MinSystems LLC

Presenting:

MinFlix Streaming Application

Produced by: Spencer Burke, Ari Torczon, Anthony Jimenez, Raymond Godina, Christian Klindt

Publication Date:
May 5, 2025

1. Introduction

1.1 Company Description

MinSystems LLC is a company that provides digital solutions to its clients via software. We are a relatively new company that wishes to stand out, and we hope to do that with our first project. Here at **MinSystems LLC** we hope to create a streaming service that can compete with big services within the market. We call it Minflix, and we strive to have a vast and appealing catalog to users. We hope to make a system that is not only user friendly but also provides many quality-of-life features to the user's liking.

---

2. Problem Description

2.1 Overview of Problem Domain

The domain of film streaming is currently dominated by large-scale, closed-source platforms such as **YouTube**, **Netflix**, **Hulu**, and **Amazon Prime**. These services typically require paid subscriptions and offer limited transparency in terms of their underlying technology. As a result, developers and smaller organizations have minimal access to open-source resources, libraries, or comprehensive documentation tailored specifically for building full-featured streaming platforms. This creates a significant barrier for educational, experimental, or independent use cases.

2.2 Client Vision

The client's vision for this project was to create a foundational codebase for a film streaming application that not only supports video playback and user interaction but also incorporates robust user tracking capabilities. This system would be built using widely adopted tools and frameworks, offering a maintainable, modular, and transparent solution that can serve as a base for further development or customization.

---

3. Solution Approach

3.1 Overview of Solution

To address the requirements of a modern streaming application, we leveraged widely adopted frameworks—React for the front-end and FastAPI for the back-end—to deliver film content directly to users' browsers with minimal latency. All user interactions and viewing metrics are persisted in a robust relational database, ensuring that relevant information is recorded reliably and can be queried efficiently. Emphasis was placed on maintaining a clean and simple codebase, with clear separation of concerns and modular components to facilitate future enhancements. Finally, the entire system was deployed to a public web host, providing a fully functional, accessible demonstration of the Minflix Streaming Application as a real-world product.

---

4. Design

4.1 Overview of Design

This document describes the data model for MinFlix based on the entity-relationship diagram (see Figure 1 below). It details the structure of the database, including entities, attributes, and relationships. This model supports our functional requirements by organizing user information, film data, and user interactions efficiently.

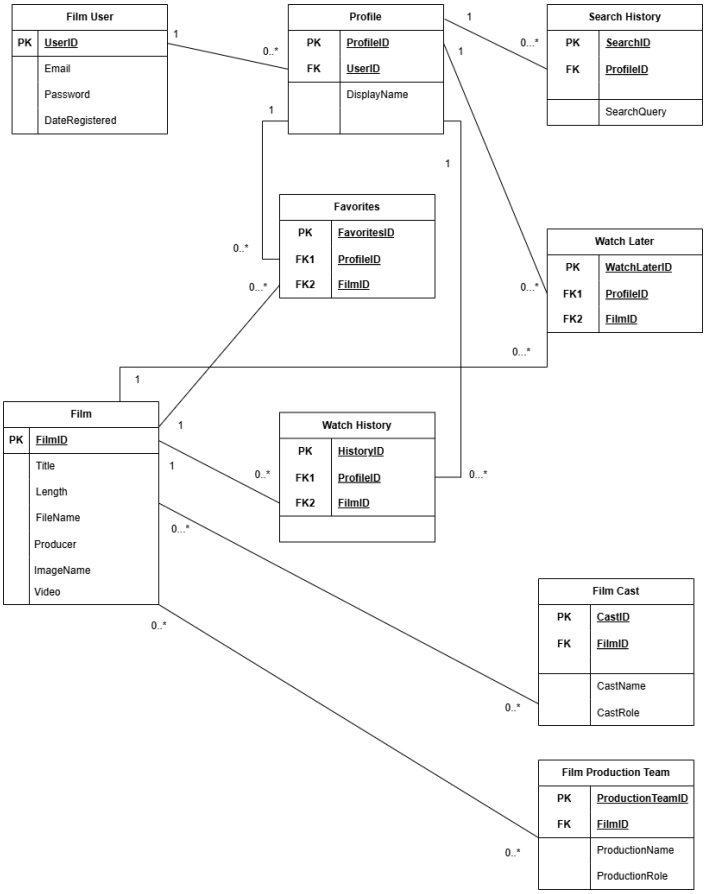


Figure 1. MinFlix Data Model Diagram

4.2 Entities and Their Descriptions

a. FilmUser

Purpose: Stores login credentials and account-level information. This includes email, password, the date registered. The date registered will provide meta information about the account.

Key Attributes:

- **UserID (PK):** Unique identifier for **User**.
- **Email:** User email address.
- **Password:** Encrypted password.
- **DateRegistered:** Timestamp of account creation.

Relationships (Figure 1):

- One-to-many with **Profile**.

