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.

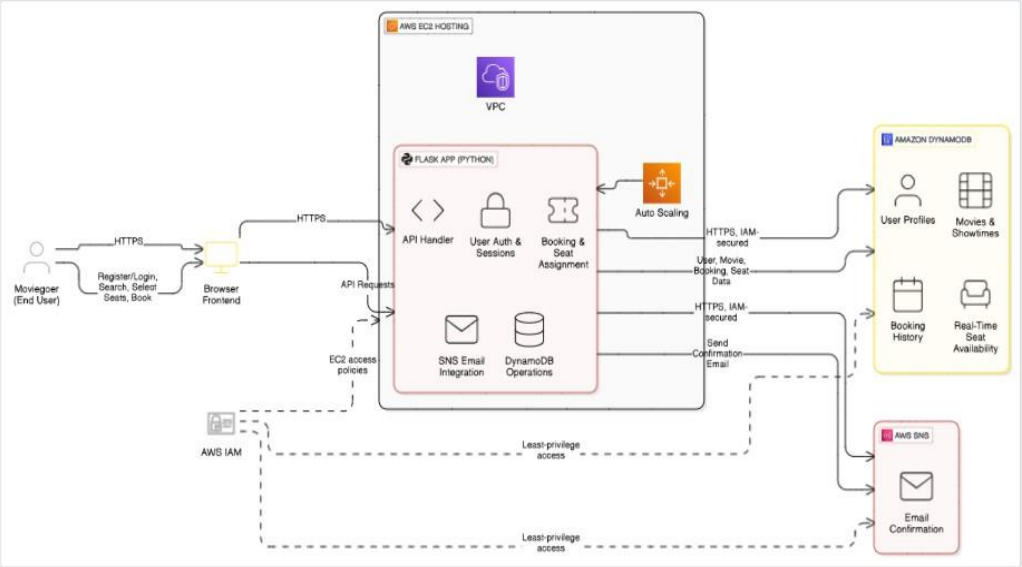
Through the integration of **AWS EC2**, the platform guarantees high availability and scalability, ensuring smooth operation even during peak traffic. The **Flask** backend facilitates efficient data handling and seamless navigation, allowing users to select movies, view showtimes, choose seats interactively, 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.

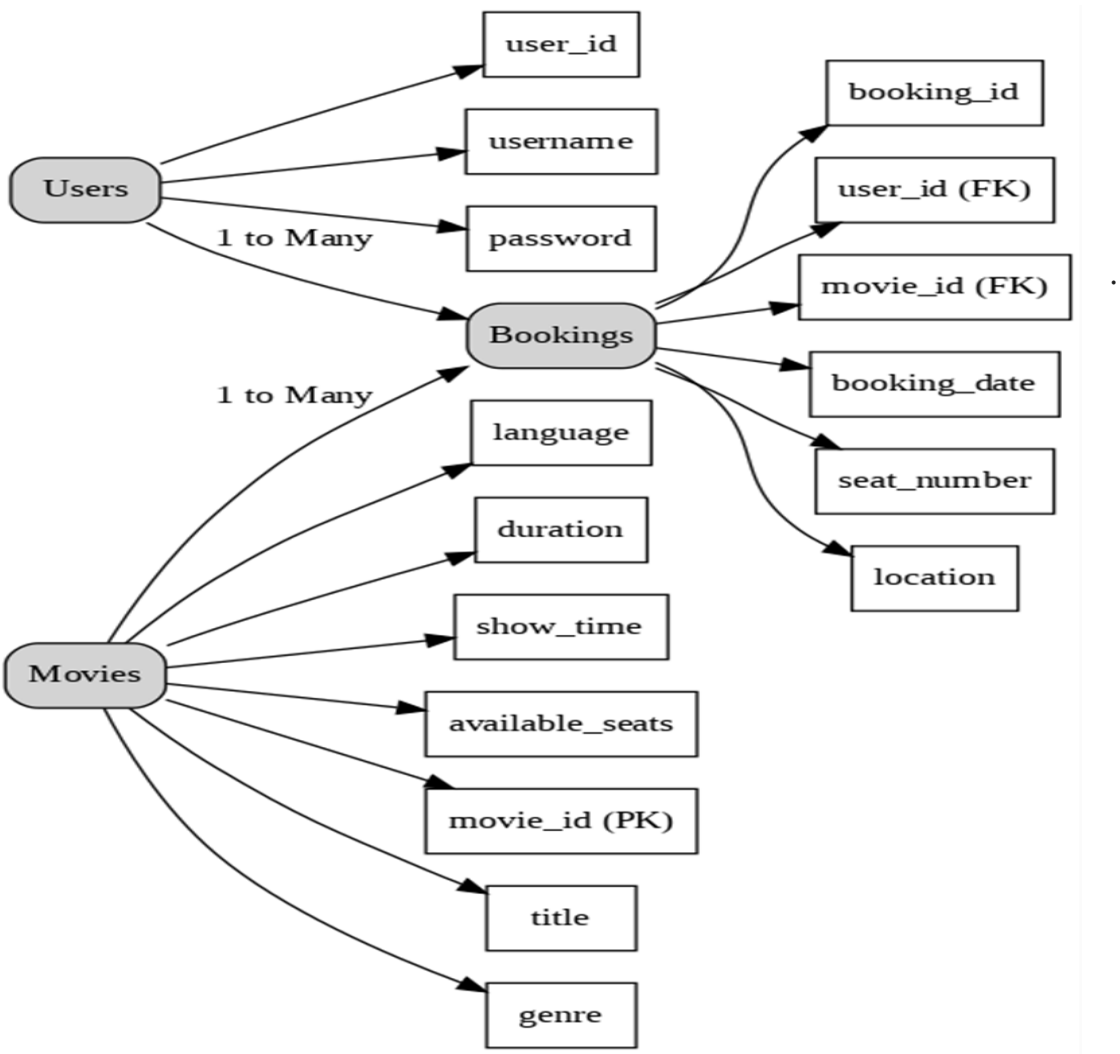
3. ARCHITECTURE:

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.

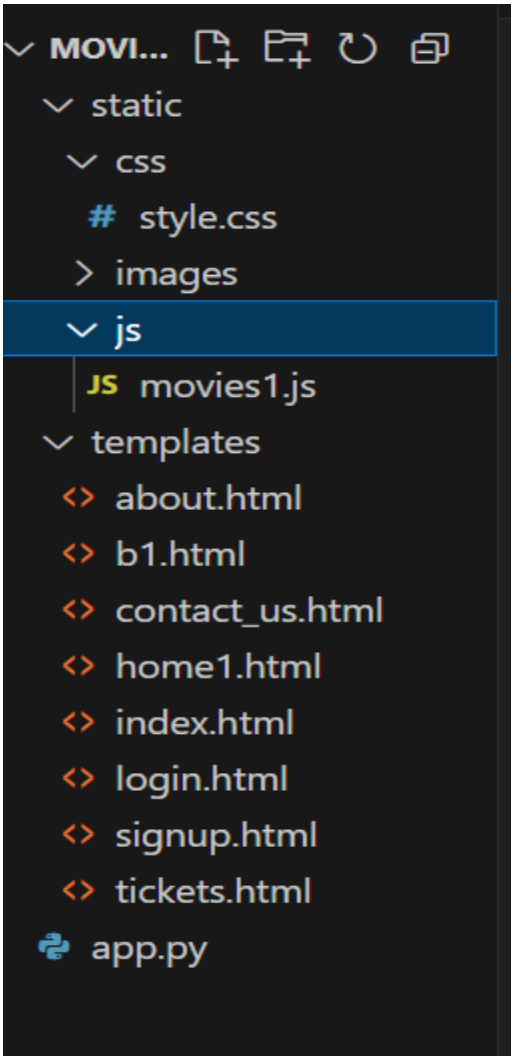


❖ PRE-REQUISTIC:

- 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:

```python
# 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.

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.

```python
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:
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'))
```

```
            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):

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.

```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:

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:

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,
```

