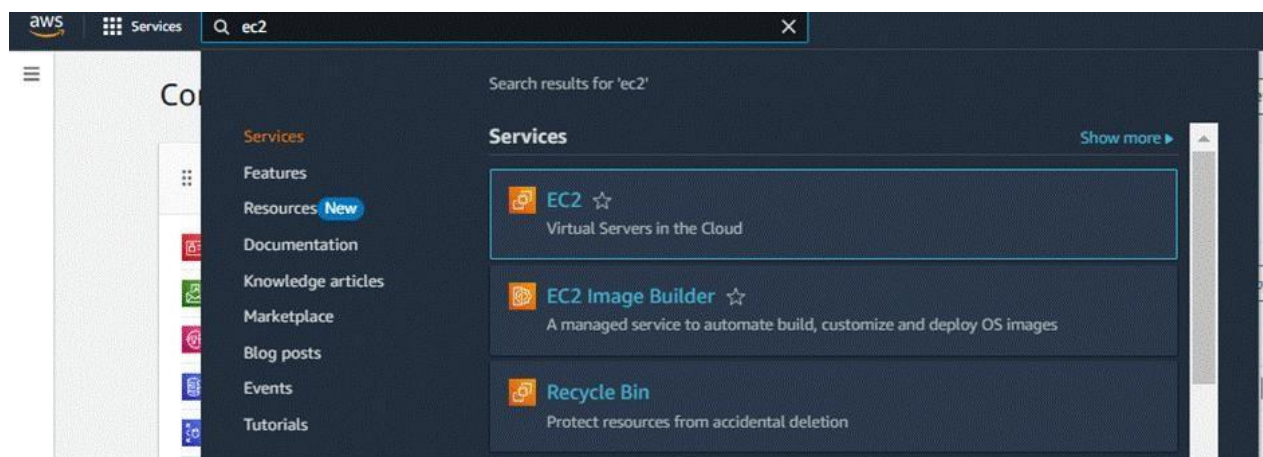
- Load your Flask app and Html files into GitHub repository.

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

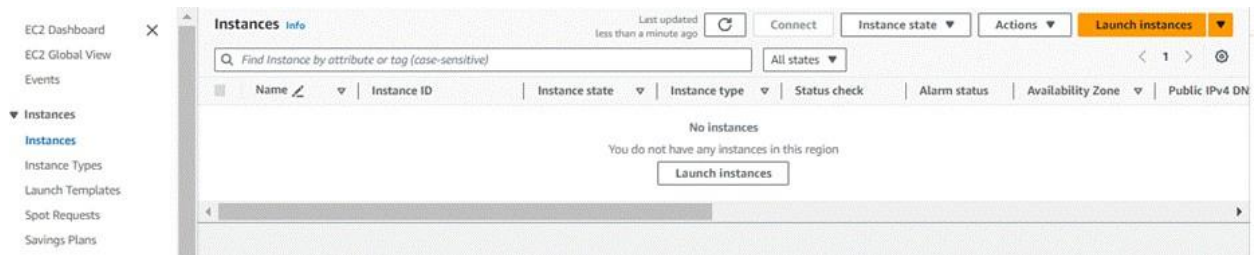


Launch an EC2 instance to host the Flask

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



- Click on Launch instance to launch EC2 instance



📌 It seems like you may be new to launching instances in EC2. Take a walkthrough to learn about EC2, how to launch instances and about best practices

Launch an instance [Info](#)

Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags [Info](#)

Name

[Add additional tags](#)

▼ Application and OS Images (Amazon Machine Image) [Info](#)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Recents

Quick Start



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

Amazon Linux

aws

macOS

Mac

Ubuntu

ubuntu

Windows

Microsoft

Red Hat

Red Hat

Browse more AMIs

Including AMIs from AWS, Marketplace and the Community

Amazon Machine Image (AMI)

Amazon Linux 2023 AMI

Free tier eligible

ami-02b49a24cfb95941c (64-bit (x86), uefi-preferred) / ami-04ad8c7fcc828fad4 (64-bit (Arm), uefi)

Virtualization: hvm ENA enabled: true Root device type: ebs

Description

Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications.

Architecture

64-bit (x86)

Boot mode

uefi-preferred

AMI ID

ami-02b49a24cfb95941c

Verified provider

- Create and download the key pair for Server access.