---

b. Profile

Purpose: Holds individual user personal details. This will hold the chosen display name of the profile. Contains the **UserID** as foreign key to connect them. This is because user accounts have several profiles and this foreign key expresses such a relationship.

Key Attributes:

- **ProfileID (PK):** Unique identifier for **Profile**.
- **UserID (FK):** References **FilmUser**.
- **DisplayName:** User's chosen name that will be seen when choosing profiles and profile dashboard.

Relationships (Figure 1):

- Many-to-one with **FilmUser**.
- One-to-many with **WatchHistory**, **WatchLater**, **Favorites**, and **SearchHistory**.

---

c. Film

Purpose: Contains film-related metadata. This will include the title, the length of the film, the genre, date of release, the thumbnail for playing the movie, and the video itself.

Key Attributes:

- **FilmID (PK):** Unique identifier for **Film**.
- **Title:** This holds the title for films.
- **Length:** The length of the film.
- **FileName:** The video file path.
- **Producer:** The name of the producer of the film.
- **ImageName:** This will be a promotional image for the film that will be seen when browsing for films.

- **Video:** This is the actual video of the film that is streaming.

Relationships (Figure 1):

- Many-to-many with **FilmCast**.
- Many-to-many with **FilmProductionTeam**.
- One-to-many with **WatchHistory**, **WatchLater**, and **Favorites**. This is seen in **Figure 1.**

---

d. WatchHistory

Purpose: Records which films a user has watched and relevant details. Users may want to watch movies they have not finished and continue from where they left off. This will also show what movies they have seen.

Key Attributes:

- **HistoryID (PK):** Unique identifier for **WatchHistory.**
- **ProfileID (FK):** References **Profile.**
- **FilmID (FK):** References **Film.**
- **Timestamp:** This will show the last part of the film the user profile left off on.
- **Completion:** Shows completion of a completed film.

Relationships (Figure 1):

- Many-to-one with **Profile** and **Film**.

---

e. WatchLater

Purpose: Lists films the user intends to watch. There will be a option for the user to choose what films they want to watch later

Key Attributes:

- **WatchLaterID (PK):** Unique identifier for **WatchLater.**
- **ProfileID (FK):** References **Profile.**
- **FilmID (FK):** References **Film.**
- **DateAdded:** The date it was added to the watch history list**.**

Relationships (Figure 1):

- Many-to-one with **Profile** and **Film.**

---

f. Favorites

Purpose: Stores films the user has marked as favorites. This is similar to watch history.

Key Attributes:

- **FavoritesID (PK):** Unique identifier for **Favorites**.
- **ProfileID (FK):** References **Profile**.
- **FilmID (FK):** References **Film.**
- **FavoritedDate:** The date it was added to the favorites list**.**

Relationships (Figure 1):

- Many-to-one with **Profile** and **Film**.

g. SearchHistory

Purpose: Tracks user search queries.

Key Attributes:

- **SearchID (PK):** Unique identifier for **SearchHistory.**
- **ProfileID (FK):** References **Profile.**
- **SearchQuery:** The searches for films the user made.

Relationships (Figure 1):

- Many-to-one with **Profile**.

---

h. FilmCast

Purpose: Represents individual cast members. States their name and role.

Key Attributes:

- **CastID (PK):** Unique identifier for **FilmCast**.

- **FilmID (FK):** References **Film.**
- **CastName:** Name of the cast member.
- **CastRole:** The role of the cast member**.**

Relationships (Figure 1):

- Many-to-many with **Film**.

---

i. FilmProductionTeam

Purpose: Represents crew members (directors, producers, etc.).

Key Attributes:

- **ProductionTeamID (PK):** Unique identifier for **FilmProductionTeam.**
- **FilmID (FK):** References **Film.**
- **ProductionName:** Name of member in production team.
- **ProductionRole**: The role of the member in the production team.

Relationships (Figure 1):

- Many-to-many with **Film**.

4.3 Token Models

While not part of the persistent database schema, the application uses a set of Pydantic models—TokenModel, TokenProfileDataModel, and related submodels—to structure authentication and session-related data. These models define the structure of JWT (JSON Web Token) payloads that are sent to the frontend upon user login or profile updates. For example, TokenProfileDataModel includes nested lists such as watch history, favorites, and watch later, ensuring the client has synchronized access to all necessary user content. These models mirror the database structure but are used only for temporary transport of user state.

4.4 Relationships

Refer back to **Figure 1** for all of these connections.