Application Entry point:

```python
            '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'))
```

```python
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

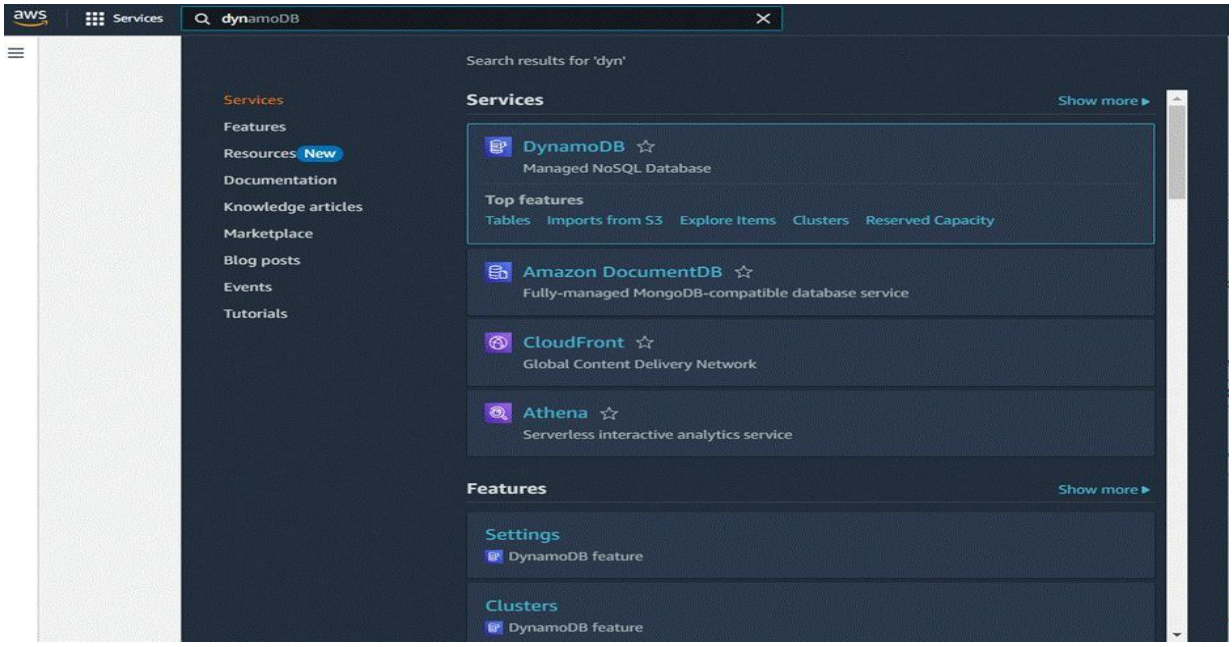
7. AUTHENTICATION:

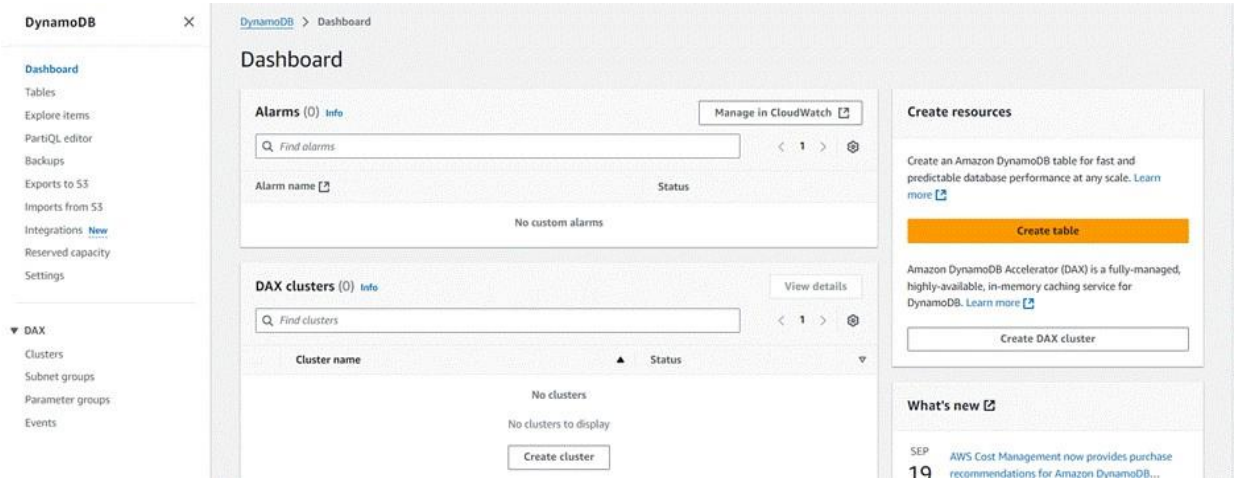
Milestone 3 : DynamoDB Database Creation and Setup

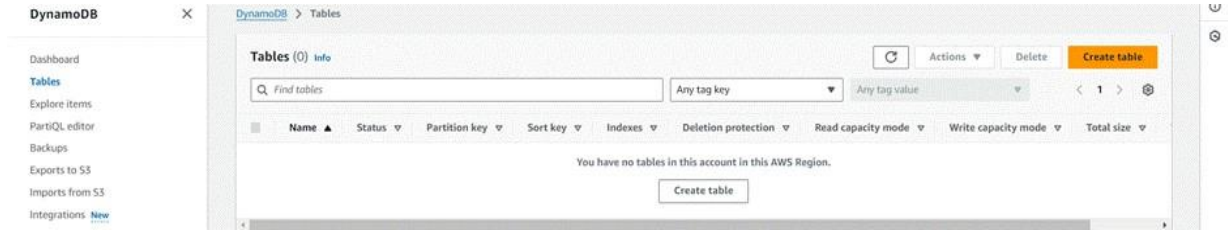
Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

Navigate to the DynamoDB

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

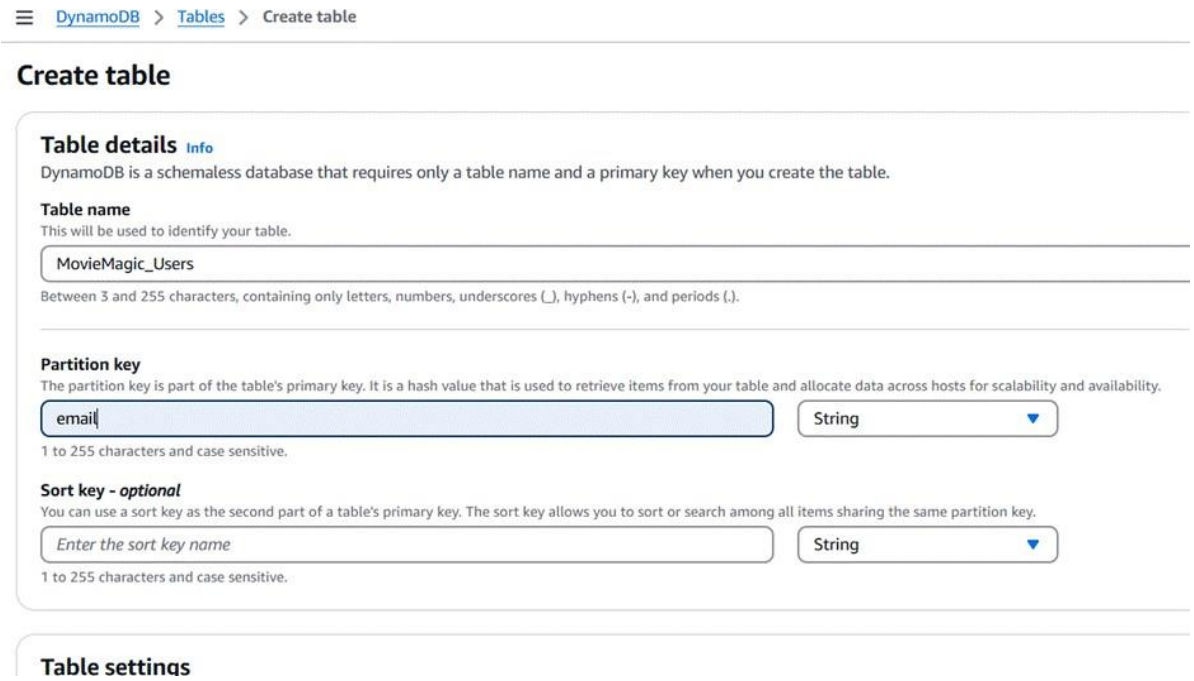






Create a DynamoDB table for storing data

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



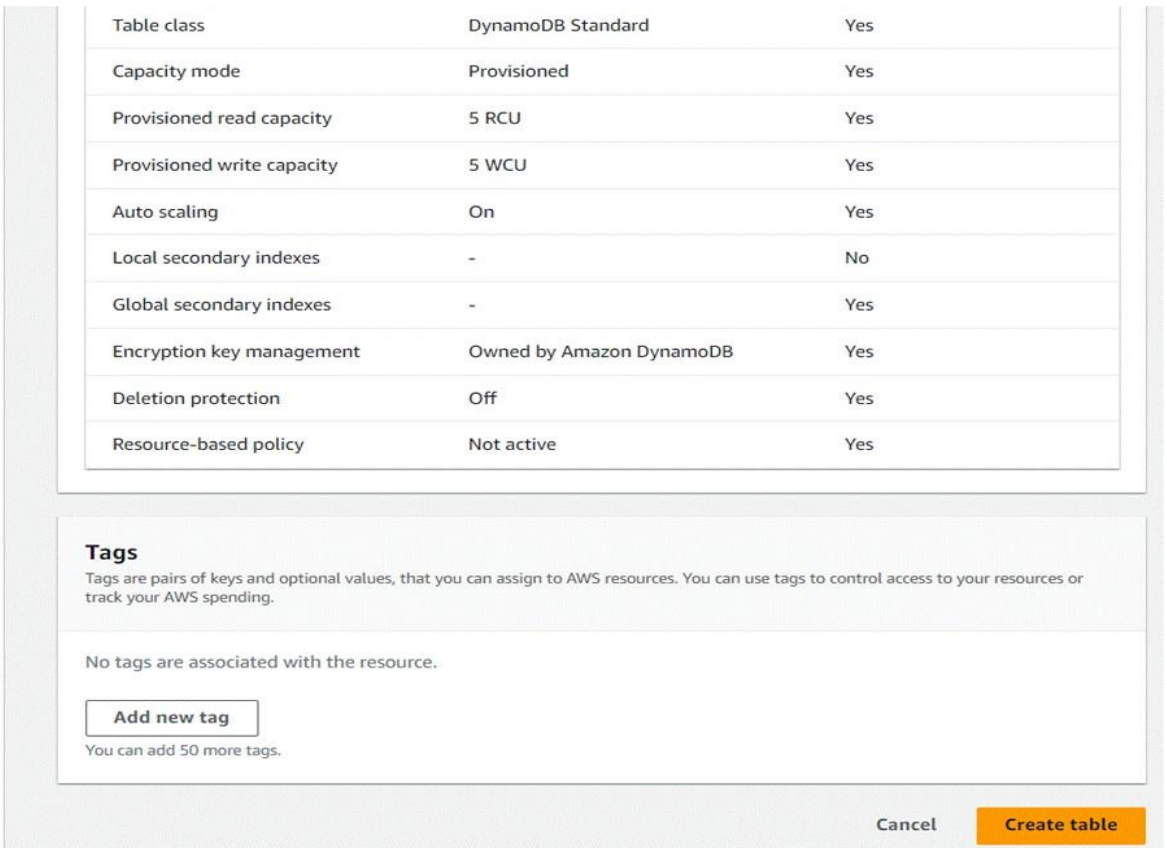


Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based 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.

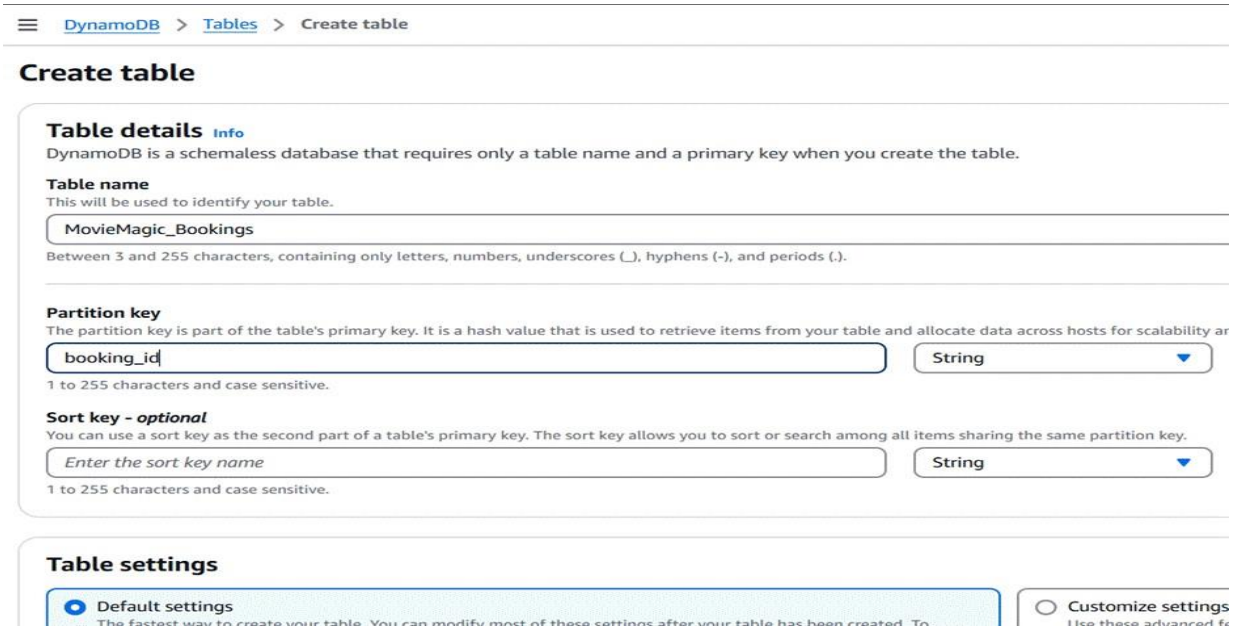
No tags are associated with the resource.

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.



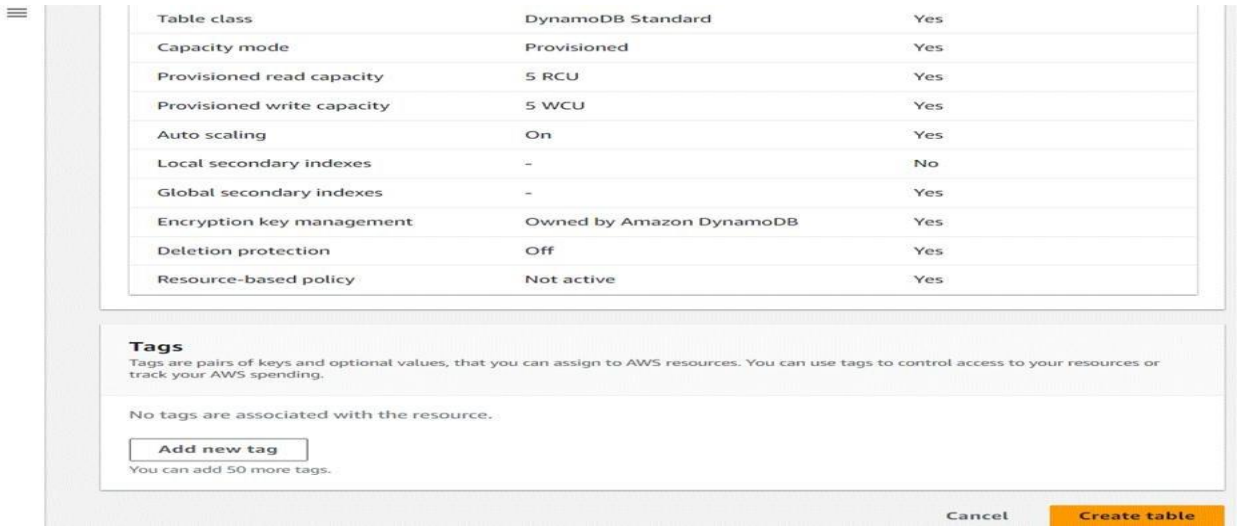


Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based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