▼ Instance type [Info](#) | [Get advice](#)

Instance type

t2.micro

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

Free tier eligible

☒ All generations

[Compare instance types](#)

[Additional costs apply for AMIs with pre-installed software](#)

▼ Key pair (login) [Info](#)

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

Key pair name - required

Select

 [Create new key pair](#)

Create key pair



Key pair name

Key pairs allow you to connect to your instance securely.

moviemagid

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type



RSA

RSA encrypted private and public key pair



ED25519

ED25519 encrypted private and public key pair

Private key file format



.pem

For use with OpenSSH



.ppk

For use with PuTTY



When prompted, store the private key in a secure and accessible location on your computer. You will need it later to connect to your instance. [Learn more](#)

[Cancel](#)

Create key pair

Description

Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications.

Architecture

64-bit (x86)

Boot mode

uefi-preferred

AMI ID

ami-078264b8ba71bc45e

Username

ec2-user

Verified provider

▼ Instance type

Info | Get advice

Instance type

t2.micro

Family: t2 1 vCPU 1 GiB Memory Current generation: true

On-Demand Linux base pricing: 0.0124 USD per Hour

On-Demand Windows base pricing: 0.017 USD per Hour

On-Demand RHEL base pricing: 0.0268 USD per Hour

On-Demand SUSE base pricing: 0.0124 USD per Hour

Free tier eligible

All generations

Compare instance types

Additional costs apply for AMIs with pre-installed software

▼ Key pair (login)

Info

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

Key pair name - required

InstantLibrary

Create new key pair

▼ Summary

Number of instances

Info

1

Software Image (AMI)

Amazon Linux 2023 AMI 2023.5.2...read more

ami-078264b8ba71bc45e

Virtual server type (instance type)

t2.micro

Firewall (security group)

New security group

Storage (volumes)

1 volume(s) - 8 GiB

Free tier:

In your first year includes

750 hours of t2.micro (or t3.micro in the Regions in which t2.micro is unavailable) instance usage on free tier AMIs per month, 750 hours of public IPv4 address usage per month, 30 GiB of EBS storage, 2 million IOs, 1 GB of snapshots, and 100 GB of bandwidth to the internet.

Can cancel

Preview code

Launch instance

Configure security groups for HTTP, And SSH Access:

▼ Network settings

Info

VPC - required

Info

vpc-03cdc7b6f19dd7211

(default)

Subnet

Info

No preference

Create new subnet

Auto-assign public IP

Info

Enable

Additional charges apply when outside of free tier allowance

Firewall (security groups)

Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

Create security group

Select existing security group

Security group name - required

launch-wizard

This security group will be added to all network interfaces. The name can't be edited after the security group is created. Max length is 255 characters. Valid characters: a-z, A-Z, 0-9, spaces, and _-:/()#,@!+=&:()!\$*

Description - required

Info

launch-wizard created 2024-10-13T17:49:56.622Z

Inbound Security Group Rules

Security group rule 1 (TCP; 22, 0.0.0.0/0)
Remove

Type ssh

Source type Anywhere

Protocol TCP

Source 0.0.0.0/0

Port range 22

Description - optional e.g. SSH for admin desktop

Security group rule 2 (TCP; 80, 0.0.0.0/0)
Remove

Type HTTP

Source type Custom

Protocol TCP

Source 0.0.0.0/0

Port range 80

Description - optional e.g. SSH for admin desktop

Security group rule 3 (TCP; 5000, 0.0.0.0/0)
Remove

Type Custom TCP

Source type Custom

Protocol TCP

Source 0.0.0.0/0

Port range 5000

Description - optional e.g. SSH for admin desktop

Add security group rule

EC2
Launch an Instance

Success
Successfully initiated launch of instance i-001b61027hcx299

Launch log

Next Steps
What would you like to do next with this instance, for example "create alarm" or "create backup"

Create billing and free tier usage alerts
To manage costs and avoid surprise bills, set up email notifications for billing and free tier usage thresholds.
Create billing alerts

Connect to your instance
Once your instance is running, log into it from your local computer.
Connect to instance
Learn more

Connect an RDS database
Configure the connection between an EC2 instance and a database to allow traffic flow between them.
Connect an RDS database
Create a new RDS database
Learn more

Create EBS snapshot policy
Create a policy that automates the creation, retention, and deletion of EBS snapshots.
Create EBS snapshot policy