- **FilmUser - Profile:** One user can have multiple profiles, but a profile can only have one user.
- **Profile - WatchHistory:** Each profile can have multiple films in their watch history, while a watch history can only be linked back to one profile.
- **Profile - WatchLater:** A profile can have multiple items in their watch later list, but the watch later list is for only one profile.
- **Profile - Favorites:** A profile can have multiple items in their favorites list, but the watch later list is for only one profile.
- **Profile - SearchHistory:** A profile can have multiple things searched in their search history, but the search history is only for that one profile.
- **Film - WatchHistory:** Each film may appear multiple times in users' watch history. Each item in the watch history can be linked to one film.
- **Film - WatchLater:** A film can be in multiple users' watch later, but a watch later can have a film appear in it only once.
- **Film - Favorites:** A film can be in multiple users' favorites, but a favorites section can have a film appear in it only once
- **Film - FilmCast:** A film can have multiple people as their cast, and a cast member can be a part of many movies.
- **Film - FilmProduction:** Films can have many members of a production team, and a production team member could work on many films.

---

5. Architecture

5.1 Overview of Architecture

This document describes the architecture design of MinFlix, referencing both the back-end-focused component diagram (**Figure 2**) and an API/front-end-focused component diagram (**Figure 3**). These diagrams provide insight into the system's modular structure, interactions, and dependencies. This design ensures a scalable and maintainable system for handling user authentication, film browsing, and streaming functionalities.

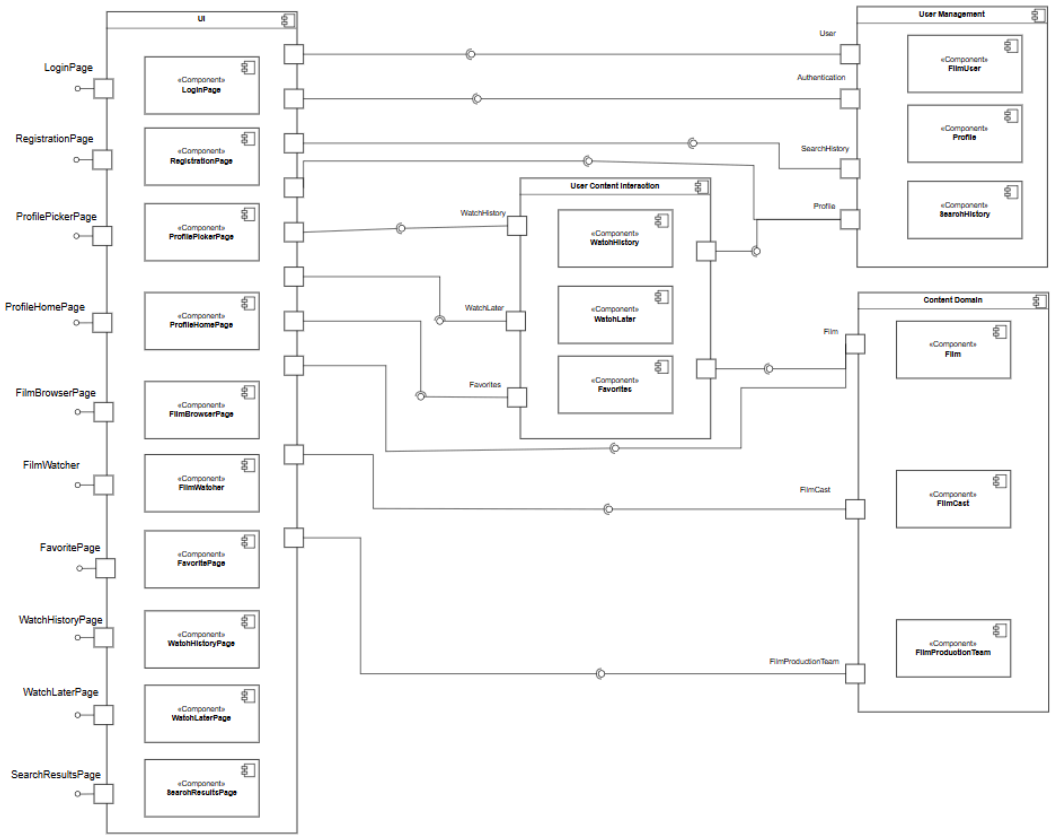


Figure 2. Architecture Design for back-end focused components

---

5.2 Back-end focused Components

a. User Management Component

Purpose:
This component contains everything that the user can't interact with themselves, but manages user information. This involves user accounts, profiles, and search history.

Sub-components (Refer to Figure 1):

- **FilmUser:** Manages authentication and account credentials.
- **Profile:** Manages user-specific settings like display name.
- **SearchHistory:** Records search queries.

Provided Interfaces (Refer to Figure 1):

- **FilmUser:** For registration, login, and account management.
- **Profile:** For retrieving and updating user profile information.
- **Authentication:** For handling authentication and sessions.
- **SearchHistory:** For managing search records.

---

b. Content Domain Component

Purpose:
Handles all Film-related data and metadata. Users will be able to see details of the film from things like the cast and even the production team.

Sub-components (Refer to Figure 2):

- **Film:** The actual data of the film such as the video.
- **FilmCast:** Actor and on-screen talent data.
- **FilmProductionTeam:** Data of the crew, such as directors, producers, etc.

