CHAPTER 1

INTRODUCTION

MovieGPT is a web application designed to provide users with a seamless platform for exploring movies and accessing detailed information such as genres, ratings, and summaries. With the ever-growing collection of movies available across streaming platforms and online databases, users often struggle to find reliable, concise, and easy-to-navigate resources to discover and learn about movies.

MovieGPT addresses this need by offering a user-friendly interface, powerful search functionality, and real-time access to movie data. Built with a focus on usability and scalability, the application leverages **React** for a responsive and interactive frontend, while **Firebase** provides backend support for authentication and database management.

The project not only aims to solve common challenges faced by movie enthusiasts but also serves as a platform to learn and implement advanced web development techniques. Its clean design, robust functionality, and focus on user experience make it a valuable tool for users seeking efficient movie exploration.

This report outlines the conceptualization, design, development, and testing processes involved in creating MovieGPT. It also discusses the challenges encountered during the development phase and how they were resolved, alongside potential areas for future improvement.

1.1 Objective

Primary Objectives

1. **Efficient Movie Search and Discovery:**
    - Provide users with a powerful and intuitive search feature to quickly find movies by title, genre, or other criteria.
    - Enable filtering and sorting options to enhance the movie discovery experience.

2. **Detailed Movie Information:**
    - Display relevant movie details such as title, genre, ratings, release year, and plot summary.
    - Present the information in a clean and user-friendly format to improve readability and accessibility.
3. **User Management and Personalization:**
    - Integrate secure user authentication using Firebase to allow personalized experiences.
    - Enable users to create accounts, log in, and manage their favorite movies or watchlists.

Secondary Objectives

4. **Responsive Design:**
    - Develop a mobile-friendly and responsive interface to ensure accessibility across devices of various screen sizes.
5. **Scalability:**
    - Use Firebase's real-time database or Firestore to efficiently store and retrieve data, ensuring scalability for future enhancements.
6. **Learning and Skill Development:**
    - Gain practical experience with **React** for front-end development and **Firebase** for back-end services.
    - Strengthen understanding of real-time data handling, authentication, and modern web development techniques.
7. **Optimized User Experience (UX):**
    - Ensure the application is lightweight, fast, and intuitive to provide a seamless experience for users.
    -

1.2 Background and Motivation

Movies are an integral part of entertainment and storytelling. With the rise of streaming platforms and the massive influx of new films, users often find it challenging to search for relevant movies or explore detailed information about their favorite ones. Many platforms provide either too much irrelevant information or lack user-friendly navigation.

The motivation for creating **MovieGPT** stems from the need for a simplified and intuitive platform that allows users to explore movies efficiently. By focusing on user needs, such as searching by title or genre and accessing relevant details like ratings, genres, and summaries, MovieGPT bridges the gap between extensive movie data and

user accessibility. This project also served as an opportunity to enhance skills in **React**, **Firebase**, and responsive web development, providing a practical learning experience.

Problem Statement

Users often face the following issues while searching for movie-related information:

1. Overwhelming and cluttered interfaces in existing movie databases or applications.
2. Difficulty finding relevant information due to poor search and filtering mechanisms.
3. Lack of personalization or features to save movies for future reference.

These challenges highlight the need for a streamlined and intuitive application. The **MovieGPT** project aims to resolve these issues by offering:

- A clean and interactive interface.
- Efficient search and filtering capabilities.
- Integration with a backend for storing and retrieving data in real-time.

1.3 Project Scope

The **MovieGPT** project focuses on developing a web application with the following objectives:

1. **Core Functionalities:**
    - A user-friendly interface for movie search and filtering.
    - Display of detailed information, such as title, genres, ratings, and overviews.
2. **Backend Integration:**
    - Firebase Authentication for secure login and user management.
    - Firebase Firestore/Realtime Database for storing and retrieving movie data.
3. **Additional Features:**
    - Allow users to create a watchlist or mark favorite movies.
    - Mobile-friendly, responsive design for accessibility across devices.
4. **Technological Learning:**
    - Gaining expertise in **React** for front-end development.
    - Implementing real-time data handling and authentication using **Firebase**.

CHAPTER 2

LITERATURE REVIEW

2.1 Research on Existing Movie Databases or Platforms

Several platforms cater to users looking for movie-related information. The most notable ones include:

1. **IMDb (Internet Movie Database):**
   - A comprehensive database with detailed information about movies, TV shows, and celebrities.
   - Features include user reviews, ratings, watchlists, and advanced search filters.
   - Known for its vast library and reliable data.
2. **TMDB (The Movie Database):**
   - A community-driven platform with open APIs for developers.
   - Features detailed movie information, user reviews, and ratings.
   - Strong API support for building third-party applications.
3. **Netflix, Prime Video, and Other Streaming Platforms:**
   - Offer movie details for their catalog but are limited to their content library.
   - Focus primarily on recommending movies based on viewing history.

2.2 How MovieGPT Improves Upon Existing Solution

1. **Simplified and User-Friendly Interface:**
   - Unlike IMDb and TMDB, which may overwhelm users with excessive details, MovieGPT focuses on providing only the most relevant information in a clean and concise format.
   - Intuitive navigation ensures a smoother user experience.

2. **Efficient Search and Filtering Mechanism:**
    - MovieGPT's search functionality is designed to be straightforward, allowing users to quickly find movies by title, genre, or ratings.
    - Filters are optimized for ease of use, reducing the complexity often encountered in platforms like IMDb.
3. **Personalized Features:**
    - Incorporates Firebase for user authentication, enabling personalized watchlists or favorite movie collections.
    - These features are not as accessible on platforms like Rotten Tomatoes.
4. **Focused Data Presentation:**
    - Displays essential movie details such as ratings, genres, and summaries without unnecessary clutter.
    - Improves upon TMDB's detailed but sometimes overwhelming approach.
5. **Open to Enhancements:**
    - While platforms like IMDb and Rotten Tomatoes are fixed in their functionalities, MovieGPT is designed to be modular, allowing for easy addition of new features such as user reviews, real-time updates, and recommendations.

CHAPTER 3

SYSTEM ANALYSIS AND SYSTEM DESIGN

3.1 System Analysis

3.1.1 **Functional Requirements**

Functional requirements define the core features and behaviors of the MovieGPT application:

1. **User Authentication:**
    - Users can register using their email and password.
    - Login functionality to provide personalized access.
    - Secure password reset feature.
2. **Movie Search:**
    - Users can search for movies by title or keywords.
    - Real-time search results with relevant suggestions.
3. **Filtering and Sorting Options:**
    - Filter movies based on genres, ratings, or release year.
    - Sort movies by popularity, ratings, or alphabetical order.
4. **Detailed Movie Information:**
    - Display essential details, including title, genre, release year, ratings, and plot summary.
5. **Watchlist Feature:**
    - Allow users to add movies to a personal watchlist or mark them as favorites.
    - Enable users to view and manage their watchlist.
6. **Mobile and Desktop Compatibility:**

- Responsive design to ensure compatibility across devices.

7. **Integration with Firebase Backend:**
   - Use Firebase Authentication for secure login and session management.
   - Store and retrieve user data, such as watchlists, in Firebase Firestore or Realtime Database.