Tags

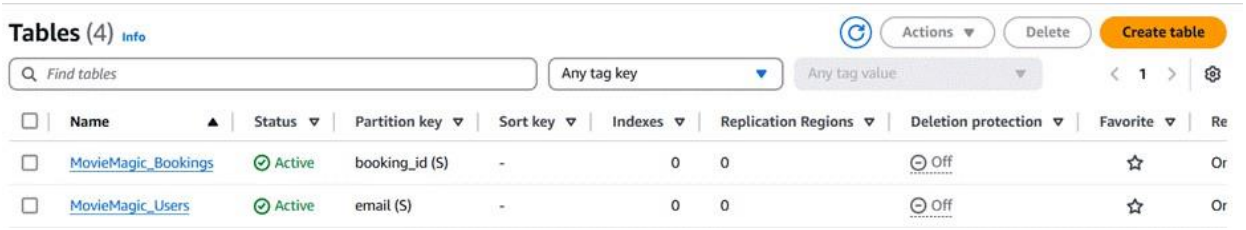
Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

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

Find tables | Any tag key | Any tag value | 1

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite	Re
MovieMagic_Bookings	Active	booking_id (S)	-	0	0	Off		Or
MovieMagic_Users	Active	email (S)	-	0	0	Off		Or

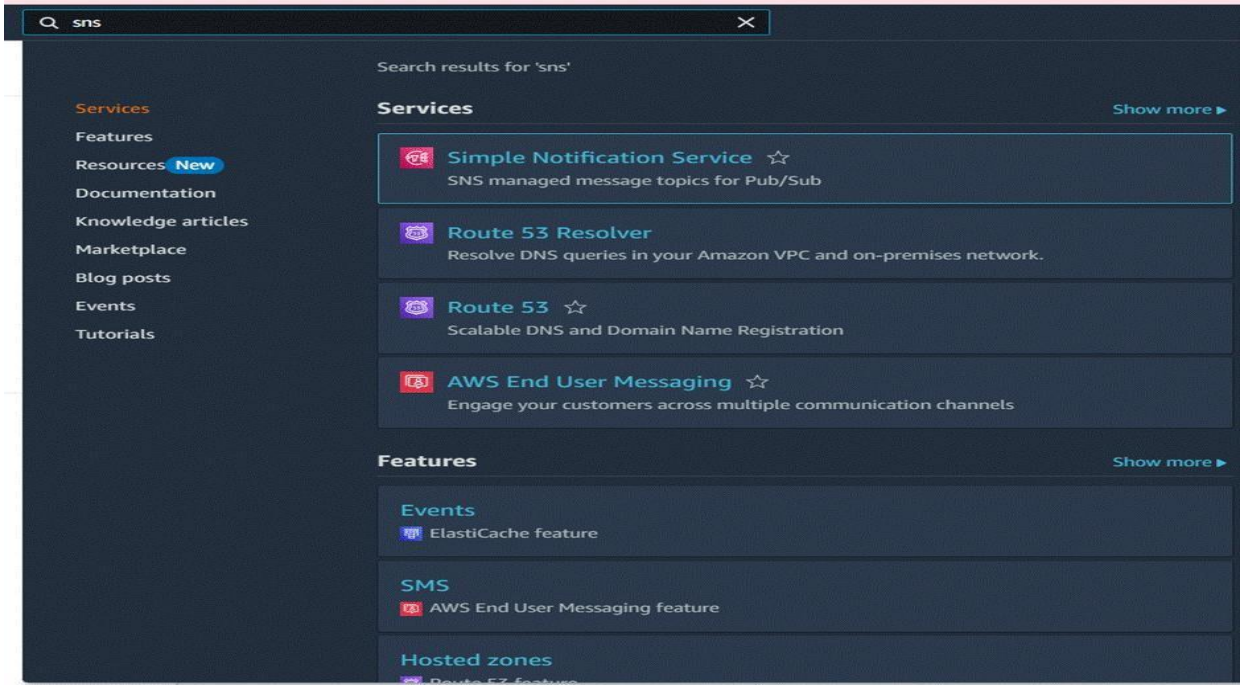
Milestone 4 : SNS Notification Setup :

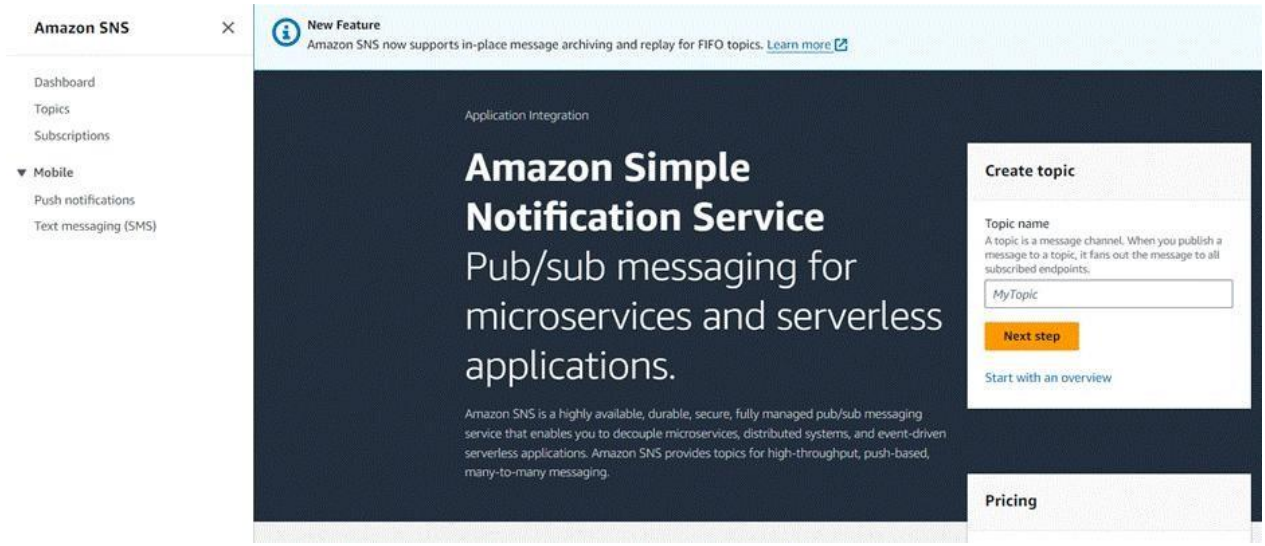
SNS Notification Setup

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.

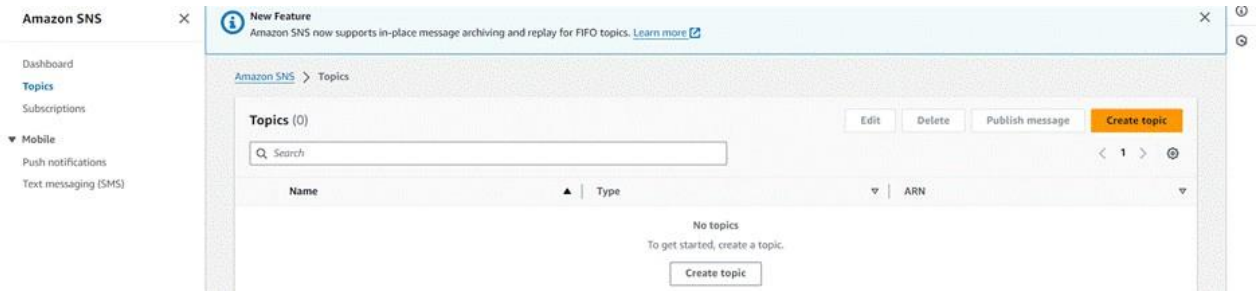
Create SNS topics for sending email notifications to users

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

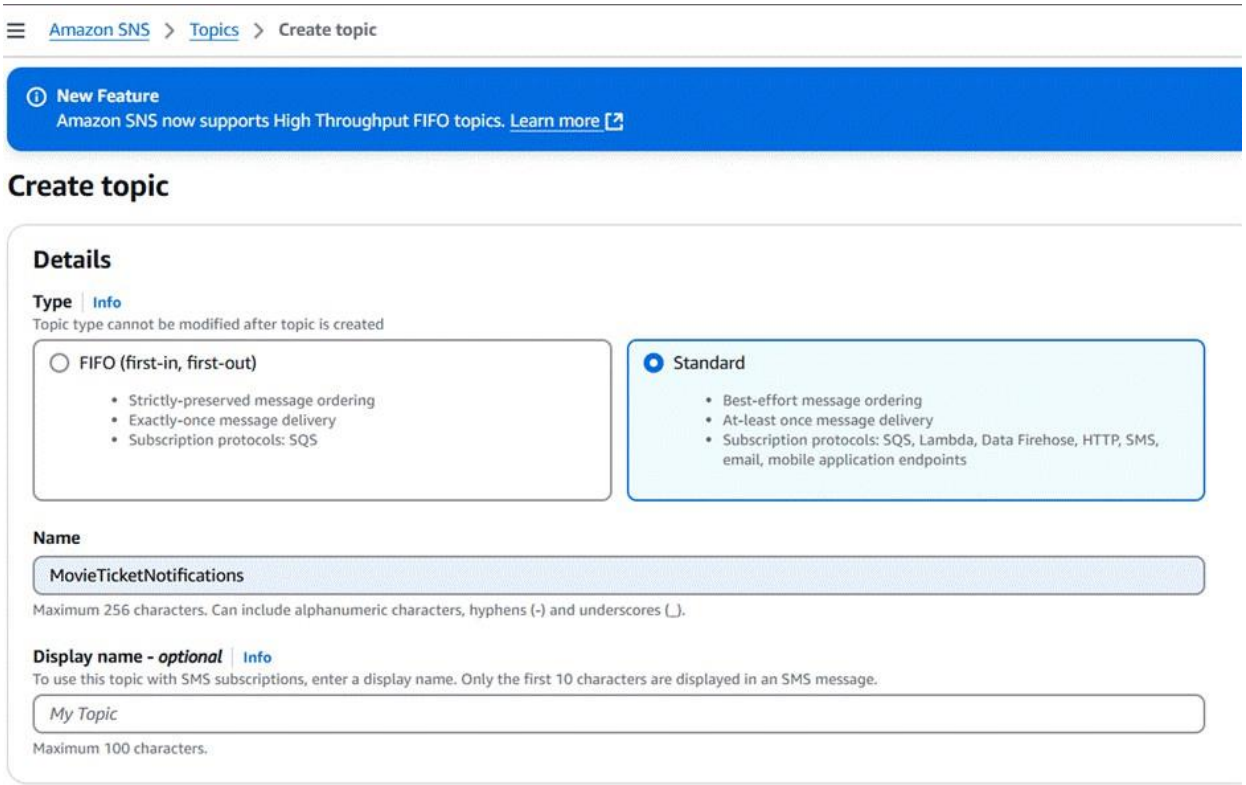


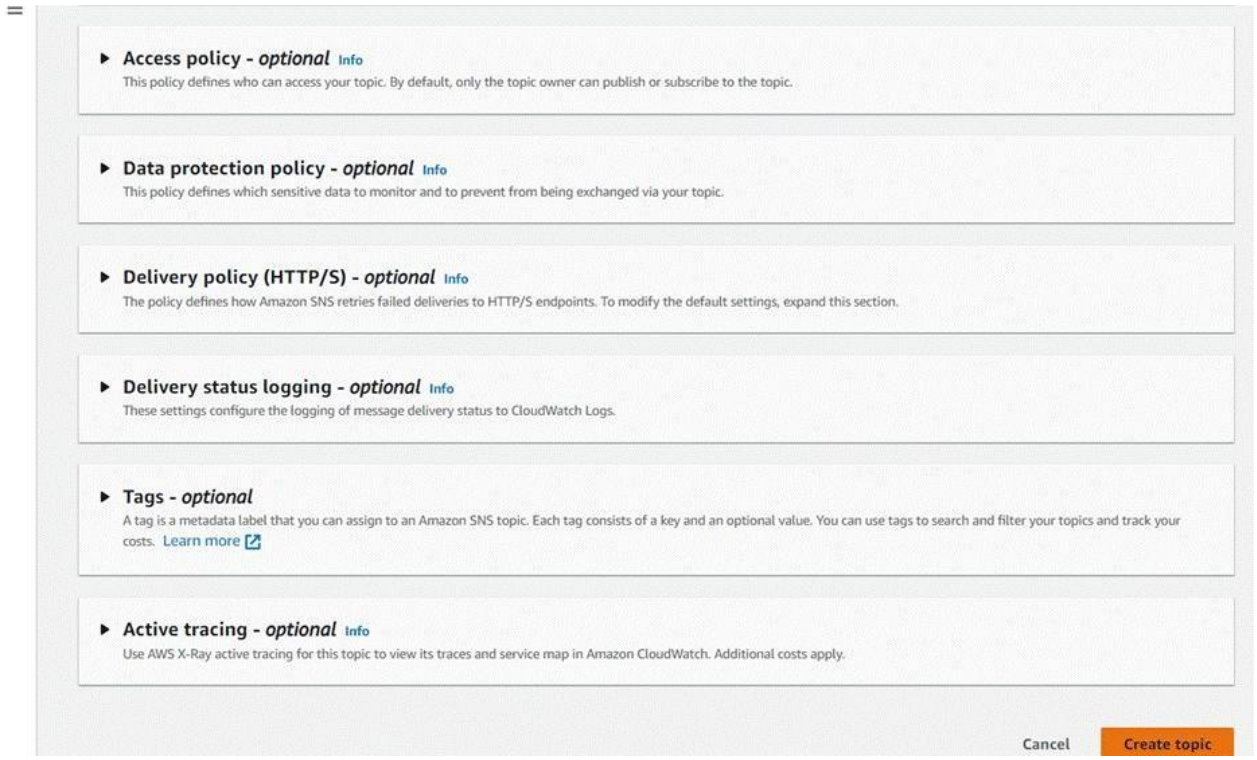


- **Click on Create Topic and choose a name for the topic**.



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

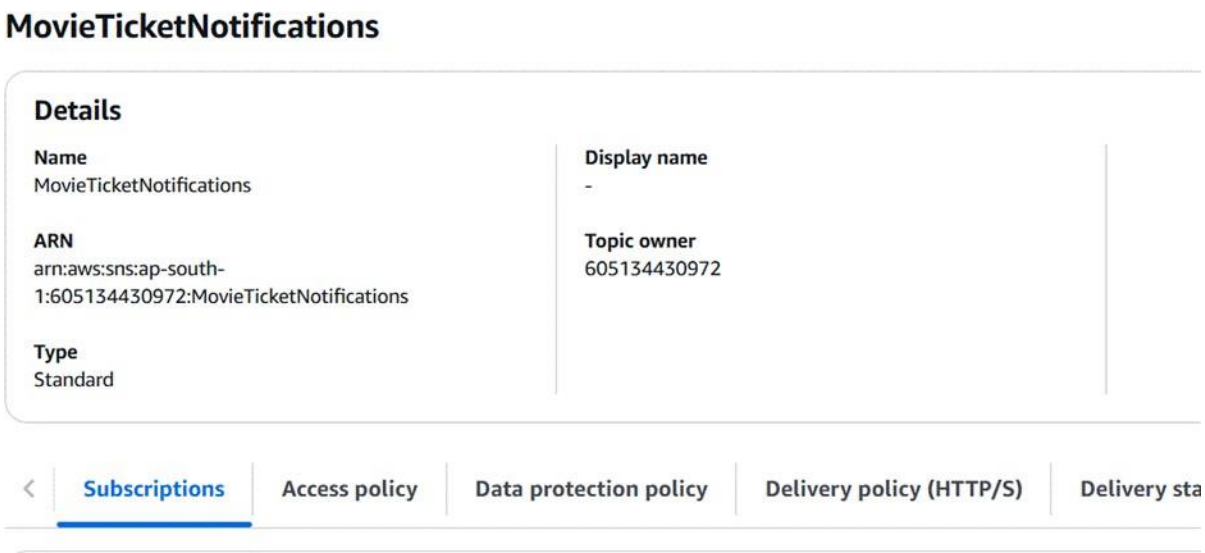




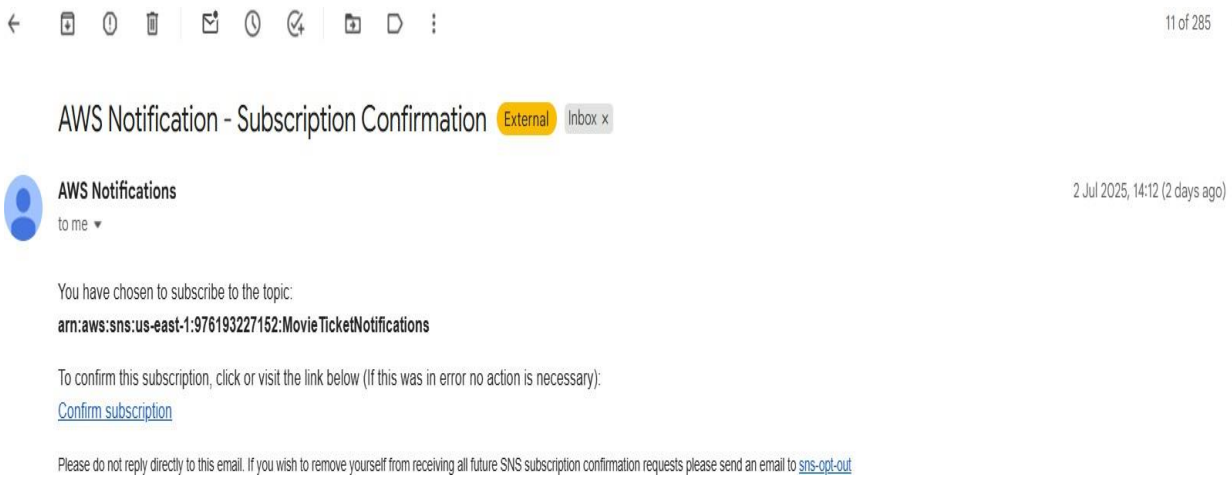