Provided Interfaces (Refer to Figure 2):

- **Film:** For retrieving film metadata.
- **FilmCast:** For retrieving cast information.
- **FilmProductionTeam:** For retrieving production crew details.

---

c. User Content Interaction Component

Purpose:
This is all content the user can interact with themselves. This data is then captured and recorded.

Sub-components (Refer to Figure 2):

- **WatchHistory:** Records films that have been viewed by the user.
- **WatchLater:** Maintains a list of films saved for future viewing.
- **Favorites:** Records films marked as favorites in a list.

Provided Interfaces (Refer to Figure 2):

- **WatchHistory:** For watch history management.
- **WatchLater:** For managing watch later lists.

- **Favorites:** For managing favorite films.

Required Interfaces (Refer to Figure 2):

- **Profile:** From the User Management component to associate interactions with a user profile.
- **Film:** From the Content Domain component to link interactions to specific films.

---

d. UI Component

Purpose:
Provides the presentation layer for users. This will be where users can interact with a functional interface.

Sub-components (Refer to Figure 2):

- **LoginPage:** For user authentication.
- **RegistrationPage:** For registering new users.
- **ProfilePickerPage:** For displaying and editing user profile data.
- **ProfileHomePage:** Dashboard for film listings and navigation bar.
- **FilmBrowsingPage:** For displaying a list of films.
- **FilmWatcher:** For displaying a chosen film.
- **FavoritePage:** For displaying favorite films.
- **WatchHistoryPage:** For displaying the history of films watched.
- **WatchLaterPage:** For displaying the films marked for watch later.
- **SearchResultsPage:** For displaying search results.

Provided Interfaces (Refer to Figure 2):

- **LoginPage:** UI capabilities for the login screen.
- **RegistrationPage:** UI capabilities for the registration screen.
- **ProfilePickerPage:** UI capabilities for the profile picking screen.
- **ProfileHomePage:** UI capabilities for the profile home screen.
- **FilmBrowsingPage:** UI capabilities for the film browsing screen.
- **FilmWatcher:** UI capabilities for film watching screen.
- **FavoritePage:** UI capabilities for favorite films screen.
- **WatchHistoryPage:** UI capabilities for watch history screen.
- **WatchLaterPage:** UI capabilities for watch later screens.
- **SearchResultsPage:** UI capabilities search results screen.

Required Interfaces (Refer to Figure 2):

- The UI component depends on all back-end services since everything is linked to the UI.
    - **From User Management:** FilmUser, Profile, Authentication, SearchHistory.
    - **From Content Domain:** Film, FilmCast, FilmProduction.
    - **From User Content Interaction:** WatchHistory, WatchLater, Favorites.

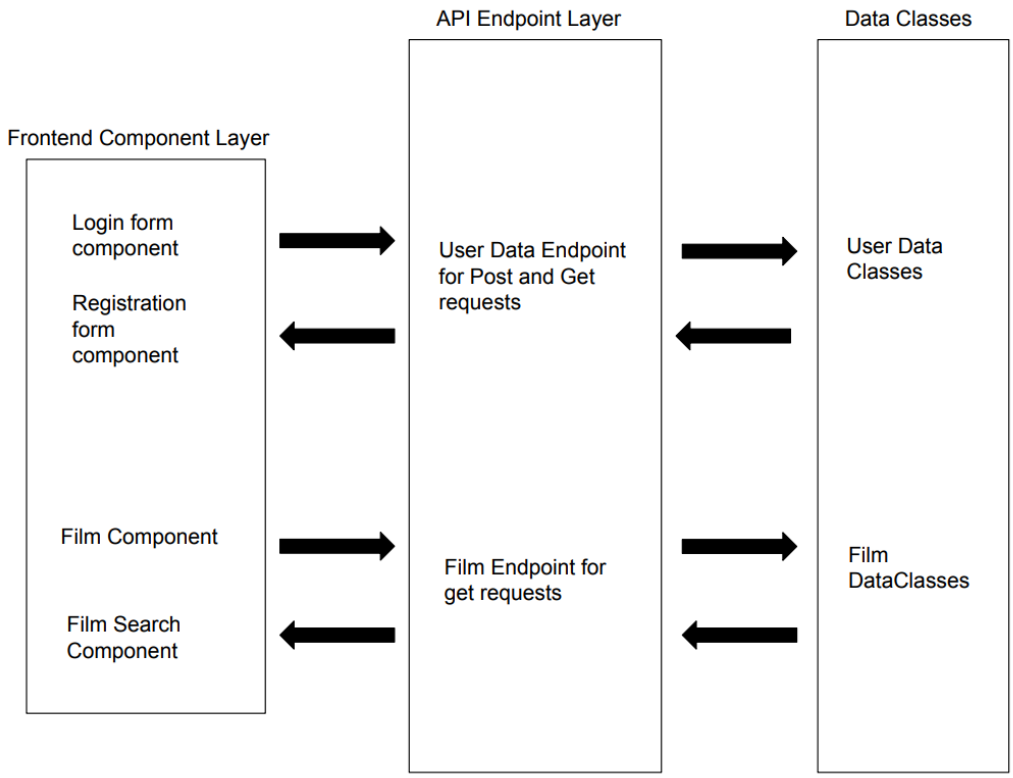


Figure 3. Architecture Design for API/front-end focused components

---

5.3 API/front-end focused Components

a. Data Classes Layer

Purpose:
These classes organize and transform raw data from back-end services into well-structured objects that can be easily consumed by the UI. They act as an

intermediary, ensuring data consistency and formatting before it reaches higher layers.