3.1.2 Non-functional Requirements

Non-functional requirements ensure the quality and usability of the application:

1. **Performance:**
   - Fast response times for search and filtering operations.
   - Optimized database queries to reduce latency.
2. **Scalability:**
   - The application should handle an increasing number of users and data without significant performance degradation.
   - Firebase's real-time database ensures horizontal scaling capabilities.
3. **Reliability:**
   - Ensure the application is available 99% of the time through Firebase's cloud infrastructure.
   - Robust error handling to avoid crashes or unexpected behavior.
4. **Security:**
   - Use Firebase Authentication to safeguard user data.
   - Encrypt sensitive data, such as passwords, and follow best practices for user authentication.
5. **Usability:**
   - Simple and intuitive user interface.
   - Provide clear error messages and feedback for invalid inputs.
6. **Maintainability:**
   - Modular codebase to allow easy updates and feature additions.
   - Use modern frameworks (React) to simplify code maintenance.

3.2 System Design

3.2.1 Architecture Diagram

The **MovieGPT** application follows a client-server architecture with React as the frontend and Firebase as the backend. Below is an explanation of the architecture:

- **Frontend (Client):**
  - Developed using React for building a dynamic and responsive user interface.
  - Handles user interactions, such as search, filtering, and watchlist management.
- **Backend (Server):**
  - Firebase provides Authentication services for user login and account management.
  - Firestore or Realtime Database is used for storing and retrieving movie data and user-specific information.
- **External APIs (Optional for Future Enhancements):**
  - Integration with movie APIs like TMDB for updated and accurate movie data.
- **Diagram Representation:**

  [User] <-> [React Frontend] <-> [Firebase Authentication & Firestore] <-> [External Movie API]

3.2.2 Database Design and Schema (Firebase Firestore)

Firebase Firestore is a NoSQL cloud database that organizes data into collections and documents. Below is the schema design for the MovieGPT project:

1. **Users Collection:**
   - watchlist: Array of movie IDs added to the user's watchlist. **Purpose:** Store user-specific data.
   - **Fields:**
   - userId: Unique identifier for the user (Auto-generated).
   - email: User's email address.
   - **Sample Document:**

```
json
{
  "userId": "u12345",
  "email": "user@example.com",
  "watchlist": ["m001", "m002"]
}
```

2. **Movies Collection:**
   - **Purpose:** Store information about movies.
   - **Fields:**
     - movieId: Unique identifier for the movie.

- title: Movie title.
- genre: Array of genres (e.g., ["Action", "Drama"]).
- rating: Numeric rating (e.g., 8.5).
- releaseYear: Year of release.
- summary: A brief description of the movie.

- **Sample Document:**

3. **Watchlist Sub-Collection (Optional):**
   - A sub-collection under each user document to store watchlist details separately for better scalability.

3.2.3 Component Diagram (Frontend Structure)

The frontend is developed using React, with a modular component-based structure. Below is an overview of the main components:

1. **App Component:**
   - Root component that manages routing and state sharing across the application.
   - Routes to key pages like home, login, and watchlist.

2. **Header Component:**
   - Displays the navigation menu and search bar.
   - Allows users to access their watchlist or log in/out.
3. **Search Component:**
   - Handles movie search functionality.
   - Dynamically updates results based on user input.
4. **MovieCard Component:**
   - Displays individual movie details in a card format.
   - Includes options like "Add to Watchlist."
5. **MovieDetails Component:**
   - Shows detailed information about a selected movie.
   - Accessed when a user clicks on a movie card.
6. **Watchlist Component:**
   - Lists movies added to the user's watchlist.
   - Provides options to remove movies.
7. **Authentication Components:**
   - **Login Component:** Form for user login.

- **Register Component:** Form for user registration.

8. **Footer Component:**
    - Provides links to additional resources or contact information.

Component Diagram Representation:

App
├── Header
├── Footer
├── Search
│ ├── MovieCard
│ └── MovieDetails
├── Watchlist
├── Login
└── Register

This structure ensures modularity, making it easier to maintain and expand the application.

CHAPTER 4

IMPLEMENTATION OF MOVIEGPT

4.1 Frontend Development

The frontend of MovieGPT is developed using **React**, leveraging its component-based architecture to create a modular, reusable, and responsive user interface.

React Components

The application consists of the following main components:

1. **App Component:**
   - Acts as the root component.
   - Manages routing using libraries like **React Router DOM** for navigation between pages (e.g., Home, Watchlist, Login).
2. **Header Component:**
   - Contains the search bar and navigation links.
   - Dynamically displays user-related options such as login/logout or watchlist access.
3. **Search Component:**
   - Handles user input to search for movies.
   - Displays real-time results based on user queries by fetching data from Firestore.
4. **MovieCard Component:**
   - Renders individual movie details (e.g., title, genre, rating) in a card format.
   - Includes buttons for actions like "Add to Watchlist."
5. **MovieDetails Component:**
   - Displays detailed information when a movie card is clicked.
   - Includes additional options, such as adding or removing from the watchlist.
6. **Watchlist Component:**
   - Fetches and displays the user's saved movies from Firestore.

- Allows users to remove movies or navigate to movie details.

7. **Authentication Components:**
   - **Login Component:** A form for user login with validation for email and password.
   - **Register Component:** A form for new user registration.
8. **Footer Component:**
   - Provides additional links and general information about the application.

State Management

State management in MovieGPT is implemented using:

- **React Context API:**
  - Shares global states such as user authentication status and the current user's data across components.
  - Example: User's watchlist is accessed from multiple components like the Watchlist and MovieCard components.
- **Local Component State:**
  - Manages transient states such as form inputs and search queries.

4.2 Backend Integration

The backend leverages **Firebase** for authentication and database services.

Firebase Authentication

1. **User Registration:**
   - New users register using their email and password.
   - Firebase Authentication manages user credentials and securely stores them.
2. **Login and Logout:**
   - Handles user login using Firebase's signInWithEmailAndPassword method.
   - Session persistence ensures users stay logged in until they manually log out.
3. **Error Handling:**
   - Displays meaningful error messages for invalid login attempts, duplicate registrations, or password errors.

Firestore Database Integration

1. **Storing Movie Data:**
   - A pre-populated Firestore collection stores movie information (e.g., title, genre, rating).
   - Example query: Fetch all movies in the "Action" genre.
2. **User Watchlist Management:**

- Each user's watchlist is stored in a sub-collection under their document in the Users collection.
- Example structure:

scss

Copy code

Users (collection)

└── userId (document)

└── watchlist (sub-collection)

└── movieId (document)

3. **Real-time Updates:**
   - Any changes to the Firestore database, such as adding or removing a movie from the watchlist, are reflected in real time on the frontend.

4.3 Key Features and Their Implementation

1. **Movie Search:**
   - **Implementation:**
     - A search bar component captures user input.
     - Firestore queries fetch matching movie titles or keywords.
     - Results are displayed dynamically as the user types.
2. **User Authentication:**
   - **Implementation:**
     - Firebase handles user login, registration, and session persistence.
     - Protected routes ensure only authenticated users can access certain pages (e.g., Watchlist).
3. **Watchlist Feature:**
   - **Implementation:**
     - Movies can be added to or removed from the watchlist with a single click.
     - Watchlist data is stored in Firestore under the user's document.
     - The Watchlist component fetches and displays all saved movies.
4. **Responsive Design:**
   - **Implementation:**
     - CSS and libraries like **Bootstrap** ensure the application is mobile-friendly.
     - Media queries adjust the layout for different screen sizes.
5. **Error Handling:**

- **Implementation:**
  - Try-catch blocks handle API and Firebase errors.
  - Example: If a search query fails, the user is notified with a relevant error message.