- **Configure the SNS topic and note down the Topic ARN.**

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.**



- **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.

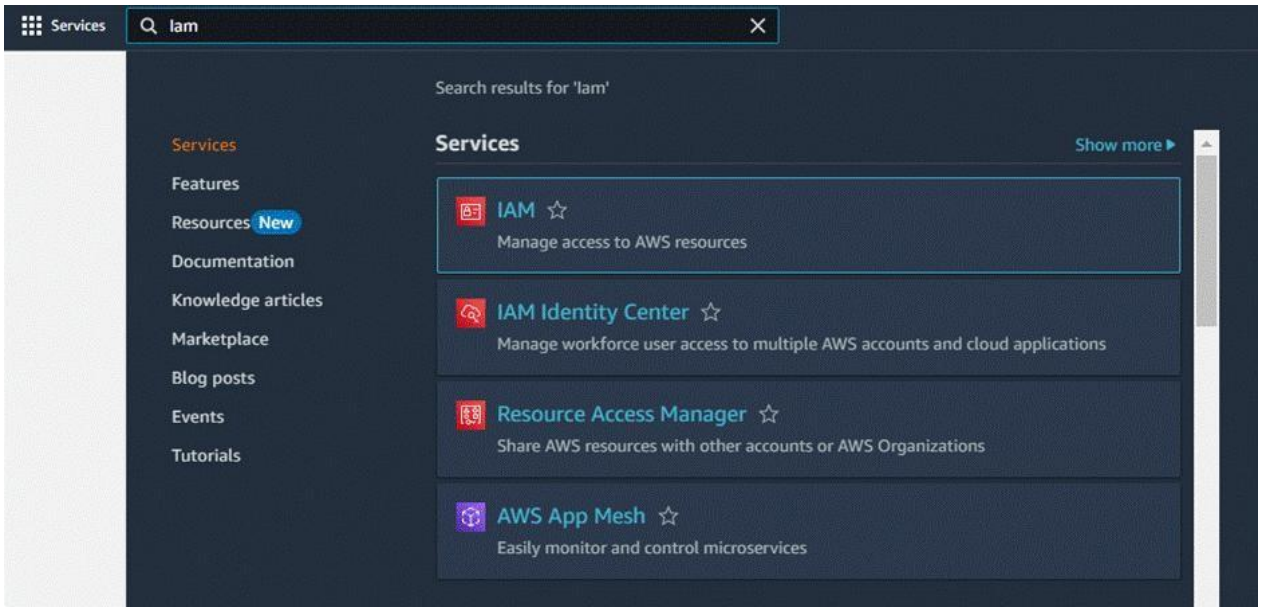
Milestone 5 : IAM Role Setup

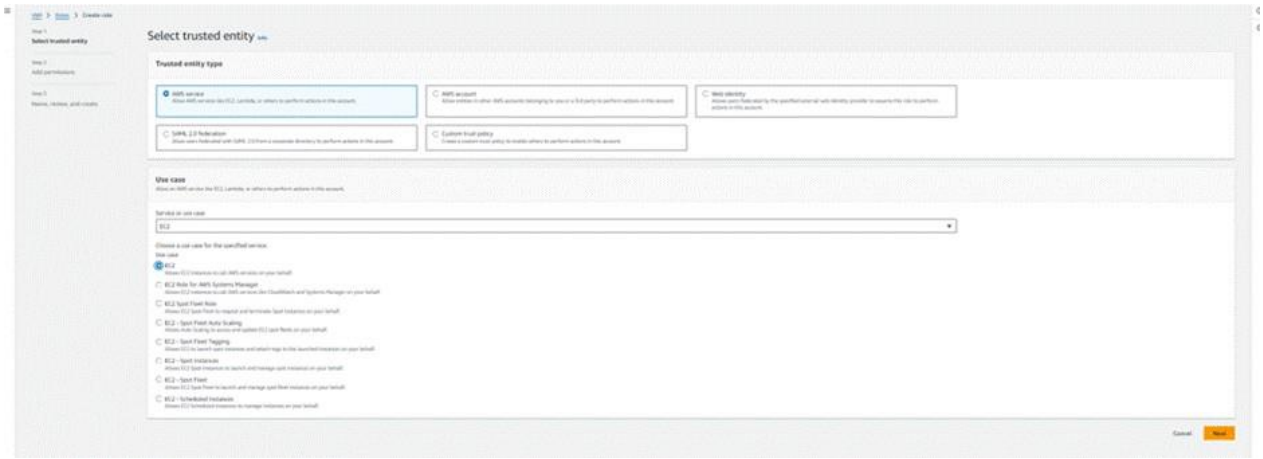
IAM Role Setup

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

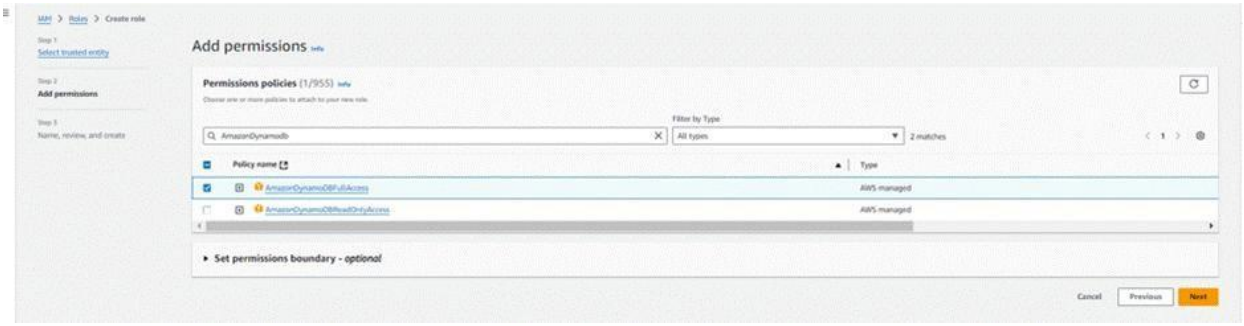
- In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.

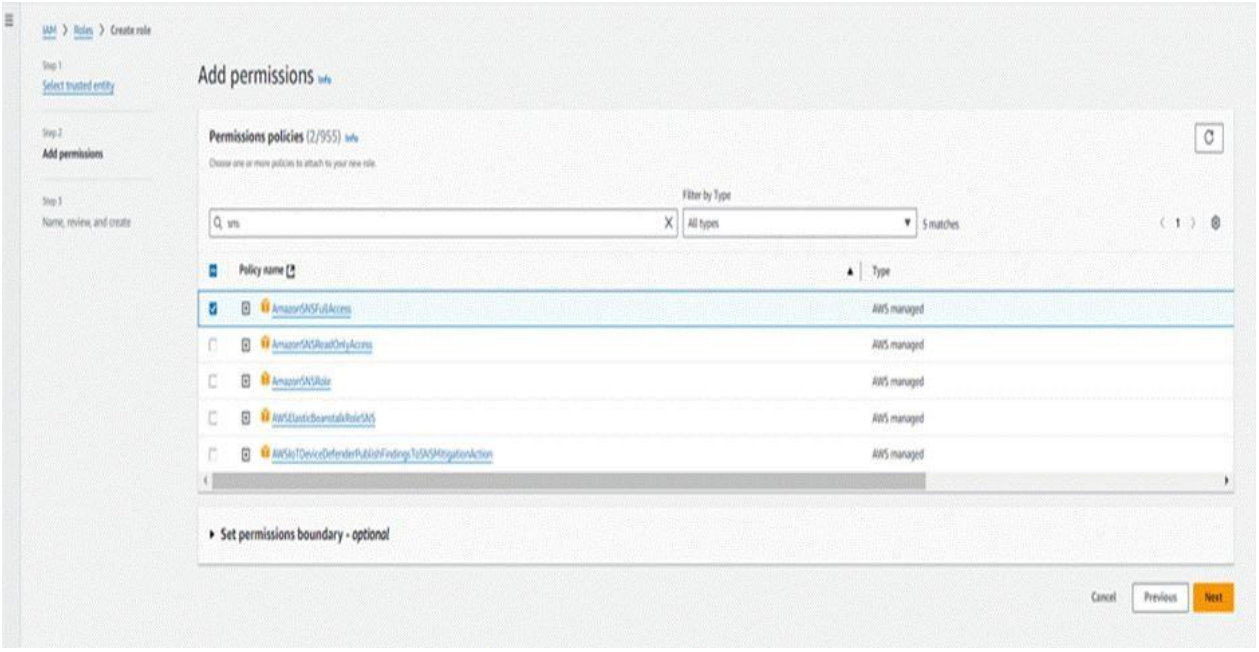




Attach Policies :

- 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.

moviemagic

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

Description
Add a short explanation for this role.

Allows EC2 instances to call AWS services on your behalf.

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

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sts:AssumeRole"
            ],
            "Principal": {
                "Service": [
                    "ec2.amazonaws.com"
                ]
            }
        }
    ]
}
```