Sub-components (Refer to Figure 3):

- **User Data Classes:** Define the structure and format of user-related data objects.
- **Film Data Classes:** Define the structure and format of film-related data objects.

b. API Endpoint Layer

Purpose:
This layer exposes endpoints that deliver data from the back-end to the Data Classes layer. It functions as the communication bridge between back-end services and the front-end, ensuring that the correct data is provided in a usable format.

Sub-components (Refer to Figure 3):

- **User Data Endpoint:** Handles POST and GET requests for user-related operations such as login, registration, and profile updates.
- **Film Data Endpoint:** Handles GET requests to retrieve film-related information including metadata, thumbnails, and streaming links.

c. front-end Component Layer

Purpose:
This layer is responsible for presenting the data to the end user and capturing user interactions. It uses the structured data from the Data Classes layer, supplied via the API Endpoint layer, to display dynamic content and drive user experience.

Sub-components (Refer to Figure 3):

- **Login Form Component:** Provides the interface for user authentication.
- **Registration Form Component:** Provides the interface for new account creation.
- **Film Component:** Displays film details, including title, thumbnail, and playback options.
- **Film Search Component:** Enables users to search for films within the application.

5.4 JWT Token and Profile Data Flow

Token-based session management is implemented using Pydantic models such as TokenModel, TokenProfileDataModel, and supporting types like TokenWatchLaterDataModel and TokenFavoriteDataModel. These structures are serialized into JWTs that encode the user's session state. After login or profile modification, the backend returns an updated token to the frontend, allowing seamless access to personalized data like watch history or favorites without requiring repeated API calls. These token models serve as an essential bridge between the **User Management** and **User Content Interaction** components in the architecture, enabling client-side synchronization and secure session handling.

5.5 Architecture Interactions

- **UI - User Management (Refer to Figure 2):** The UI requires user data, user authentication, profile data, and search history. Dependency arrows from UI's required ports point to the provided interfaces on the User Management component.
- **UI - Content Domain (Refer to Figure 2):** The UI requires film details, cast, and production crew information.
- **UI - User Content Interaction (Refer to Figure 2):** The UI requires access to the user's watch history, watch later lists, favorites.
- **User Content Interaction - User Management (Refer to Figure 2):** The User Content Interaction requires profile from user management to link any interaction with watch history, watch later, or favorites to a specific profile.
- **User Content Interaction - Content Domain (Refer to Figure 2):** The User Content Interaction requires film from user management to link any interaction with watch history, watch later, or favorites relating to a specific film.
- **API Endpoint Layer - Data Classes Layer (Refer to Figure 3):** The API Endpoint Layer depends on the Data Classes Layer to provide structured and consistent data formats for user and film data.
- **Front-end Component Layer - API Endpoint Layer (Refer to Figure 3):** The front-end Component Layer relies on the API Endpoint Layer to retrieve the formatted data needed for display.

---

6. Implementation

6.1 Overview of Implementation

This project is a full-stack streaming service web application named **MinFlix**, designed to provide users with an interactive interface to browse, watch, and manage classic films. The platform supports user registration, profile management, and session-based authentication via JWT. Users can curate lists of favorite films and watch-later items, and stream media through a responsive interface. The system is divided into two main repositories which deal with the **Frontend** and **Backend**, each with specific responsibilities and technologies.

6.2 Stack and Deployment

a. Technology Stack

Frontend:

The frontend is built using ReactJS for component-based rendering and React Router for dynamic navigation. JWT-decode is used to extract payload data from authentication tokens, and Jest provides support for frontend unit testing.

- ReactJS
- React Router
- JWT-decode
- Jest
- HTML
- CSS

Backend:

The backend is implemented using FastAPI, a modern, high-performance Python web framework. Data is modeled using SQLModel, and schemas are validated using Pydantic. Authentication is handled securely using bcrypt for hashing and jose for generating/verifying JWTs. The backend is tested with Pytest and uses Pydantic settings for environment configuration.

- FastAPI
- SQLModel
- Pydantic
- Pydantic-Settings

- Bcrypt
- Jose
- Psycopg
- Pytest

Database:

- PostgreSQL

b. Deployment Strategy

Frontend:

The frontend is deployed using Cloudflare Pages. A CI/CD pipeline automatically builds the React project and deploys it to Cloudflare's global CDN. Environment variables are used to configure the API endpoint based on environment (e.g., production vs. local development).