Manage detailed monitoring
Enable or disable detailed monitoring for the instance. If you enable detailed monitoring, the Amazon EC2 console displays monitoring graphs with a 1-minute period.
Manage detailed monitoring

Create Load Balancer
Create a application, network gateway or classic Elastic Load Balancer.
Create Load Balancer

Create AWS budget
AWS Budgets allows you to create budgets, forecast spend, and take action on your costs and usage from a single location.
Create AWS budget

Manage CloudWatch alarms
Create or update Amazon CloudWatch alarms for the instance.
Manage CloudWatch alarms

Disaster recovery for your instances
Recover the instances you just launched into a different Availability Zone or a different Region using AWS Elastic Disaster Recovery (DRS).
Disaster recovery for your instances

Monitor for suspicious runtime activities
Amazon GuardDuty enables you to continuously monitor for malicious runtime activity and unauthorized behavior, with near real-time visibility into on-host activities occurring across your Amazon EC2 workload.
Monitor for suspicious runtime activities

Get instance screenshot
Capture a screenshot from the instance and view it as an image. This is useful for troubleshooting an unreachable instance.
Get instance screenshot

Get system log
View the instance's system log to troubleshoot issues.
Get system log

View all instances

- 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.

Instances (1/1) [Info](#)

Last updated
less than a minute ago

Connect

Instance state ▼

Actions ▼

Launch i

Find Instance by attribute or tag (case-sensitive)

All states ▼

<input checked="" type="checkbox"/>	Name ↗	Instance ID	Instance state ▼	Instance type ▼	Status check	Alarm status	Availability Zone
<input checked="" type="checkbox"/>	MovieMagic	i-047dbc81fe3dcad56	⊖ Stopped 🔍 🔍	t2.micro	-	View alarms +	ap-south-1b

i-047dbc81fe3dcad56 (MovieMagic)

[Details](#)

[Status and alarms](#)

[Monitoring](#)

[Security](#)

[Networking](#)

[Storage](#)

[Tags](#)

▼ Instance summary [Info](#)

Instance ID

[🔍](#) i-047dbc81fe3dcad56

Public IPv4 address

-

Private IPv4 addresses

[🔍](#) 172.31.1.152

IPv6 address

-

Instance state

⊖ Stopped

Public IPv4 DNS

-

Hostname type

Private IP DNS name (IPv4 only)

- Now connect the EC2 with the files

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 ~]$
```

Clone Your Flask Project from GitHub

git clone: <https://github.com/Bhavya-B1309/Movie-Magic-Smart-Movie-Ticket-Booking-System>

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

This will download your project to the EC2 instance.

- To navigate to the project directory, run the following command: `cd MovieMagic`
- 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=5000`

```
a, click, blinker, python-dotenv, flask, boto3
Successfully installed blinker-1.9.0 boto3-1.39.1 botocore-1.39.1 click-8.1.8 flask-3.1.1 importlib-metadata-8.7.0 itsda
ngerous-2.2.0 jinja2-3.1.6 markupsafe-3.0.2 python-dotenv-1.1.1 s3transfer-0.13.0 werkzeug-3.1.3 zipp-3.23.0
WARNING: You are using pip version 21.3.1; however, version 25.1.1 is available.
You should consider upgrading via the '/usr/bin/python3 -m pip install --upgrade pip' command.
[ec2-user@ip-172-31-94-148 ~]$ git clone https://github.com/AlladiPavani/Movie-Magic-Smart-Movie-Ticket-Booking-System.g
it
Cloning into 'Movie-Magic-Smart-Movie-Ticket-Booking-System'...
remote: Enumerating objects: 195, done.
remote: Counting objects: 100% (195/195), done.
remote: Compressing objects: 100% (183/183), done.
```

- Verify the Flask app is running: **`http://your-ec2-public-ip`**
- Run the Flask app on the EC2 instance