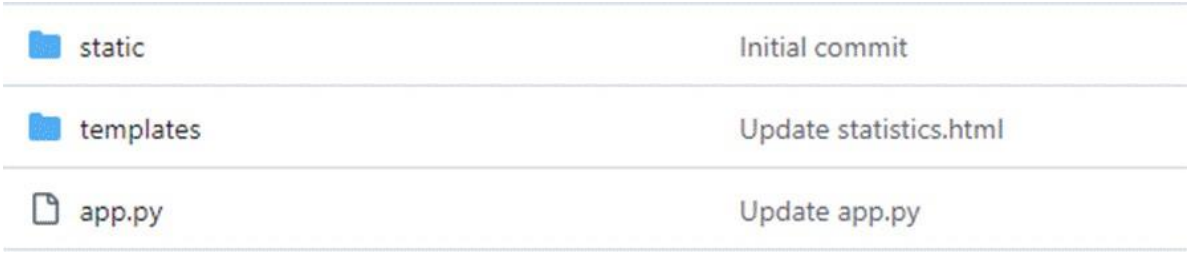
Milestone 6 : EC2 Instance setup

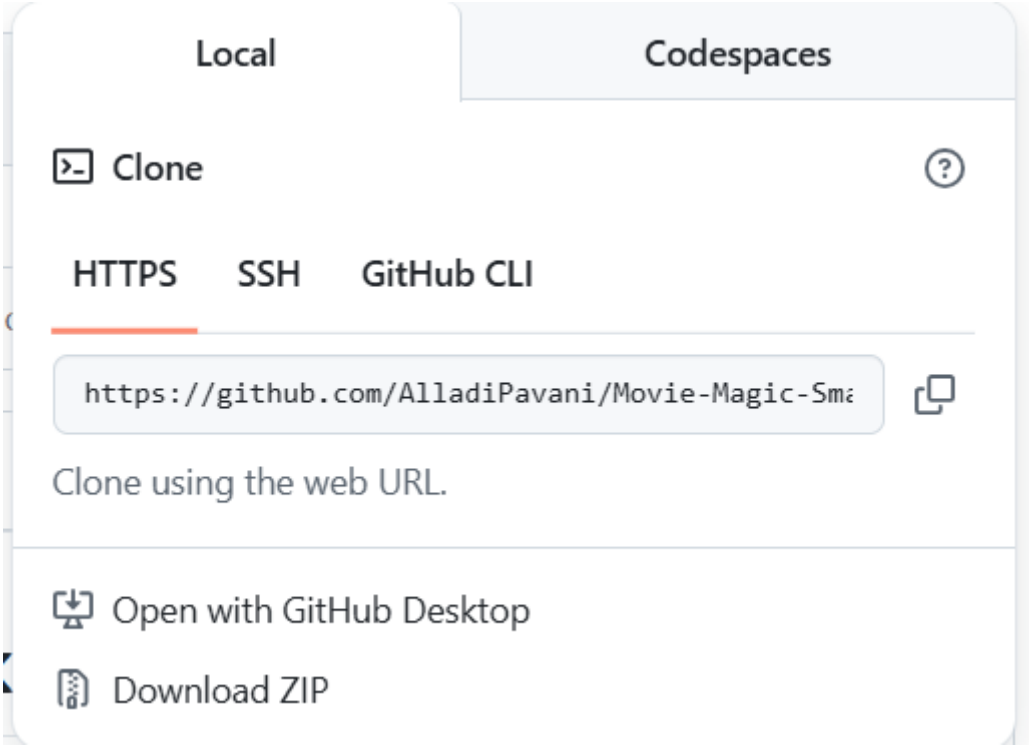
EC2 Instance setup

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

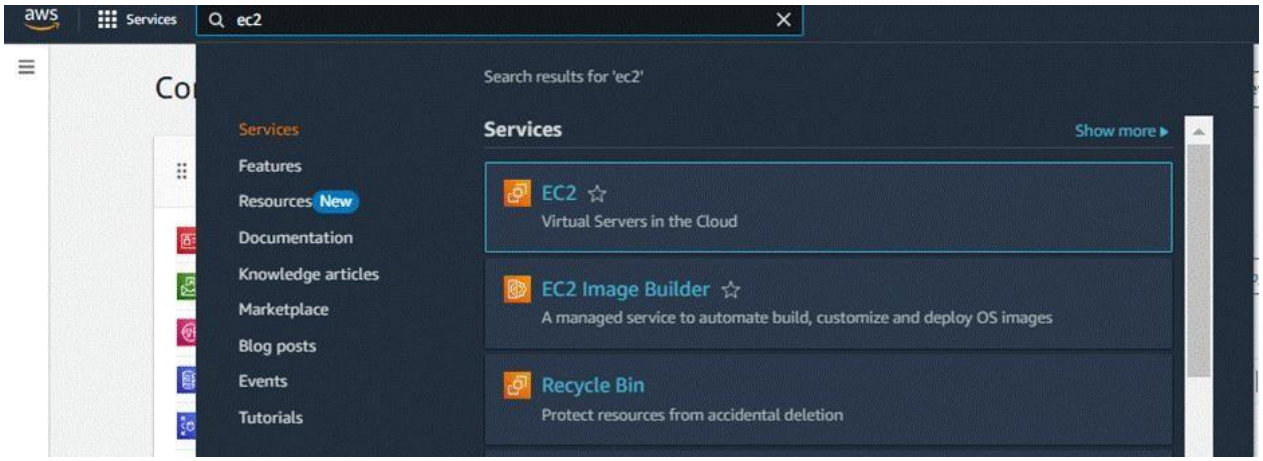
- Load your Flask app and Html files into GitHub repository.



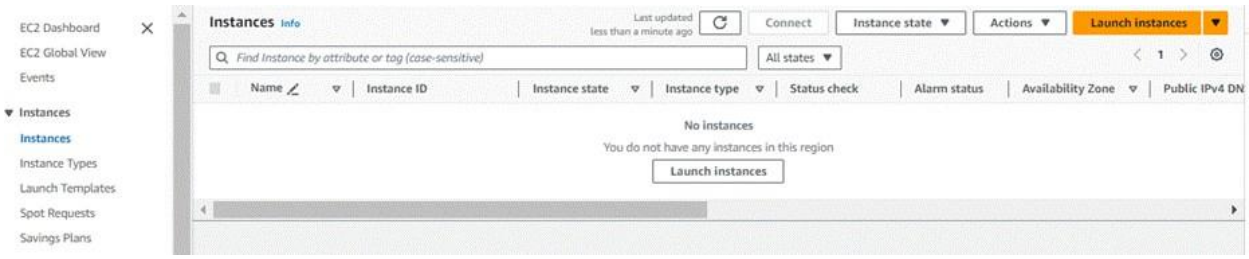


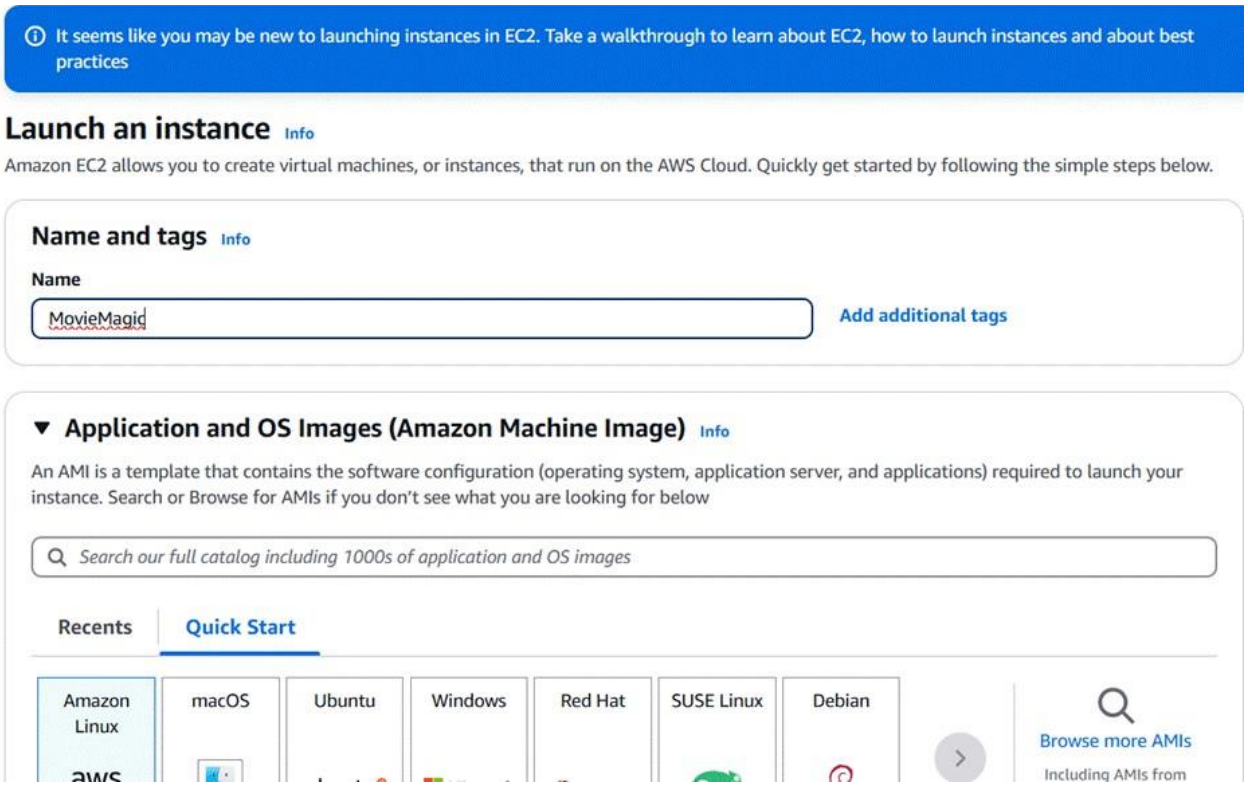
Launch an EC2 instance to host the Flask