- Built with Cloudflare CI/CD pipeline
- Served through Cloudflare CDN
- Uses environment variables to configure API base URL

Backend:

The backend is containerized using Docker and deployed on Railway.com. The Dockerfile defines all necessary dependencies and mounts a static volume for static media files (films and images). Environment variables configure the app's behavior on deployment.

- Docker container deployment
- Mounted static volume (films, images)
- Environment variable configuration
- PostgreSQL database also hosted via Railway

6.3 Frontend Major Features and Component

a. LoginPage.js

Purpose: Serves as the entry point for users to authenticate and access their profiles.

Key Functionality:

- <AuthenticationForm isLogin={true} />: A reusable form component customized for login functionality.
- useNavigate(): Enables navigation to the registration page.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the GradientBackground component for consistent background styling.

Main Logic Snippet:

```
<h1>Login Page</h1>
<AuthenticationForm isLogin={true} />
<button onClick={() => navigate('/register')}>Go to Registration</button>
```

The second line is the reusable form component that the user will use to enter their information to authenticate login. The next line transitions the user to the registration page using React Router when clicked.

Design Pattern Used:

- **Adapter pattern:** This is through the AuthenticationForm which is used in the login and register logic.

---

b. RegistrationPage.js

Purpose: Enables new users to create an account on MinFlix.

Key Functionality:

- <AuthenticationForm isLogin={false} />: Same reusable form but adapted for registration.
- useNavigate(): Provides backward navigation to the login screen.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the GradientBackground component for consistent background styling.

Main Logic Snippet:

```
<AuthenticationForm isLogin={false} />
<button onClick={() => navigate('/')}>Go to Login</button>
```

The first line is the reusable form component that the user will use to enter their information to authenticate registration.Provides easy navigation back to the login page.

Design Pattern Used:

- **Adapter pattern:** This is seen again via the AuthenticationForm.

---

c. ProfilePickerPage.js

Purpose: Allows logged-in users to select or manage profiles. It includes functionality to add and edit profiles using forms and view them as clickable profile buttons.

Key Functionality:

- **JWT Token Validation:** Ensures user session is valid before loading data.
- **Dynamic Profile Loading:** Loads profiles from token data using getTokenData().
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the GradientBackground component for consistent background styling.
- Utilizes the LogoutButton component.

Main Logic Snippet:

```
<ul id='profileList'>
    {profiles.map(profile => (
        <li key={profile.id} id='profileListItem'>
            <button
                id='profileButton'
                onClick={() => handleProfileSelect(profile.id)}
            >
                {profile.displayname}
            </button>
        </li>
    ))}
</ul>
```

This is the code that will render in the circular buttons used to access a profile made in the user accounts.

Design Patterns Used:

- **Command Pattern:** Encapsulation of handleAddProfileSubmit and handleEditProfileSubmit.
- **Decorator Pattern:** GradientBackground wraps the main component for consistent styling.

---

d. ProfileHomePage.js

Purpose: Displays all relevant profile data for the selected user profile, including name, favorites, and watch later film lists.

Key Functionality:

- Retrieves and displays the display name of the selected profile based on the token data.
- Loads and displays the list of favorite films for the current profile.
- Loads and displays the list of watch later films for the current profile.
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the EditProfileForm to allow the user to rename and update their profile.
- Utilizes the ActionButton component for consistent button styling.

- Utilizes the GradientBackground component for consistent background styling.

Main Logic Snippet:

```
for (let i = 0; i < tokenData.profiles.length; i++) {
    if (tokenData.profiles[i].id == profileId) {
        setDisplayName(tokenData.profiles[i].displayname);
    }
}
```

This retrieves and sets the display name for the given profile.

Design Patterns Used:

- **Iterator pattern:** Maps over arrays.

---

e. FilmBrowserPage.js

Purpose: Allows users to browse the complete catalog of films and navigate back to their profile.

Key Functionality:

- Loads all available films from the backend using getFilmData.
- Parses and stores the film data in localStorage for reuse across components.
- Passes the full list of film IDs to the FilmList component for rendering.
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the GradientBackground component for consistent background styling.

Main Logic Snippet:

```
const filmIds = films.map(film => film.id);
// set the film ids
setFilmIds(filmIds);
```

This will load all movies available from the backend.

Design Patterns Used:

- **Iterator pattern:** This is used to loop over all available films.

---

f. FilmWatcher.js

Purpose: Streams the selected film from the backend. Retrieves film data using the filmId from the URL.

Key Functionality:

- Retrieves the film ID from the URL parameters and fetches the corresponding film's metadata from localStorage.
- Constructs the video source URL using the backend's base URL and the film's filename.
- Renders a video player with playback controls for the selected film.
- Displays the film title above the video player.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the GradientBackground component for consistent background styling.