```
[ec2-user@ip-172-31-94-148 ~]$ git clone https://github.com/AlladiPavani/Movie-Magic-Smart-Movie-Ticket-Booking-System.g
it
Cloning into 'Movie-Magic-Smart-Movie-Ticket-Booking-System'...
remote: Enumerating objects: 195, done.
remote: Counting objects: 100% (195/195), done.
remote: Compressing objects: 100% (183/183), done.
remote: Total 195 (delta 28), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (195/195), 1.82 MiB | 21.91 MiB/s, done.
Resolving deltas: 100% (28/28), done.
[ec2-user@ip-172-31-94-148 ~]$ ls
Movie-Magic-Smart-Movie-Ticket-Booking-System
[ec2-user@ip-172-31-94-148 ~]$ cd Movie-Magic-Smart-Movie-Ticket-Booking-System
[ec2-user@ip-172-31-94-148 Movie-Magic-Smart-Movie-Ticket-Booking-System]$ cd Moviemagic
[ec2-user@ip-172-31-94-148 Moviemagic]$ 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://172.31.94.148:5000
Press CTRL+C to quit
```

- Access the website through: **`your-ec2-public-ip`**

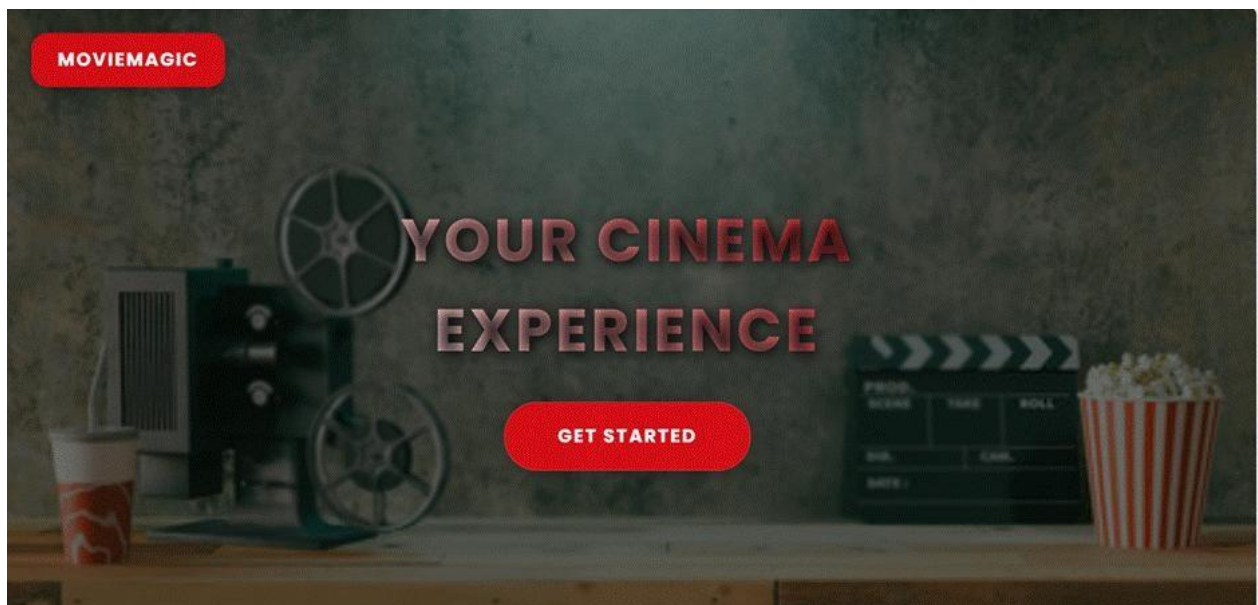
- Public IPs: [http:// 18.215.178.87:5000](http://18.215.178.87:5000)

Milestone 8 : Testing and Deployment

Testing and deployment involve verifying that your application works as expected before making it publicly accessible. Start by testing locally or on a staging environment to catch bugs and ensure functionality. Once tested, deploy the application to an EC2 instance, configure necessary services, and perform a final round of live testing to confirm everything runs smoothly in the production environment.

Functional testing to verify the Project

Index Page:



Register Page:

The image shows the register page of the 'MOVIE MAGIC' application. The page has a dark background. In the center, there is a dark gray rounded rectangle with a red border. At the top of this rectangle, the text 'CREATE ACCOUNT' is displayed in a bold, red font. Below this text are three input fields: 'Full Name', 'Email', and 'Password'. Each input field has a red border and a red placeholder text. Below the input fields is a red rounded rectangle with the text 'SIGN UP' in white. At the bottom of the register form, there is a link that says 'Already have an account? Login Here' in a small, red font.

Login Page:

WELCOME BACK

LOGIN

New user? [Create Account](#)

Home page:

MovieMagic

Home Logout

Welcome to MovieMagic

Your one-stop destination for movie tickets in Hyderabad

Telugu

Enter City (e.g., Hyderabad)

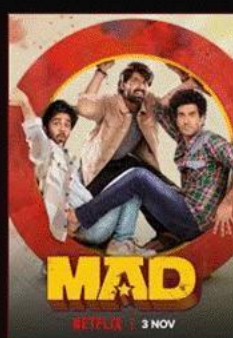
Search

Top Trending Movies



Court

Action, Drama



Mad^2

Action, Historical



Robinhood

Action, Thriller



Presence

Action, Thriller

Known Issues :

1. Booking Logic Issues

- Overbooking of seats due to lack of concurrency control.
- Double booking when the "Book" button is clicked multiple times.
- Delay in seat availability status resulting in booking conflicts.

2. User Authentication & Security

- Passwords stored in plain text instead of using secure hashing.
- Insecure session management leading to session hijacking.
- Absence of email or phone verification allows fake account creation.

3. Date and Time Handling Issues

- Timezone mismatches causing incorrect show timings.
- Allowing bookings for past shows due to missing validations.

4. Communication Failures

- Email or SMS confirmations not delivered due to server issues.
- Incorrect ticket details (movie name, seat, timing) sent to users.

5. Payment Integration Issues