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

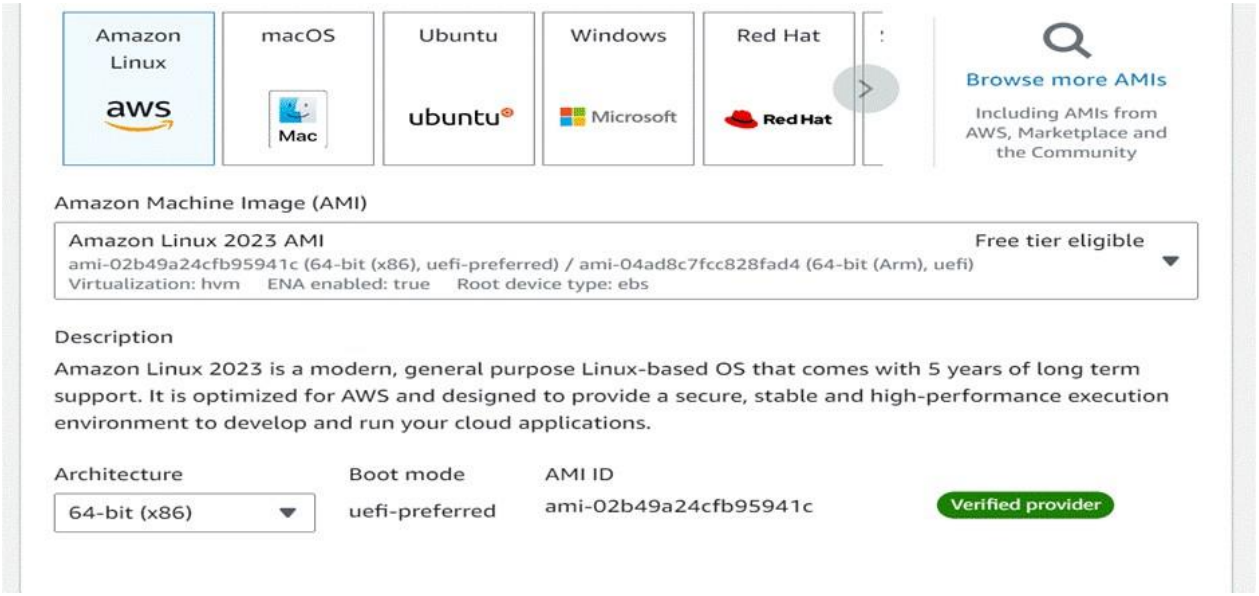


- Click on Launch instance to launch EC2 instance

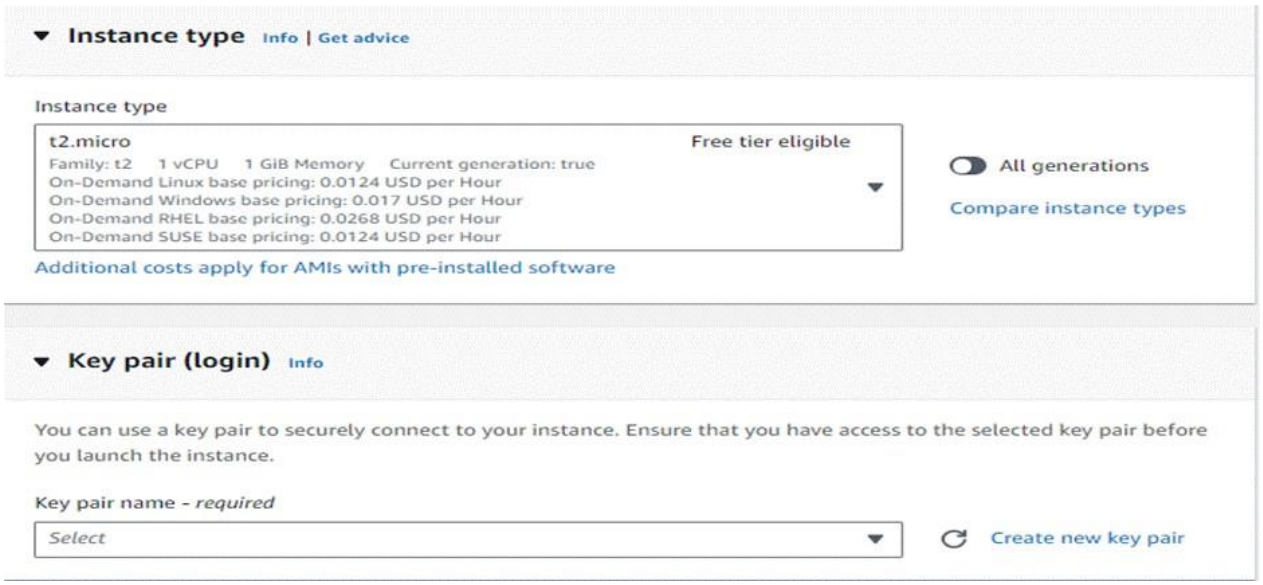


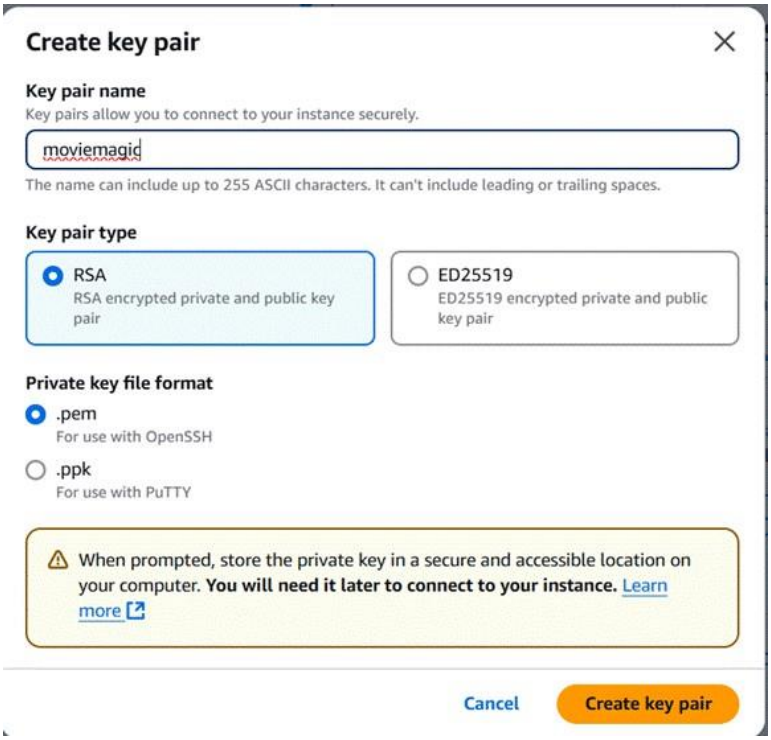


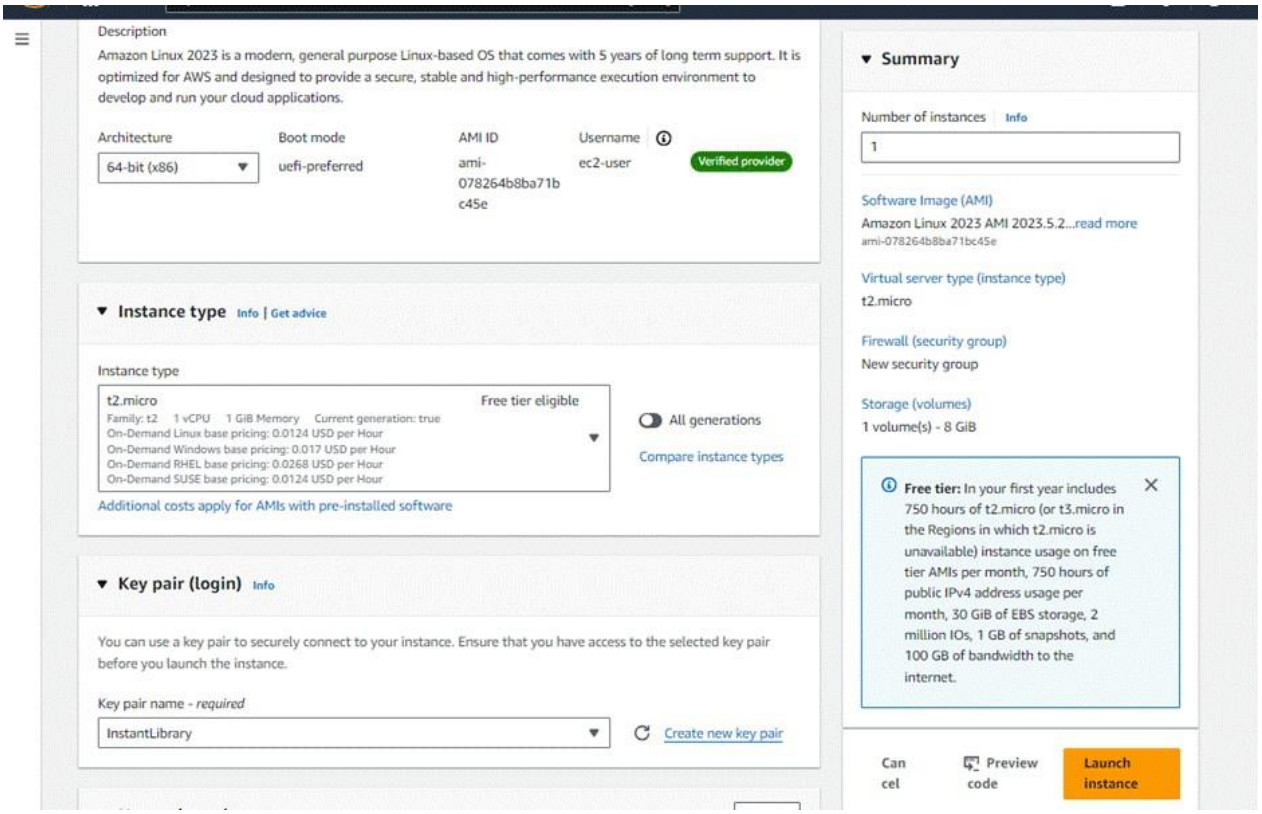
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).



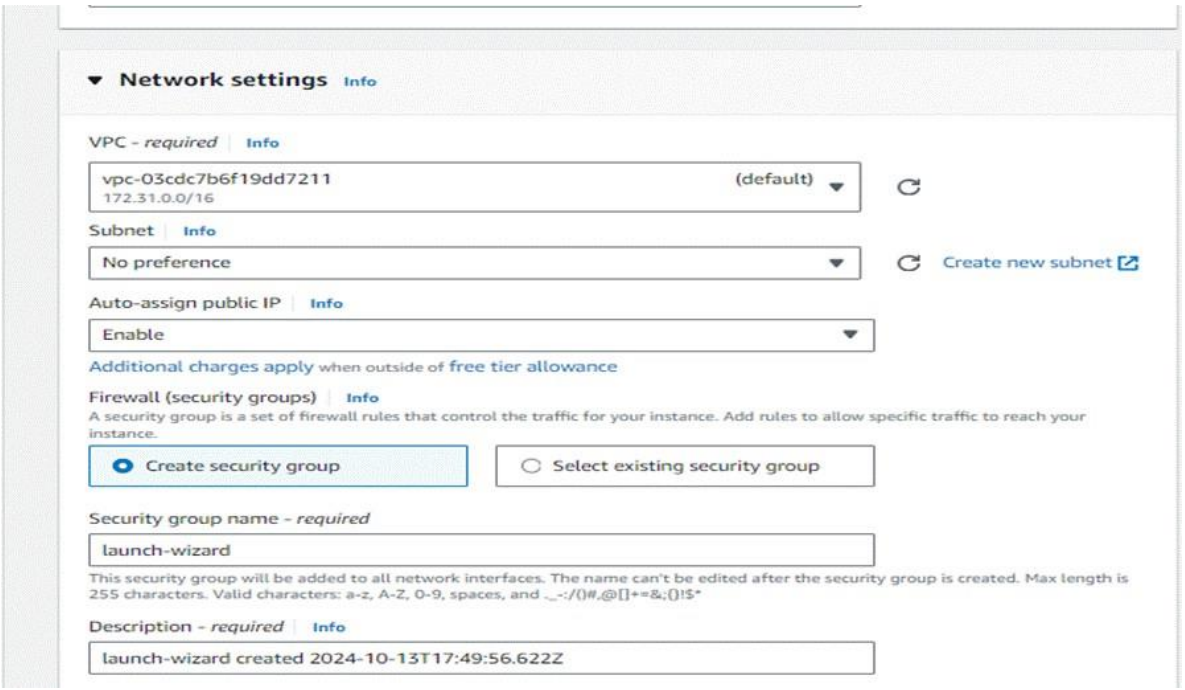
- Create and download the key pair for Server access.

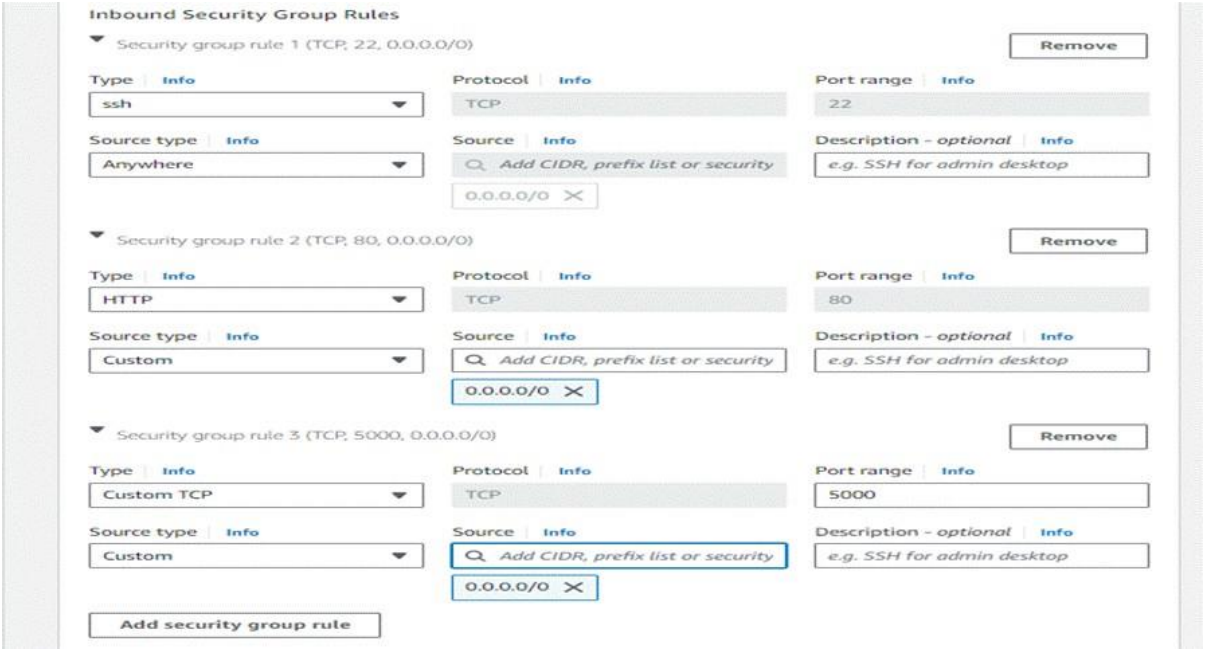

Instance type Info | Get advice
Instance type
t2.micro
Free tier eligible
Family: t2 1 vCPU 1 GiB Memory Current generation: true
On-Demand Linux base pricing: 0.0124 USD per Hour
On-Demand Windows base pricing: 0.017 USD per Hour
On-Demand RHEL base pricing: 0.0268 USD per Hour
On-Demand SUSE base pricing: 0.0124 USD per Hour
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