CHAPTER 5

TECHNOLOGY USED AND SETUP

5.1 Frontend Technologies

1. **React.js:**
    - A JavaScript library for building dynamic and interactive user interfaces. React's component-based architecture allows for efficient rendering and reuse of UI elements.
    - **React Router DOM** is used for navigation between pages like home, login, and watchlist.
    - **Axios** or **Firebase SDK** is used for making API calls to fetch movie data from Firebase and other sources if needed.
    - **Redux** or **React Context API** for state management across the application, especially for managing user authentication states and the watchlist.
    - **CSS/SCSS** for styling and creating a responsive layout that works well across different screen sizes.
2. **Firebase SDK:**
    - Firebase provides a suite of tools for authentication (Firebase Authentication) and data storage (Firestore) in this project.
    - The **Firebase Firestore** database is used to store user information and movie data.

5.2 Backend Technologies

Firebase Authentication:

- Manages user registration, login, and session persistence without the need for setting up a traditional backend server.
- Supports email/password-based authentication and social logins (optional).

- **Firebase Firestore Database:**

- A NoSQL cloud database that stores movie data and user information. Data is structured in collections and documents, allowing for easy retrieval and real-time updates.

- **Optional API Integration (for future enhancements):**
  - **TMDB (The Movie Database) API** or other public movie APIs can be used to fetch the latest movie details, ratings, and reviews.

5.3 Development Tools

- **Visual Studio Code (VSCode):**
  - A lightweight and powerful code editor with support for React.js, JSX, and JavaScript. Extensions like Prettier and ESLint can be used to maintain clean and consistent code.
- **Node.js & npm (Node Package Manager):**
  - Node.js is used for running JavaScript on the server side (for local development). npm is used to manage libraries like React, Firebase SDK, and other dependencies.
- **Git and GitHub:**
  - Git for version control and GitHub for hosting the project repository and collaboration.
- **Firebase Console:**
  - Used for setting up and managing Firebase services like Authentication, Firestore, and hosting.

5.4 HARDWARE REQUIREMENTS

The hardware requirements for developing and running MovieGPT are minimal and can be managed on most modern machines. Here are the key requirements:

1. **Development Machine (for coding and testing):**
   - **Processor:** Intel Core i3 or equivalent (preferably i5 or higher for faster development).
   - **RAM:** Minimum 4GB RAM (8GB or higher recommended for smooth development experience).
   - **Storage:** At least 10GB of free disk space to accommodate the development environment, dependencies, and project files.
   - **Graphics:** Integrated graphics are sufficient for frontend development unless advanced graphical features are being used.
   - **Operating System:** Windows 10/11, macOS, or Linux.
2. **Production (User-facing)**

Requirement	Minimum Specification	Recommended Specification
Processor (CPU)	Intel Core i3 or equivalent	Intel Core i5 or higher
RAM	4GB	8GB or higher
Storage	10GB free disk space	20GB or more (especially if you plan to store large files locally)
Graphics	Integrated graphics (sufficient for frontend development)	Dedicated graphics (optional, if advanced graphical features are needed)
Operating System	Windows 10/11, macOS, or Linux	Any OS (Windows, macOS, Linux) with up-to-date versions

- Since MovieGPT is a web application, users can access it from any device with a modern browser (e.g., Chrome, Firefox, Safari, Edge). There are no specific hardware requirements for end-user.

This specifies the storage requirement for the PC. It should have a hard disk with a capacity of 5 gigabytes (GB) or more. This is where you store your operating system, software applications, and data. A PC with a 5 GB or larger hard disk provides ample storage for the operating system, software applications, and user data. This ensures smooth performance, accommodates growing storage needs, and allows for data backups and future expansion.

5.5 SOFTWARE REQUIREMENTS

1. **Operating System:**
   - **Windows 10/11**, **macOS** (10.12 or higher), or **Linux** (Ubuntu 20.04 or higher).
2. **Development Tools:**
   - **Visual Studio Code (VSCode):** A popular code editor for JavaScript and React development.
   - **Node.js (v14 or higher):** A runtime for executing JavaScript code on the server-side during development.
   - **npm (v6 or higher):** Package manager for handling dependencies in the React project.
3. **Libraries and Frameworks:**
   - **React.js (v18 or higher):** A JavaScript library for building user interfaces.
   - **React Router DOM:** For handling navigation between different views (pages).

- **Firebase SDK (v9 or higher):** For Firebase Authentication, Firestore, and other Firebase features.
- **Axios (optional):** For making API calls to fetch movie data from external APIs (like TMDB).
- **CSS/SCSS (optional):** For styling and building responsive layouts.

4. **Version Control:**
   - **Git:** For managing source code versions and collaborating with team members.
   - **GitHub or GitLab (optional):** Cloud-based hosting platforms for version control repositories.
5. **Firebase Services:**
   - **Firebase Console:** For managing Firebase projects, setting up Authentication, Firestore, and Hosting.
   - **Firebase Firestore:** NoSQL database for storing user and movie data.
   - **Firebase Authentication:** For handling user sign-up, login, and session management.

Table 5.1 Software Requirements

S. No.	Description	Type
1	Operating System	• Windows 10 • or 11
2	Front End	• Vite, React • JS, JS, • Tailwind • CSS
3	Back End	• Node JS
4	IDE	• VSCode
5	IDE	• VS Code
6	Browser	• Chrome, • Firefox, • Edge

Operating System

The specified operating system requirement for the project development environment is either Windows 10 or 11, with the option to use a newer version if available. This ensures compatibility with the latest software development tools, libraries, and frameworks required for the project. Additionally, it provides a consistent and stable platform for developers to create, test, and deploy the project's software components. By standardizing the operating system environment, it facilitates collaboration among team members and simplifies software configuration management, version control, and troubleshooting processes throughout the project lifecycle.

Front End

The frontend of a project is what users interact with directly, including the interface, design, and user experience. The goal is to create an intuitive, visually appealing, and responsive interface that guides users seamlessly through the application.

React JS

React.js is a JavaScript library for building user interfaces. It simplifies UI development by breaking it down into reusable components and managing updates efficiently with a virtual DOM. Developers use JSX to write UI components, enabling a declarative approach to defining UIs based on application state

Tailwind CSS

Tailwind CSS is a utility-first CSS framework that streamlines frontend development by providing a set of pre-defined utility classes for styling HTML elements. It prioritizes simplicity, responsiveness, and customization, making it easy for developers to create modern and responsive user interfaces efficiently.

Vite

Vite is a fast build tool for modern web development, specifically designed to optimize the development experience for frontend projects. It leverages native ES modules in modern browsers to deliver instant server startup and rapid hot module replacement (HMR). With Vite, developers can build and serve frontend applications quickly, enhancing productivity and facilitating a smooth development workflow.

Back End

The backend of a project comprises server-side code, databases, and APIs that handle data processing, business logic, authentication, security, and communication with clients. It powers the functionality of the application, manages data storage, and ensures that users can interact with the system securely and efficiently.

MongoDB

MongoDB is a NoSQL, document-oriented database designed for managing large volumes of data across distributed systems. It features a flexible, schema-less data model that allows data to be stored in JSON-like documents with dynamic schemas. This flexibility enables developers to store complex data structures and adapt quickly to changing requirements without needing to define a rigid schema upfront. MongoDB supports horizontal scaling through sharding, distributing data across multiple servers, and is optimized for high- performance read and write operations, making it suitable for high-throughput applications. Its rich query language supports ad hoc queries, indexing, and real-time aggregation, while replication ensures high availability and redundancy through replica sets.