- Successful payment without successful seat booking.
- No support for refunds in case of booking failure or cancellation.
- Insecure payment flow vulnerable to bypass or manipulation.

6. User Interface and Experience Issues

- Booking page not responsive on mobile devices.
- Confusing seat layout or lack of visual seat selection confirmation.
- Unintuitive or buggy date/time picker components.

7. Performance and Scalability Issues

- Lag in rendering seat maps for large theaters.
- Application crashes or slows down during high traffic (e.g., big movie releases).

8. Data Consistency Issues

- Inconsistent data between bookings and movie listings.
- Pricing errors due to incorrect tax, discount, or category application.

Future Enhancements :

1. Payment Gateway Integration

- Integrate with secure and popular payment gateways (e.g., Razorpay, Stripe, PayPal).
- Add support for UPI, net banking, credit/debit cards, and wallet payments.

2. Real-Time Seat Locking System

- Temporarily lock selected seats during the checkout process to prevent overbooking.
- Auto-release locked seats if the user does not complete the booking within a time limit.

3. Movie Recommendation Engine

- Suggest movies to users based on watch history, ratings, and genres.
- Implement basic AI/ML models for personalized recommendations.

4. User Profile & Booking History

- Allow users to view past bookings, favorite movies, and account preferences.
- Enable easy rebooking of previously watched movies or theaters.

5. Multi-language and Regional Support

- Offer the UI in multiple languages (English, Hindi, Telugu, etc.).
- Filter movies by language and region.

6. Theater Admin Panel

- Provide theater owners with a dashboard to manage movie schedules, seat layouts, and bookings.
- View analytics like seat occupancy, booking trends, and revenue.

7. Reviews and Ratings

- Allow users to rate and review movies after watching.
- Show average ratings and popular reviews on the movie details page.

8. Email and SMS Notifications

- Send booking confirmations, reminders, and promotional offers.
- Notify users about ticket cancellations or movie time changes.

9. Mobile App Support

- Develop Android and iOS apps for faster access and native features like push notifications.
- Sync bookings across web and mobile platforms.

10. Ticket Cancellation and Refund Policy

- Allow users to cancel bookings with partial/full refund based on timing.
- Automate the refund process based on payment method.

11. Advanced Filtering and Search

- Search by movie, theater, location, language, show timing, or rating.
- Add filters for seat types (VIP, recliner, couple, etc.).

12. Loyalty Program

- Introduce reward points for frequent users.
- Offer discounts, early access to bookings, or freebies for loyal customers.

13. QR Code and Digital Ticketing

- Generate QR code tickets for scanning at the theater entrance.
- Allow users to download, email, or save tickets digitally.

14. Offline Booking Sync (Theater Kiosk)

- Build a system for local theater counters to sync bookings with the main server.
- Prevent double bookings between online and offline sales.

15. Accessibility Features

- Add screen reader support, larger fonts, and high-contrast mode.

- Provide filters for accessible seating for users with disabilities.

Reference :

Github link : <https://github.com/Bhavya-B1309/Movie-Magic-Smart-Movie-Ticket-Booking-System>