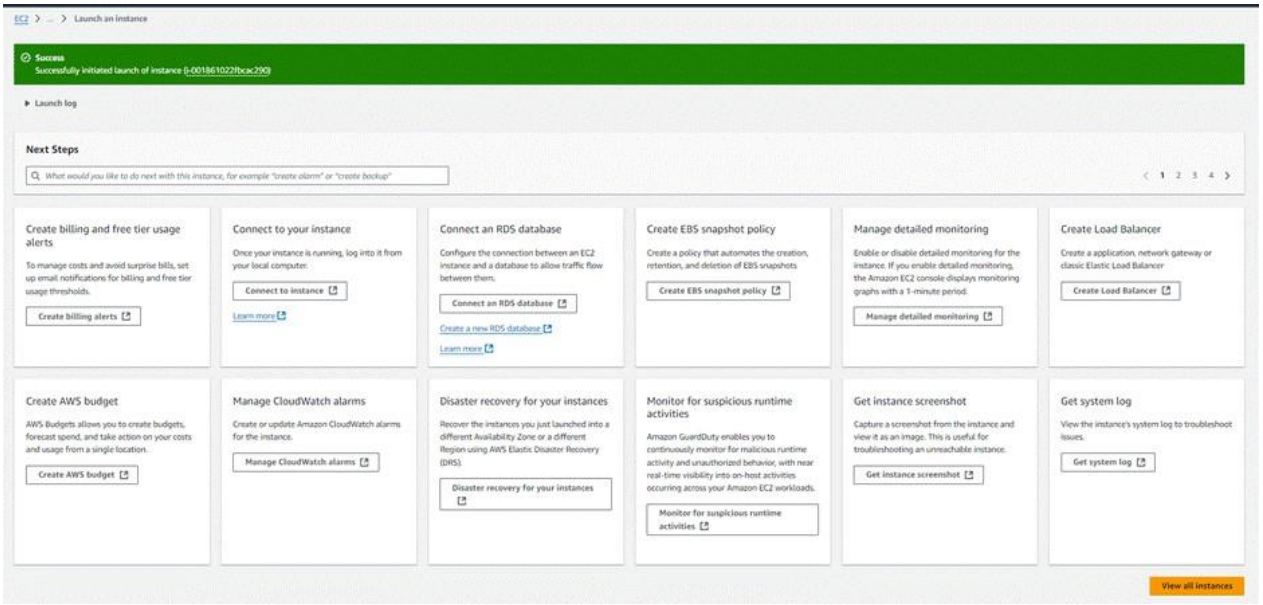

Create key pair
Key pair name
Key pairs allow you to connect to your instance securely.
moviemagic
The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.
Key pair type
RSA
RSA encrypted private and public key pair
ED25519
ED25519 encrypted private and public key 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



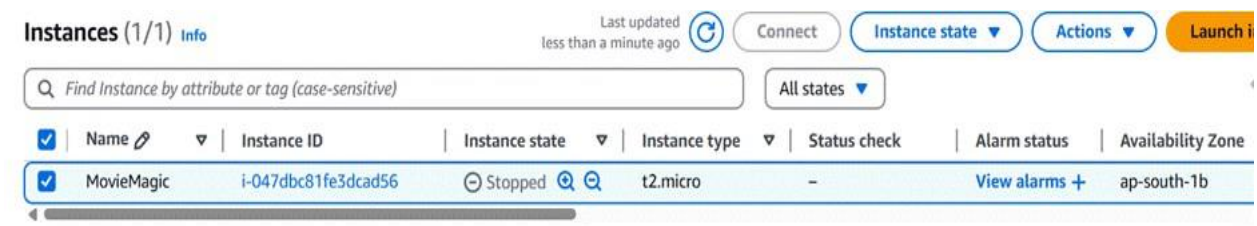
Configure security groups for HTTP, And SSH Access:

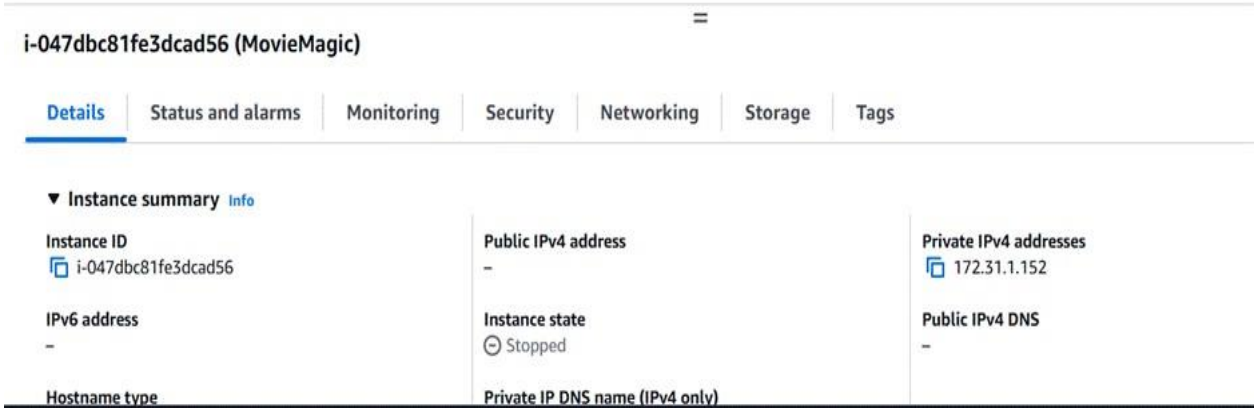




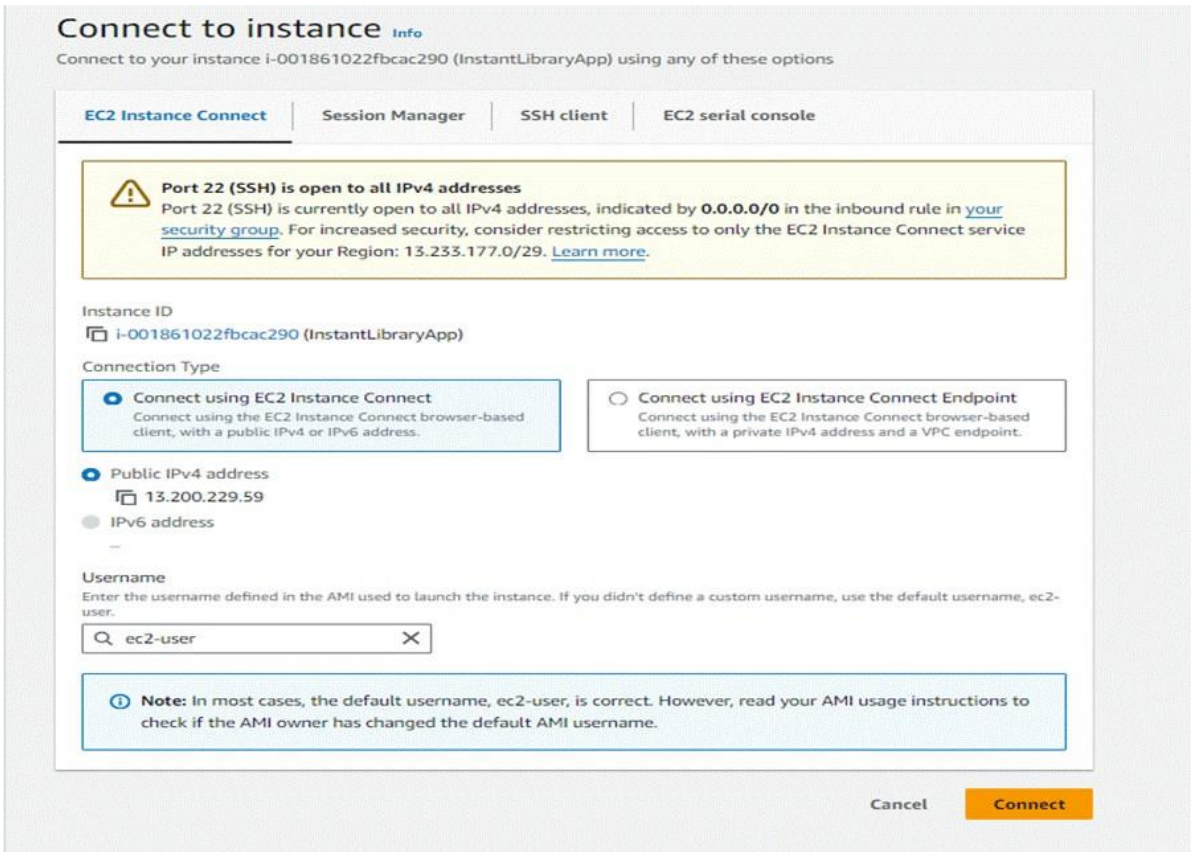


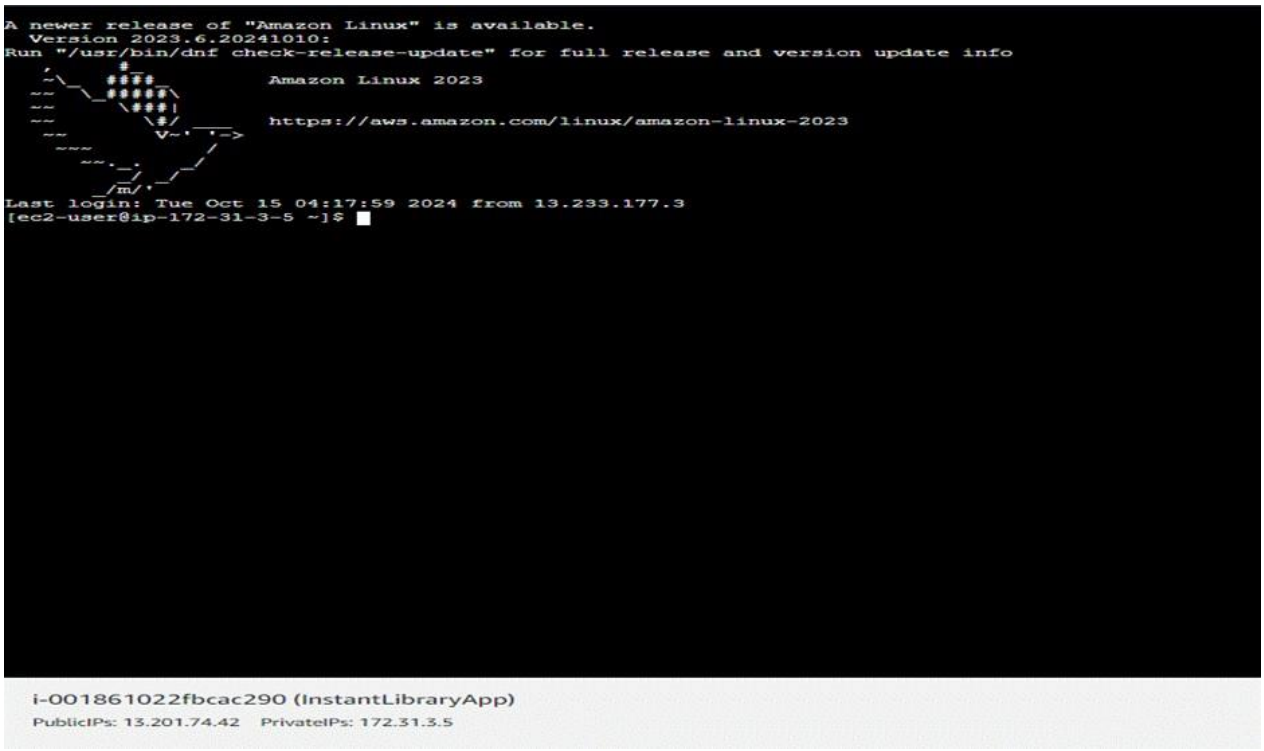
- To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected,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

Deployment Using EC2

Deployment on an EC2 instance involves launching a server, configuring security groups for public access, and uploading your application files. After setting up necessary dependencies and environment variables, start your application and ensure it's running on the correct port. Finally, bind your domain or use the public IP to make the application accessible online.

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

Clone Your Flask Project from GitHub

git clone: https://github.com/KagithaJayavamsee/Movie-Magic-Smart-Movie-Ticket-Booking-System

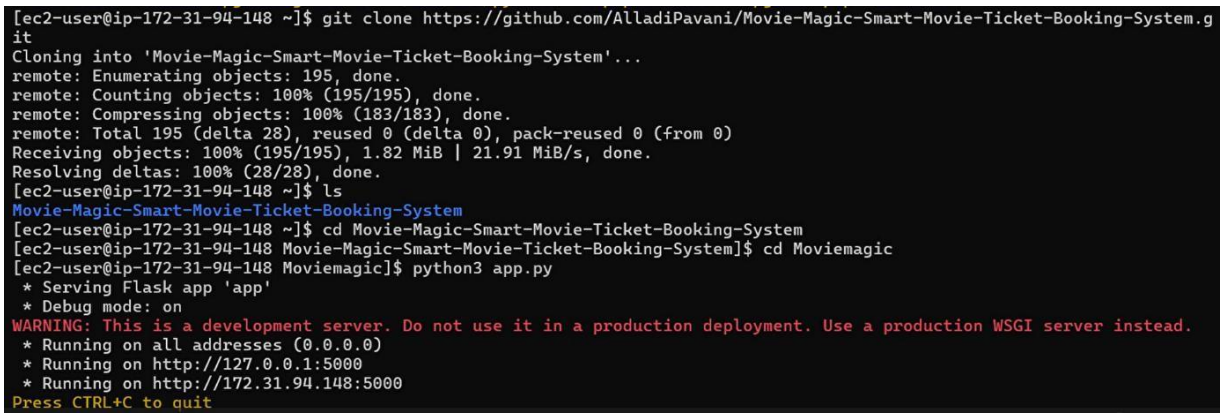
Clone your project repository from GitHub into the EC2 instance using Git.