IDE

An Integrated Development Environment (IDE) is a software tool that combines various features to facilitate programming tasks. It typically includes a code editor, compiler/interpreter, debugger, build automation tools, version control integration, and project management capabilities. IDEs increase developer productivity by providing a centralized environment for coding, debugging, and managing projects, ultimately leading to more efficient software development.

Visual Studio Code

Visual Studio Code (VS Code) is a highly popular and versatile integrated development environment (IDE) developed by Microsoft. It's favoured by developers across various platforms and programming languages due to its extensive features and flexibility. Visual Studio Code (VS Code) is the preferred integrated development environment for coding.

Browser

Accessing a project via a web browser allows users to interact with web-based applications or websites. It provides accessibility, cross-platform compatibility, userfriendly interfaces, security features, scalability, and easy updates, making it a crucial aspect of modern computing. Mozilla Firefox, Google Chrome, Microsoft Edge, any of the browsers can be used to access the software.

5.6 Optional Setup (for Deployment and Hosting):

1. **Firebase Hosting:**
   - MovieGPT can be deployed on Firebase Hosting for free (with limitations on traffic and storage). Firebase Hosting allows you to easily deploy your React application to a global content delivery network (CDN), ensuring fast access from any region.
2. **Cloudflare (optional):**
   - Cloudflare can be used for enhanced security, CDN services, and DNS management if needed.

5.5 Setup Steps for Development:

1. **Clone the Repository:**

   git clone <repository_url>

   cd <project_folder>
2. **Install Node.js dependencies:**

   Install the necessary libraries via npm.

   npm install
3. **Set up Firebase project:**
   - Go to the Firebase Console and create a new project.
   - Set up Firebase Authentication and Firestore.
   - Get the Firebase SDK configuration and add it to your React project.
4. **Run the Development Server:**

   npm start
5. **Access the Application:**

CHAPTER 6

TESTING

6.1 Unit Testing :

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.

In object-oriented programming, a unit is often an entire interface, such as a class, but it could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing. Ideally, each test case is independent from the others. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist in testing a module in isolation. Unit tests are typically written and run by software developers to ensure that codemeets its design and behaves as intended. Benefits of Unit Testing.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it offers several benefits.

Find Problems Early:

Unit testing finds problems early in the development cycle. In test-driven development (TDD), which is frequently used in both extreme programming and scrum, unit tests are created before the code itself is written. When the tests pass, that code is

considered complete. The same unit tests are run against that function frequently as the larger code base is developed either as the code is changed or via an automated process with the build.

The unit tests then allow the location of the fault or failure to be easily traced. Since the unit tests alert the development team of the problem before handing the code off to testers or clients, it is still early in the development process.

Facilitates Change:

Unit testing allows the programmer to refactor code or upgrade system libraries later, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified. Unit tests detect changes that may break a design contract.

Documentation:

Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit, and how to use it, can look at the unit tests to gain a basic understanding of the unit's interface (API). Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviours that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in the development unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

6.2 Integration Testing

Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

The purpose of integration testing is to verify the functional, performance, and reliability requirements placed on major design items. These "design items", i.e., assemblages (or groups of units), are exercised through their interfaces using black-box

testing, with success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface.

Test cases are constructed to test whether all the components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e., unit testing. Software integration testing is performed according to the software development life cycle (SDLC) after module and functional tests. Some different types of integration testing are big-bang, top- down, and bottom-up, mixed (sandwich) and risky- hardest. Other Integration Patterns are collaboration integration, backbone integration, layer integration, client-server integration, distributed services integration and high-frequency integration.

This method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing. A type of big-bang integration testing is called "usage model testing" which can be used in both software and hardware integration testing. The basis behind this type of integration testing is to run user-like workloads in integrated user-like environments. In doing the testing in this manner, the environment is proofed, while the individual components are proofed indirectly through their use. Usage Model testing takes an optimistic approach to testing because it expects to have few problems with the individual components.

The strategy relies heavily on the component developers to do the isolated unit testing for their product. The goal of the strategy is to avoid redoing the testing done by the developers, and instead flesh-out problems caused by the interaction of the components in the environment.

To be more efficient and accurate, care must be used in defining the user-like workloads for creating realistic scenarios in exercising the environment. This gives confidence that the integrated environment will work as expected for the target customers.

Top-Down and Bottom-Up

Bottom-up testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher-level components.

The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower-level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready.

This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage. Top-down testing is an approach to integrated testing where the top integrated modules are tested, and the branch of the module is tested step by step until the end of the related module. Sandwich testing is an approach to combine top-down testing with bottom-up testing.

Black-Box Testing

Black box testing is a technique used to test the functionality of a software application without having knowledge of its internal structure or implementation details. It focuses on the inputs and outputs of the system and verifies if the expected outputs match the desired results. Here are some examples of black box testing techniques that can be applied to the KIET Event Management App:

6.3 Equivalence Partitioning:

- Identify different categories of inputs for the app, such as valid and invalid inputs, and divide them into equivalence classes.
- Test representative values from each equivalence class to ensure the app behaves consistently within each class.

Boundary Value Analysis:

Identify the boundaries or limits for inputs in the app, such as minimum and maximum values, and test values at those boundaries.

Test values just above and below the boundaries to verify the app's behaviour at critical points.

Decision Table Testing:

Identify the different conditions and rules that govern the behaviour of the app.

Create a decision table with combinations of conditions and corresponding expected results.

Test different combinations of conditions to validate the app's decision-making process.

State Transition Testing:

Identify the different states that the app can transition between.

Define the valid and invalid transitions between states.

Test different sequences of state transitions to verify the app's behaviour.

Error Guessing:

Use experience and intuition to guess potential errors or issues in the app.

Create test cases based on those guesses to verify if the app handles the errors correctly.

6.4 Compatibility Testing:

Test the app on different platforms, browsers, or devices to ensure compatibility. Verify that the app functions correctly and displays appropriately across different environments.

Usability Testing:

Evaluate the app's user interface and interactions from the perspective of an end-user. Test common user scenarios and assess the app's ease of use, intuitiveness, and overall user experience.

Security Testing:

Test the app for potential security vulnerabilities or weaknesses.

Verify if the app handles user authentication, data encryption, and access control appropriately.

Performance Testing:

Test the app's performance under different load conditions, such as a high number of concurrent users or large data sets.

Verify if the app responds within acceptable time limits and performs efficiently.

During black box testing, test cases are designed based on the app's specifications, requirements, and user expectations. The focus is on validating the functionality, user

interactions, and expected outputs without considering the internal implementation details of the app.

White-Box Testing

White box testing, also known as structural testing or glass box testing, is a software testing technique that examines the internal structure and implementation details of the application. It aims to ensure that the code functions as intended and covers all possible execution paths. Here are some examples of white box testing techniques that can be applied to the KIET Event Management App:

6.5 Unit Testing:

Test individual units or components of the app, such as functions or methods, to verify their correctness.

Use techniques like code coverage analysis (e.g., statement coverage, branch coverage) to ensure that all code paths are exercised.

Integration Testing:

Test the interaction between different components or modules of the app to ensure they work together seamlessly. Verify the flow of data and control between the modules and check for any integration issues or errors.

6.6 Path Testing:

Identify and test different paths or execution flows through the app, including both positive and negative scenarios.

Execute test cases that cover all possible paths within the code to ensure complete coverage. **Decision Coverage:**

Ensure that every decision point in the code (e.g., if statements, switch cases) is tested for both true and false conditions.