Main Logic Snippet:

```
const film = films.find(film => film.id == filmId);

if (film) {
    setFilmName(film.title);
    setSource(`${API_BASE_URL}/film/${film.file_name}`);
}
```

Finds the correct film and sets the source URL for streaming.

UI Features:

- <video> tag to stream the film.
- Back navigation to film browser.

Design Pattern Used:

- **Adapter pattern**: Input from localStorage adapts into a streaming source.

---

g. App.js

Purpose: The App.js file serves as the routing backbone of the frontend application using react-router-dom. It maps each URL path to a corresponding page component, enabling navigation throughout the application.

Key Functionality:

- Initializes client-side routing.
- Defines routes for login, registration, profile selection, film browsing, and watching.
- Uses React Router's Routes and Route components.

Main Logic Snippet

```
<Route path='/' element={<LoginPage />} />
<Route path='/register' element={<RegistrationPage />} />
<Route path='/profiles' element={<ProfilePickerPage />} />
<Route path='/profile/:profileId' element={<ProfileHomePage />} />
<Route path='/browse/:profileId' element={<FilmBrowserPage />} />
<Route path='/watch/:filmId' element={<FilmWatcher />} />
```

This defines the navigation structure of the app by assigning specific URL paths to individual page components using React Router.

---

h. Network.js

Purpose: Network.js handles all HTTP requests between the frontend and backend. It centralizes all API communication and authentication logic, including login, registration, profile management, and film retrieval.

Key Functionality:

- Encapsulates all API endpoints for login, registration, profile edits, and film interactions
- Adds and verifies JWT-based authentication tokens
- Constructs authenticated fetch requests with appropriate headers
- Handles response parsing and error management

Main Logic Snippet

```
const apiRequest = async (endpoint, options = {}, requiresAuth = false) => {
  const url = `${API_BASE_URL}${endpoint}`;

  // headers
  const headers = {
    ...options.headers,
  };

  // Add authentication if required
  if (requiresAuth) {
    const token = getAuthToken();
    if (!token) {
      throw new Error('Authentication required but no token found');
    }
    headers['Authorization'] = `Bearer ${token}`;
  }

  // Merge options
  const requestOptions = {
    ...options,
    headers,
    credentials: 'include'
  };

  // Make the request
  const response = await fetch(url, requestOptions);

  // Handle common errors
  if (!response.ok) {
    let errorMessage;
    try {
      const errorData = await response.json();
      errorMessage = errorData.detail || `Request failed with status: ${response.status}`;
    } catch (e) {
      errorMessage = `Request failed with status: ${response.status}`;
    }
    throw new Error(errorMessage);
  }

  // Return the response (as text, then caller can parse if needed)
  return await response.text();
```

This function builds and sends a fetch request to the backend API. It optionally adds JWT authentication headers and manages error handling uniformly across all network requests.

Design Pattern Used:

- **Command Pattern**: Each exported function like login, register, addProfile, etc., encapsulates a specific action or command as a function that can be reused by any component needing that behavior.

---

i. FavoritePage.js

Purpose:
Serves as the screen where a profile's favorite films are displayed.

Key Functionality:

- getTokenData(): Reads the JWT payload to extract this profile's favorites.
- Renders the list of favorite films.
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the GradientBackground component for consistent background styling.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the Navbar component for organized look for searching.

Main Logic Snippet:

```
useEffect(() => {
    loadFavorites();
}, []);

const loadFavorites = () => {
    try {
        // Get token data
        const tokenData = getTokenData();
        // Get the right profile object
        const profile = tokenData.profiles.find(profile => profile.id == profileId);
        // Get the list of favorites
        const favoriteFilmIds = profile.favorites.map(item => item.film_id);
        // Set the favorite film ids
        setFavoriteFilmIds(favoriteFilmIds);
    }
```

This block invokes loadFavorites() once on mount, pulls the right profile from the decoded token, and stores its favorites IDs in state.

Design Pattern Used:
Adapter pattern: FilmList is reused here with isFilmBrowser={false} to adapt a generic list component for "favorites" display.

---

j. SearchResultsPage.js

Purpose:
Displays the list of films matching the user's most recent search terms.

Key Functionality:

- localStorage.getItem('searchResults'): Reads comma-separated film IDs from storage.
- <FilmList bannerDisplay='Search Results' filmIds={searchHistory} isFilmBrowser={false} />: Reuses the same film-list component for search output.
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the GradientBackground component for consistent background styling.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the Navbar component for organized look for searching.

Main Logic Snippet:

```
const loadSearchResults = () => {
    // if we have navigated to the page, then the search history exists
    const searchHistory = localStorage.getItem('searchResults');
    const searchHistoryIds = searchHistory.split(',').map(id => parseInt(id, 10));
    setSearchHistory(searchHistoryIds);
};
```

Here, loadSearchResults parses the stored IDs into an array of numbers and updates component state.

Design Pattern Used:
Iterator pattern: Iterates over the stored string of IDs to build the searchHistory array.