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

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



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

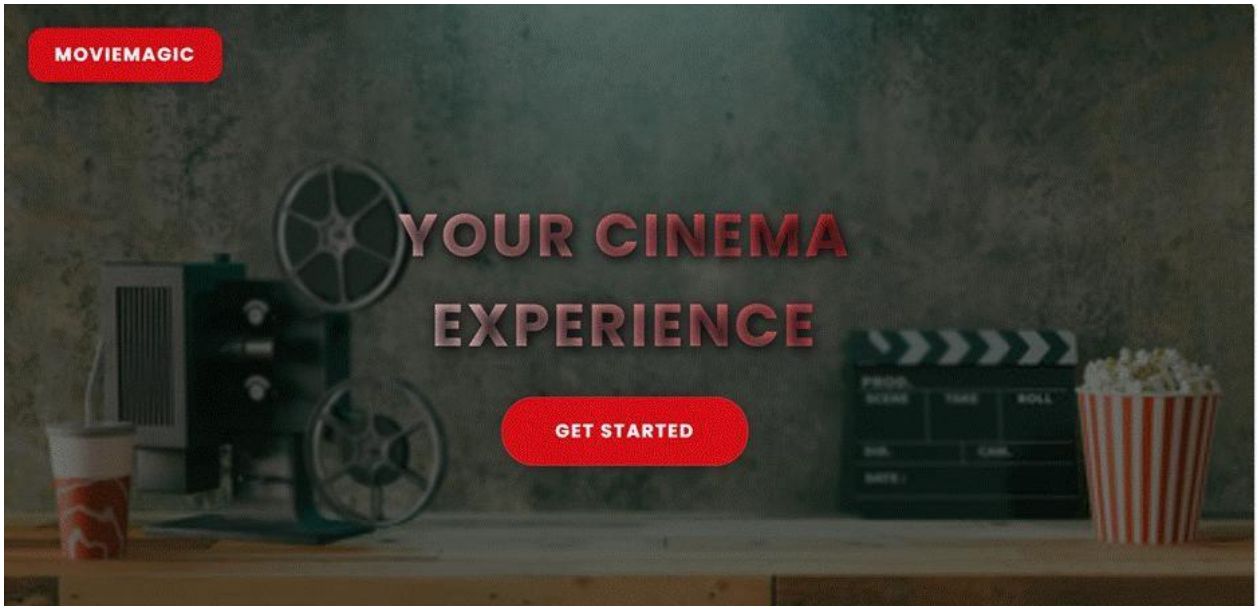
- Public IPs: http:// 18.215.178.87:5000

Milestone 8 : Testing and Deployment

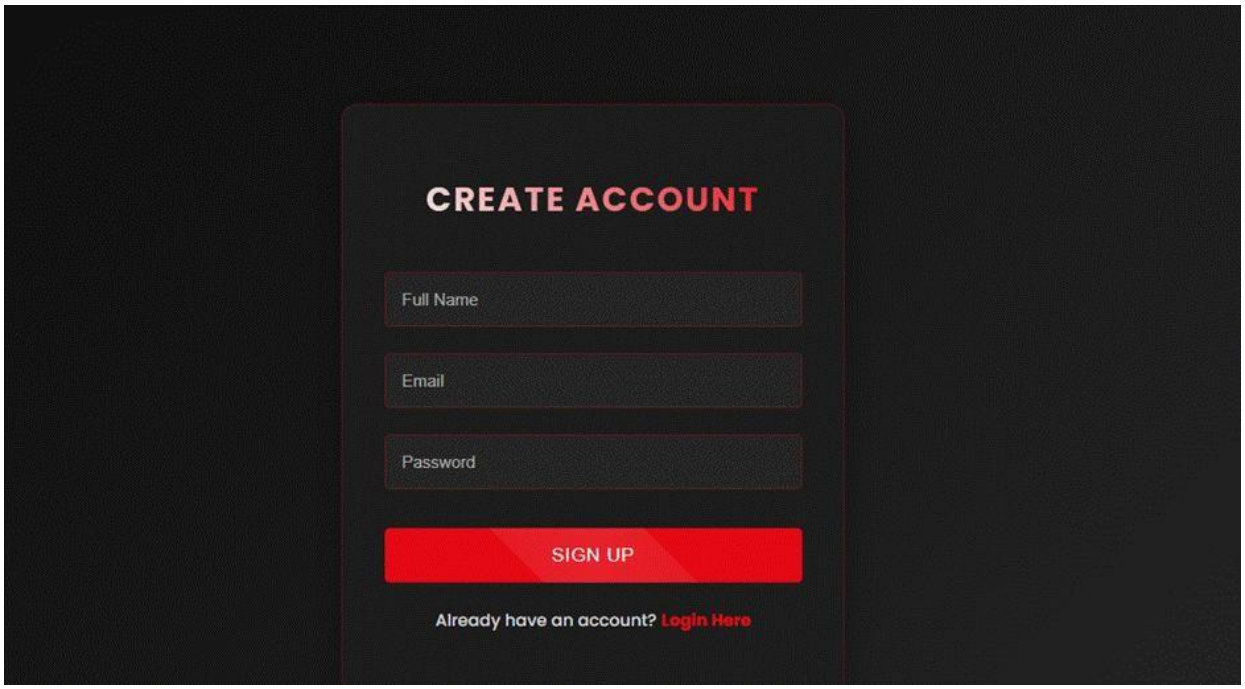
Testing and deployment involve verifying that your application works as expected before making it publicly accessible. Start by testing locally or on a staging environment to catch bugs and ensure functionality. Once tested, deploy the application to an EC2 instance, configure necessary services, and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

Functional testing to verify the Project

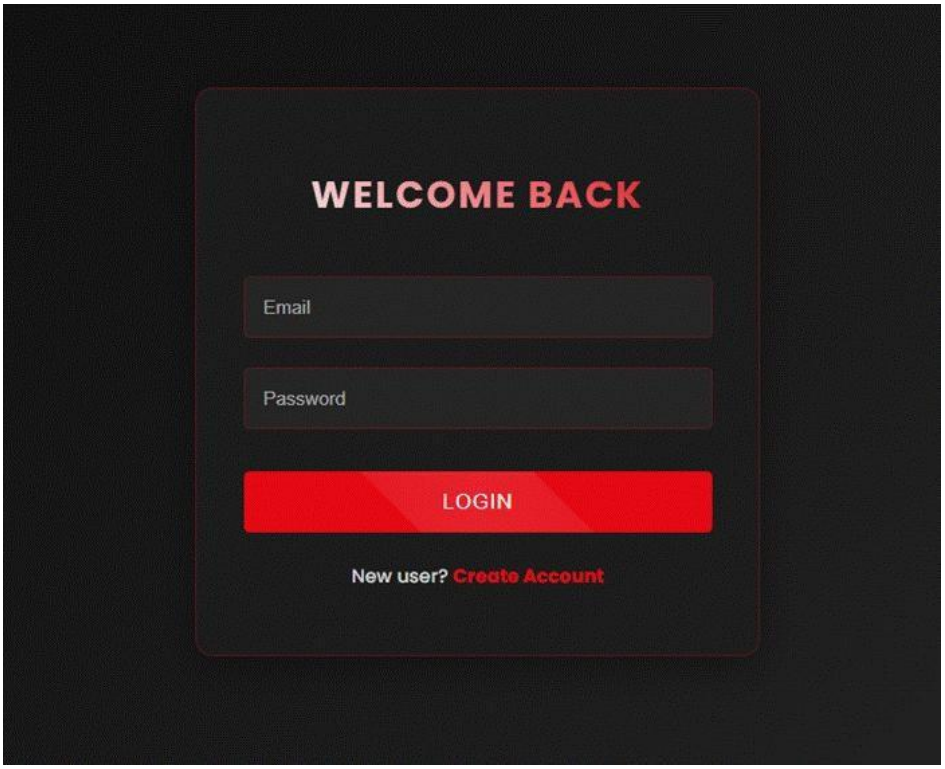
Index Page:



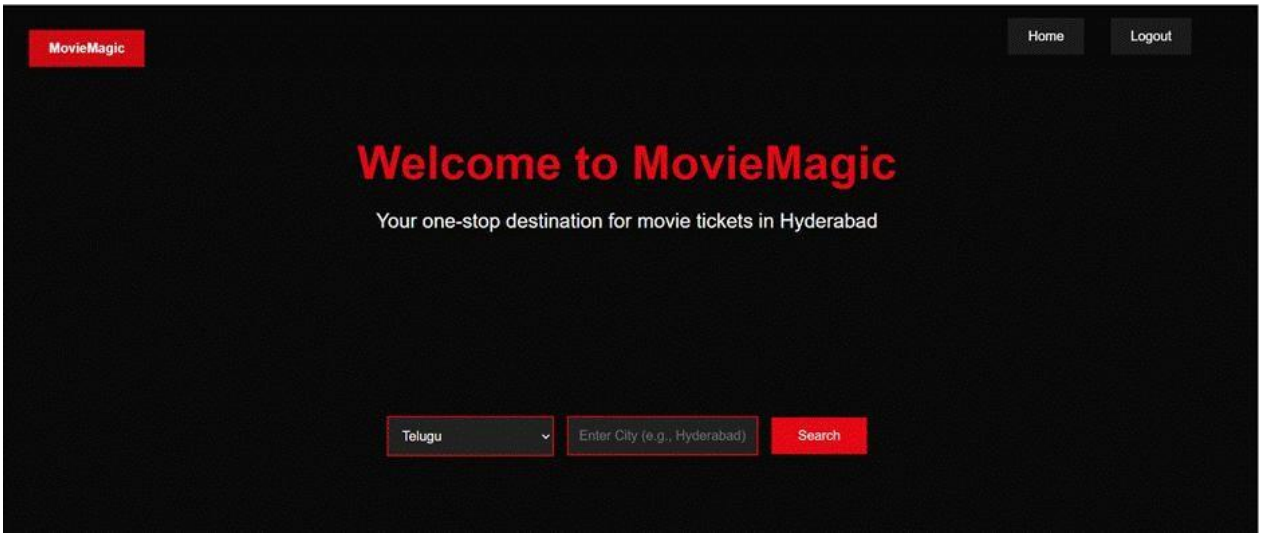
Register Page:

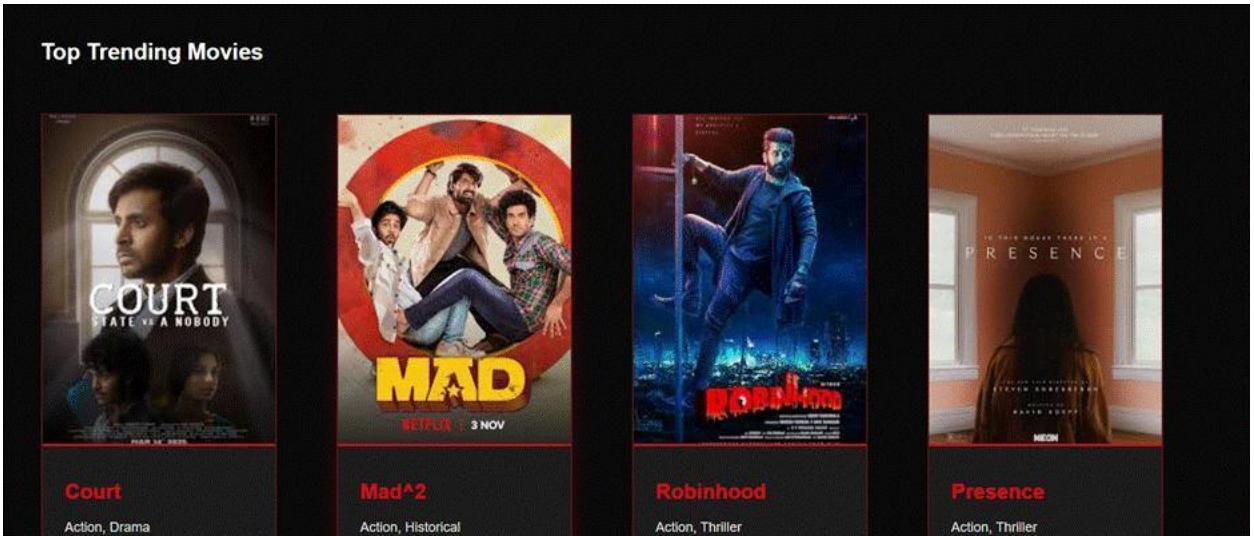


Login Page:



Home page:





Known Issuses :

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.