Validate that the app makes the correct decisions based on the specified conditions.

Code Review:

Analyse the code and its structure to identify any potential issues or vulnerabilities.

Review the adherence to coding standards, best practices, and potential optimizations.

Performance Testing:

Assess the app's performance from a code perspective, such as identifying any bottlenecks or inefficient algorithms.
Measure the execution time of critical code sections and evaluate resource usage.

6.7 Security Testing:

Review the code for potential security vulnerabilities, such as SQL injection, cross-site scripting (XSS), or authentication weaknesses.
Verify the implementation of secure coding practices, data encryption, and access control mechanisms.

6.8 Error Handling Testing:

Test how the app handles and recovers from unexpected errors or exceptions.
Validate that error messages are clear, meaningful, and do not expose sensitive information.

Code Coverage Analysis:

Use tools to measure the code coverage achieved by the tests, such as statement coverage, branch coverage, or path coverage.
Aim for high code coverage to ensure that all parts of the code are exercised.
During white box testing, the tester has access to the application's internal code, allowing for amore detailed examination of its behavior.

6.9 System Testing :

System testing is a level of software testing that evaluates the complete system as a whole, rather than focusing on individual components or modules. It ensures that all components of the KIET Event Management App work together seamlessly and meet the specified requirements. Here are some examples of system testing techniques that can be applied to the app:

6.10 Functional Testing:

Verify that all functional requirements of the app are met. Test various functionalities such as event creation, registration, club directory search, user log in and registration, event notifications, etc. Validate that the app behaves as expected and produces the correct outputs based on different inputs.

6.11 User Interface Testing:

Test the graphical user interface (GUI) of the app for usability, consistency, and responsiveness.

Check the layout, navigation, buttons, forms, and other UI elements to ensure they are visually appealing and intuitive.

Validate that the app adheres to the design guidelines and provides a seamless user experience.

6.12 Performance Testing:

Evaluate the performance of the app under different load conditions. Measure response times, throughput, and resource utilization to ensure the app can handle the expected user load without significant degradation. Identify and address any performance bottlenecks or scalability issues.

6.13 Compatibility Testing:

- Test the app on different devices, platforms, and browsers to ensure compatibility. Verify that the app works correctly on various operating systems (e.g., iOS, Android) and different screen sizes.
- Validate that the app functions properly on different web browsers (if applicable).

6.14 Security Testing:

- Assess the app's security measures to protect user data and prevent unauthorised access. Perform vulnerability scanning, penetration testing, and authentication testing to identify and address any security vulnerabilities.
- Test the app's resilience against common security threats, such as cross-site scripting (XSS) and SQL injection.

6.15 Integration Testing:

- Test the integration of the app with external systems, such as databases, mapping services, or notification services.
- Validate that data is exchanged correctly between the app and external systems.
- Verify that the app's functionality remains intact when integrated with other systems.

6.16 Recovery Testing:

- Simulate system failures or interruptions and evaluate the app's ability to recover and resume normal operation.
- Test scenarios such as unexpected shutdowns, network failures, or interrupted database connections.
- Ensure that the app can gracefully handle such situations and recover without data loss or integrity issues.

6.17 Regression Testing:

- Re-test previously tested features and functionalities to ensure that recent changes or additions did not introduce new bugs or regressions.
- Execute a set of comprehensive test cases to cover critical areas of the app and ensure that no existing functionality is compromised.

CHAPTER 7

RESULTS

7.1 Landing Page

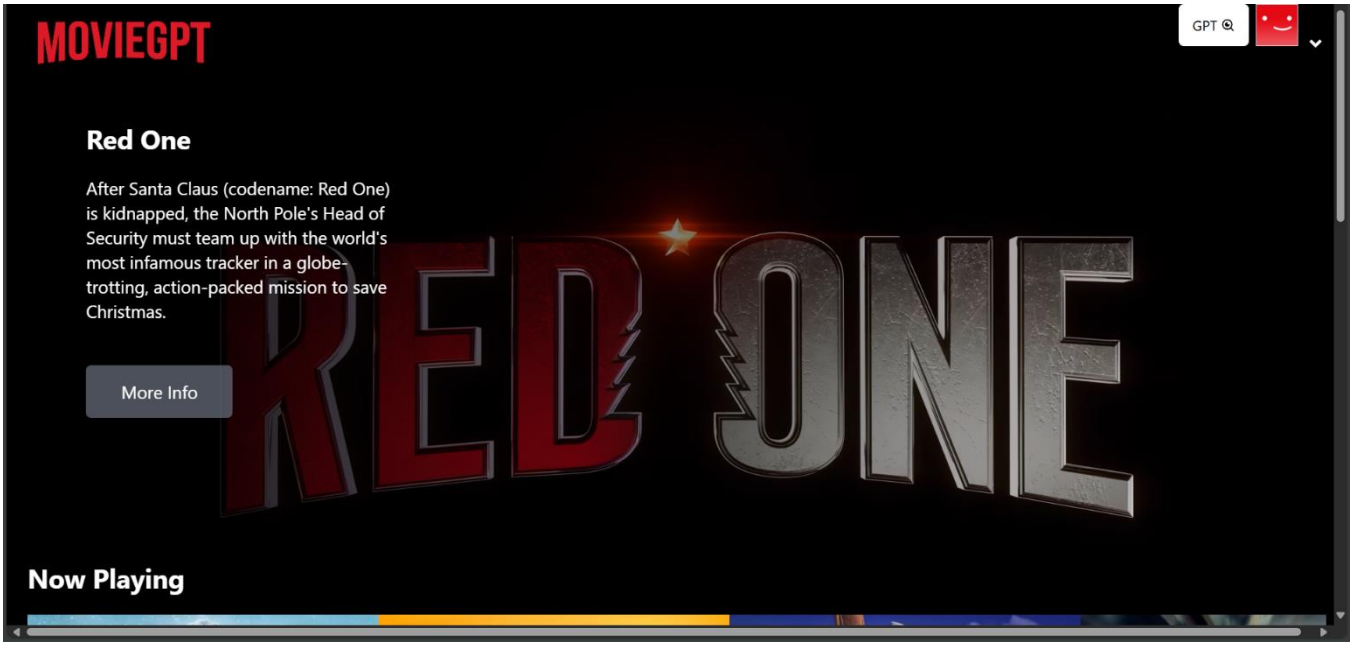

Fig 7.1 Home page

The image shows a webpage of a Movie Recommendation site named "MovieGpt,". This page only bring the options to either login or sign up.

7.2 Registration Page

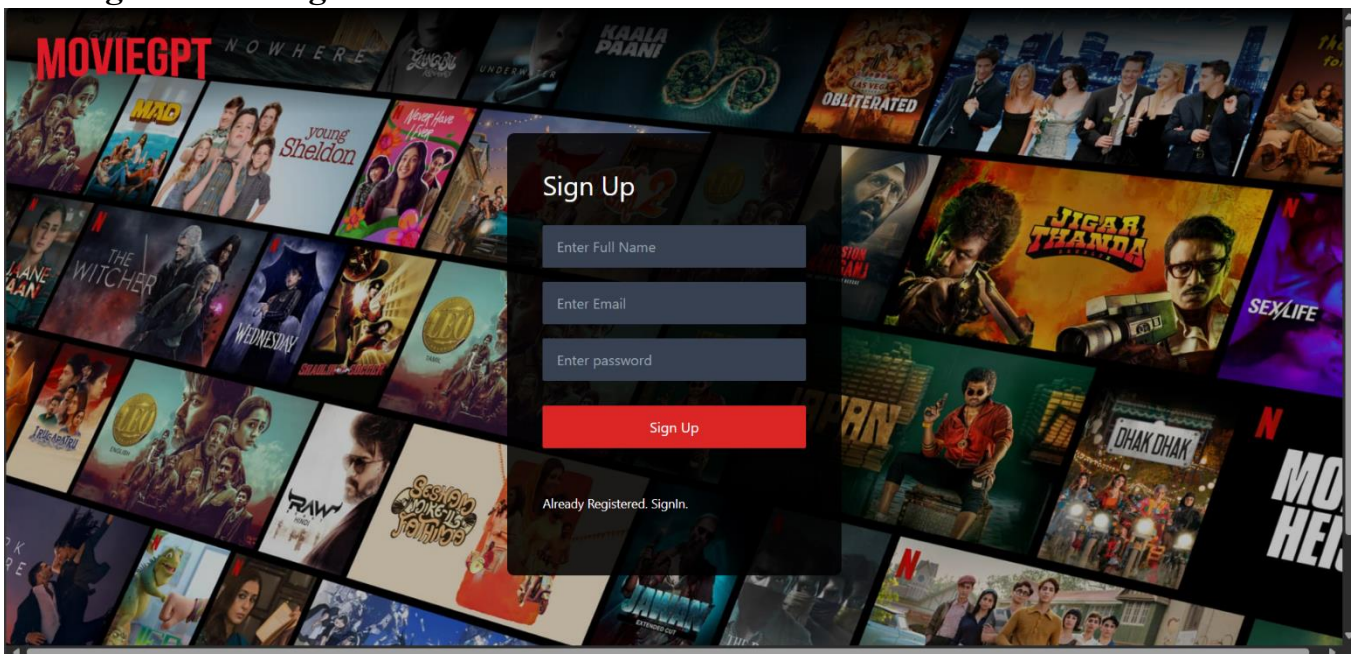

Fig 7.2 Registration page

7.3 Login Page

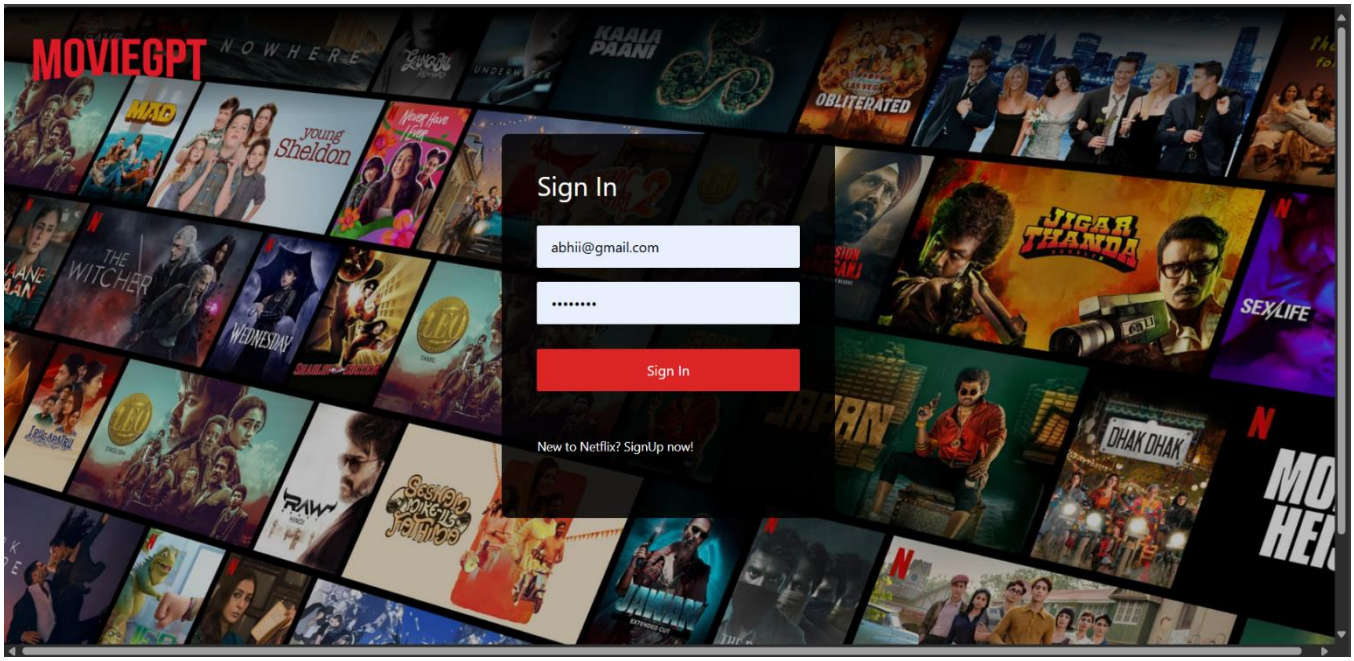

Fig 7.3 Login Page

7.4 Home Page

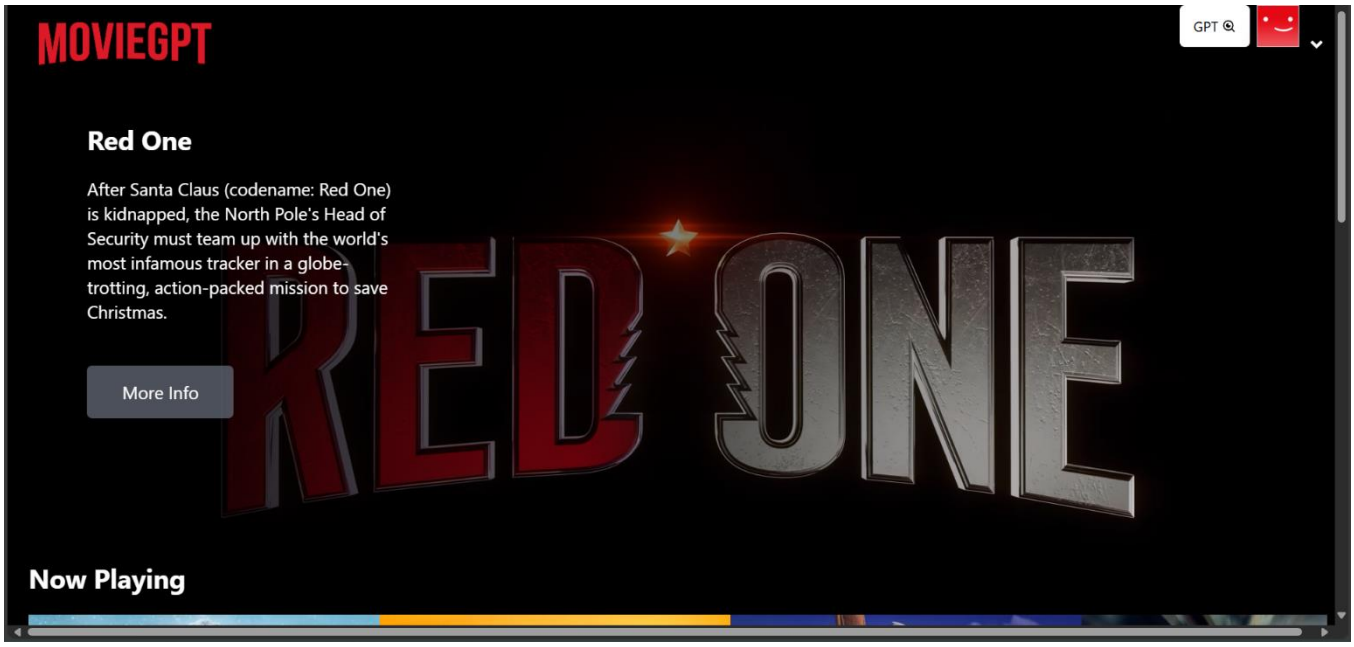

Fig 7.4 Home Page with GPT search Button

The **Home Page** of MovieGPT serves as the central hub for users to explore recent and upcoming movies:

1. **Dynamic Trailers**: A video trailer of the latest or most anticipated movie plays automatically, offering a glimpse into trending content. This visually engaging element creates an immersive user experience.
2. **Top Movies Section**: A curated list of the highest-rated and trending movies is displayed. Users can quickly identify popular films based on reviews and engagement.
3. **Upcoming Movies Section**: A dedicated section highlights upcoming movie releases with posters and titles, keeping users informed about future entertainment options.
4. **Navigation Bar**: The top navigation bar includes links to other pages, such as the GPT Search Page, user profile, and more, ensuring seamless browsing.

The Home Page design prioritizes an interactive and visually rich experience, ensuring users feel captivated the moment they visit the platform.

7.5 GPT Search Page

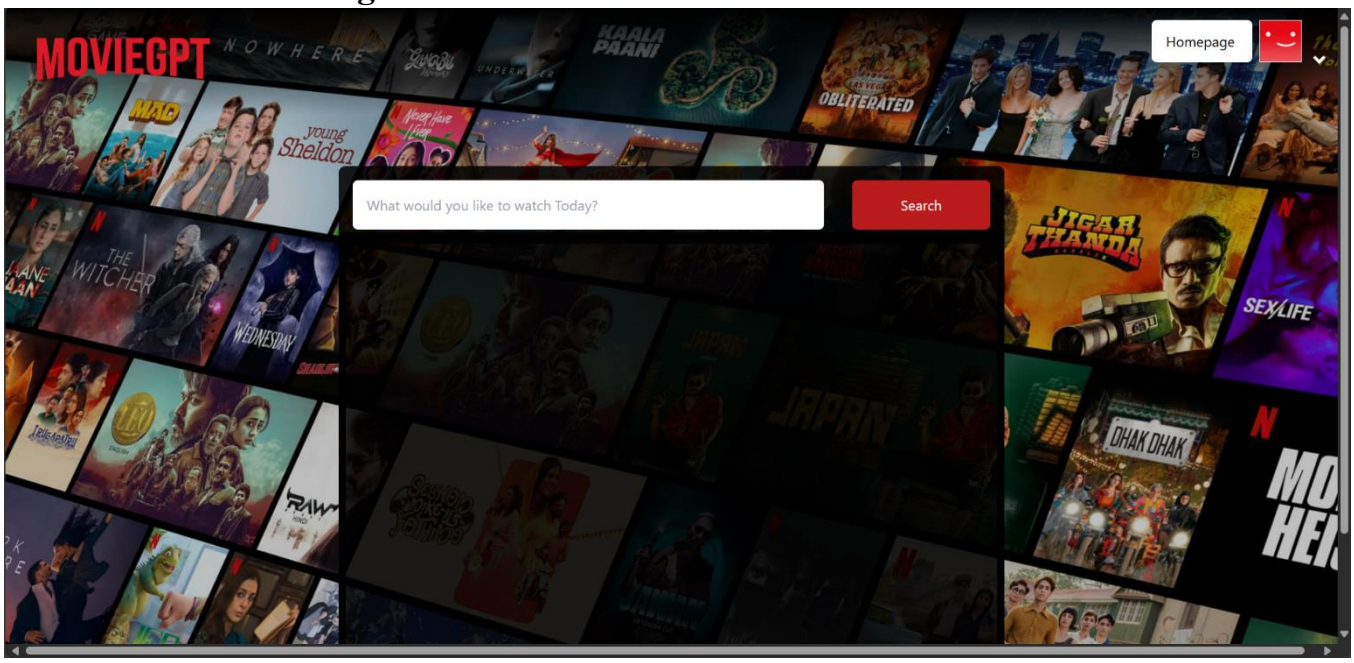

Fig 7.5 Gpt Search Page

The GPT Search Page allows users to interact with an intelligent movie recommendation system through natural language prompts:

1. Custom Movie Search: Users can enter specific prompts such as *"Comedy movies with a twist"* or *"Action-packed thrillers from the 90s"*. The system processes these inputs and provides tailored movie recommendations.
2. Interactive Interface: A prominent search bar allows users to enter their queries easily, while a responsive button executes the search.
3. Search Results: The page dynamically updates to display a list of movies that match the custom query, including posters, titles, and summaries.
4. GPT-Driven Intelligence: Powered by GPT, the search functionality leverages advanced AI to deliver highly relevant and nuanced results, setting MovieGPT apart from traditional movie recommendation systems.

This page enhance user engagement by providing a personalized and intelligent way to explore movies based on unique interests or moods.

7.6 Home Page

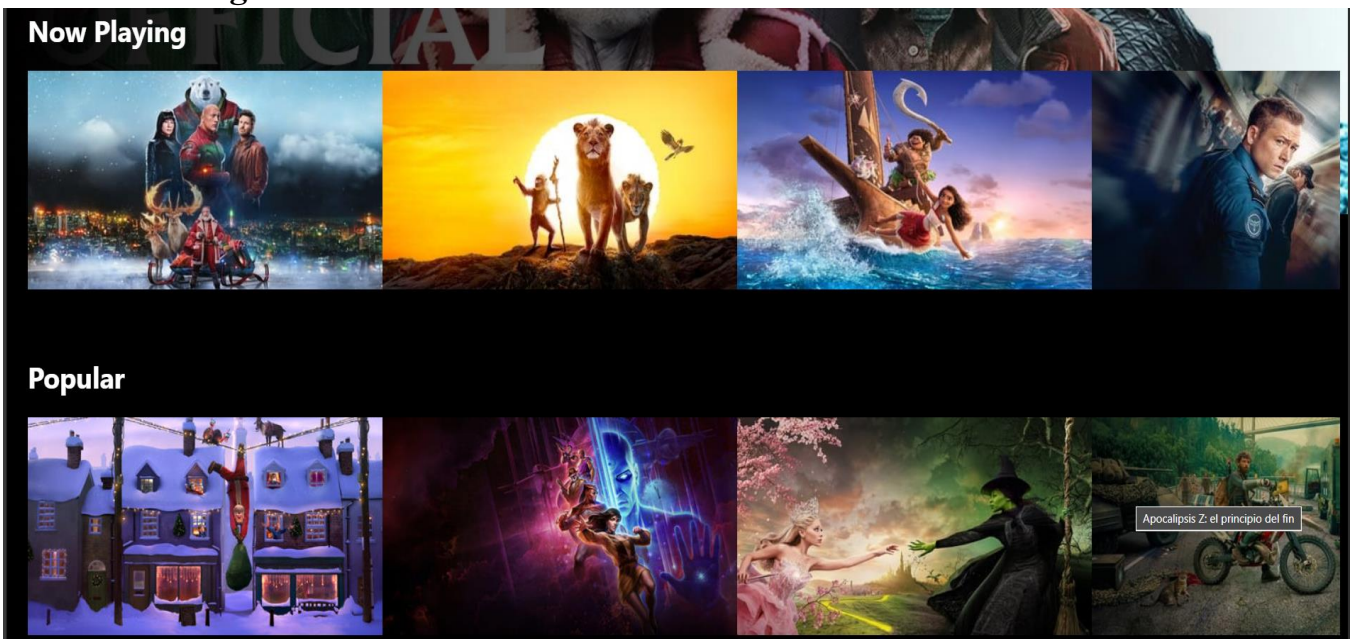

Fig 7.6

The **Home Page** of MovieGPT is designed to captivate users from the very first glance, blending entertainment with an interactive, seamless experience.

1. **Cinematic Experience Right at Your Fingertips**:

   A high-definition trailer of the latest blockbuster movie welcomes users, instantly immersing them into the world of cinema. The dynamic and visually striking video player at the center of the page brings the excitement of a movie theater to the comfort of your device.

2. **Explore Top Hits and Anticipate Upcoming Gems**:

   Below the trailer, users can dive into a curated selection of the **Top Movies**—a showcase of critically acclaimed and trending films, ensuring they're always in the know about the must-watch content. Adjacent to this, the **Upcoming Movies** section serves as a sneak peek into the future of entertainment, featuring posters and titles that build anticipation for the next big release.

3. **Elegant and User-Friendly Design**:

   The sleek and intuitive layout ensures effortless navigation. Users can scroll through recommendations, hover over posters for quick insights, or navigate to other sections with a single click. The bold typography and vibrant visuals create a perfect balance of style and functionality, making every visit an engaging experience.

4. **A Personalized Gateway to Entertainment**:

   The Home Page is more than just a static landing screen—it's a gateway to a world of tailored movie content. Whether you're a casual viewer or a die-hard movie enthusiast, this page sets the tone for an unforgettable journey through the world of cinema.

CHAPTER 8

CONCLUSION

In conclusion, **MovieGPT** is a robust and user-friendly web application that allows users to search for and save their favorite movies while maintaining a smooth and responsive experience across devices. The project has provided extensive learning opportunities, from developing a comprehensive frontend using React.js to integrating Firebase services for authentication and real-time data management.

The future scope includes the addition of more advanced features such as personalized recommendations, external API integrations, and mobile app development. The project has laid the groundwork for future improvements, and I am excited to continue building on this foundation to create an even more powerful and engaging movie tracking platform.

8.1 Summary of Achievements

The **MovieGPT** project was successfully developed with a focus on creating a movie recommendation and tracking platform that combines dynamic frontend features and a robust backend infrastructure. Some key achievements include:

- **User Authentication System**:
  - Implemented a secure and easy-to-use user authentication system using **Firebase Authentication**. Users can sign up, log in, and maintain session persistence across the application.
- **Movie Search and Display**:
  - Integrated a movie search feature that allows users to search for movies by title, genre, or other keywords. The search functionality pulls movie data in real-time from the Firestore database.
- **Watchlist Functionality**:

- Developed a feature allowing users to save their favorite movies in a personalized **watchlist**, which is stored in **Firebase Firestore**. Users can add or remove movies with a single click.

- **Responsive Design**:
  - Ensured that the application is responsive across various devices using CSS and media queries. This guarantees a seamless user experience on desktops, tablets, and mobile phones.
- **Real-time Data Updates**:
  - Leveraged Firebase's real-time capabilities to provide instant updates when movies are added or removed from the watchlist. This ensures users have an up-to-date experience without needing to refresh the page.
- **Easy Deployment**:
  - Deployed the application to **Firebase Hosting**, which provides fast and secure delivery of content globally. The serverless architecture allowed for rapid deployment with minimal maintenance.

8.2 Reflections on the Project and Learning Outcomes

Building the **MovieGPT** project provided valuable learning experiences and allowed me to expand my skills in both frontend and backend technologies. Below are key reflections and learning outcomes:

- **Frontend Development with React**:
  - Through this project, I gained a deeper understanding of **React.js**, including state management with **React Context API** and handling routing with **React Router DOM**. These concepts allowed me to create a dynamic and responsive UI with reusable components.
- **Firebase Integration**:
  - I learned how to integrate **Firebase** for both **Authentication** and **Firestore**, which simplified the backend setup by eliminating the need to create and manage a separate server. Firebase's real-time database capabilities enhanced the app's responsiveness and performance.
- **UI/UX Design Considerations**:
  - Designing a smooth and intuitive user experience was critical. I learned the importance of responsive design and how to implement a UI that functions well across different screen sizes and devices.

- **Project Management and Deployment**:
    - I also became familiar with managing a project through version control (using **Git**) and deploying it to **Firebase Hosting**. This taught me the importance of efficient code management and proper deployment strategies for web applications.

CHAPTER 9

FUTURE SCOPE

9.1 Future Scope

While the current version of **MovieGPT** meets the core requirements, there are several opportunities to enhance the platform and provide additional features:

- **Integration with Movie APIs (TMDB/OMDB)**:
  - Although the application currently relies on Firebase for data storage, integrating external movie databases like **TMDB** (The Movie Database) or **OMDB** (Open Movie Database) could significantly improve the scope of movie information available. This could include ratings, reviews, trailers, and more accurate movie data.
- **Advanced Movie Recommendations**:
  - Implementing a movie recommendation algorithm based on user preferences and watch history could enhance user engagement. Machine learning or collaborative filtering could be explored to provide tailored movie suggestions.
- **User Reviews and Ratings**:
  - Adding functionality for users to rate and review movies would create a more interactive platform. These ratings could be used for sorting movies in the search results or influencing recommendations.
- **Social Features**:
  - Incorporating social features, such as the ability to share watchlists with friends, follow other users, or comment on movies, could make the platform more engaging and community-driven.
- **Mobile App Version**:

    - Developing a mobile application using **React Native** or **Flutter** could provide a native mobile experience for users, ensuring they have access to their watchlist and movie recommendations on the go.
- **Analytics and User Feedback**:
    - Integrating **Google Analytics** or Firebase's native analytics tools could help track user behavior and interactions within the app. This data can be used to refine the user experience and optimize features based on usage patterns.
- **Admin Panel for Movie Management**:
    - An **admin panel** could be introduced to allow admins to manage the movie database, moderate user-generated content (such as reviews), and perform maintenance tasks.

CHAPTER 10

BIBLIOGRAPHY

1. **Books and Articles**
    - *"Learning React: Functional Web Development with React and Redux"* by Alex Banks and Eve Porcello
    - *"Firebase Essentials"* by Charles McKeever
2. **Online Documentation**
    - React Documentation: https://reactjs.org/docs/getting-started.html
    - Firebase Documentation: https://firebase.google.com/docs
    - React Router Documentation: https://reactrouter.com/
    - TMDB API Documentation: https://www.themoviedb.org/documentation/api
    - MDN Web Docs: https://developer.mozilla.org/
3. **Tutorials and Blog Posts**
    - *"How to Set Up Firebase Authentication in React"* - Medium Article
    - *"Build a React App with Firebase Firestore"* - DigitalOcean Tutorial
4. **Tools and Libraries**
    - React.js: https://reactjs.org/
    - Firebase: https://firebase.google.com/
    - Axios: https://axios-http.com/
    - Material UI: https://mui.com/
5. **Online Courses**
    - *"React - The Complete Guide"* by Maximilian Schwarzmüller - Udemy
    - *"Firebase for Web"* by Stephen Grider - Udemy
6. **Miscellaneous**
    - GitHub: https://github.com/
    - Stack Overflow: https://stackoverflow.com/