---

k. WatchHistoryPage.js

Purpose:
Shows all films a profile has previously watched.

Key Functionality:

- getTokenData(): Reads JWT to find this profile's watch_history.
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the GradientBackground component for consistent background styling.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the Navbar component for organized look for searching.

Main Logic Snippet:

```
useEffect(() => {
  loadWatchHistory();
}, []);

const loadWatchHistory = () => {
  // Get token data
  const tokenData = getTokenData();
  // Get the right profile object
  const profile = tokenData.profiles.find(profile => profile.id == profileId);
  // Get the watch history list
  const watchHistoryFilmIds = profile.watch_history.map(item => item.film_id);
  // Set the watch history film ids
  setWatchHistoryFilmIds(watchHistoryFilmIds);
}
```

On mount, extracts watch_history IDs from the token and stores them in state.

Design Pattern Used:
Adapter pattern: Reuses FilmList for "watch history" by passing appropriate filmIds.

---

l. WatchLaterPage.js

Purpose:
Lists films the profile has marked "Watch Later."

Key Functionality:

- getTokenData(): Reads the JWT payload to extract this profile's watch later.
- Displays a list of films marked for watch later
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the GradientBackground component for consistent background styling.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the Navbar component for organized look for searching.

Main Logic Snippet:

```
useEffect(() => {
    loadWatchLater();
}, []);

const loadWatchLater = () => {
    try {
        // Get token data
        const tokenData = getTokenData();
        // Get the right profile object
        const profile = tokenData.profiles.find(profile => profile.id == profileId);
        // Get the list of favorites
        const watchLaterFilmIds = profile.watch_later.map(item => item.film_id);
        // Set the favorite film ids
        setWatchLaterFilmIds(watchLaterFilmIds);
    }
```

Fetches watch_later IDs once, stores them for rendering.

Design Pattern Used:
Adapter pattern: FilmList again flexibly displays a different film subset.

6.4 Key Components

a. AuthenticationForm.js

Purpose: Provides an interface for both user login and registration based on the isLogin prop.

Key Functionality

- Handles user authentication by interacting with backend APIs.
- Validates password confirmation during registration.
- Toggles password visibility.
- Navigates to the profiles page upon successful authentication.

Main Logic Snippet

```
const authFunction = isLogin ? login : register;
const token = await authFunction(username, password);

console.log("Authentication successful!");

// Store the token
localStorage.setItem('authToken', token);

// Navigate to profiles page
navigate('/profiles');
```

This selects the appropriate authentication function based on the isLogin flag, executes it with the provided credentials, stores the received token in local storage, and navigates the user to the profiles page.

---

b. EditProfileForm.js

Purpose: Allows users to rename and update an existing profile.

Key Functionality

- Toggles visibility of the edit form.
- Validates token before submission.
- Updates the profile's display name.
- Refreshes the profile list upon successful update.

Main Logic Snippet

```
const newToken = await editProfile(displayName, newDisplayName);

// Success!
localStorage.setItem('authToken', newToken);

setDisplayName('');
setNewDisplayName('');
loadProfile();
```

This code calls the editProfile function with the current and new display names, updates the stored authentication token with the new one received, and refreshes the profile list to reflect the changes.

---

c. GradientBackground.js

Purpose: Serves as a wrapper component that applies a consistent gradient background to its child components, enhancing visual aesthetics.

Key Functionality

- Wraps child components with a styled div.
- Applies gradient styling through CSS.

Main Logic Snippet

```
const GradientBackground = ({ children }) => {
  return <div className="gradient-background">{children}</div>;
};
```

This returns a div with the class gradient-background, encapsulating any child components passed to it, thereby applying the gradient styling.

Design Patterns Used:
Decorator Pattern: Enhances the appearance of child components without modifying their behavior by wrapping them with additional styling.

---

d. ActionButton.js

Purpose: Provides a reusable button component with consistent styling and customizable properties for various actions across the application.

Key Functionality

- Renders a button with a customizable label, type, and ID.
- Handles click events through the onClick prop.
- Applies consistent styling via CSS classes.

Main Logic Snippet

```
const ActionButton = ({ label, onClick, type = 'button', id }) => {
    return (
      <button
        className="action-button"
        onClick={onClick}
        type={type}
        id={id}
      >
        {label}
      </button>
```

This code defines a button element that utilizes the action-button CSS class for styling, assigns the provided onClick handler, sets the button type, the ID, and displays the label.

---

e. LogoutButton.js

Purpose: Facilitates user logout by clearing session data and redirecting to the login page.

Key Functionality

- Clears authentication token from local storage.
- Redirects the user to the login page.
- Utilizes the ActionButton component for consistent styling.

Main Logic Snippet

```
<ActionButton label="Logout" id="logoutButton" onClick={() => {
    // remove info from session
    localStorage.clear();
    // navigate back to login
    navigate("/");
}} />
```

This renders an ActionButton labeled "Logout" that, when clicked, clears all data from local storage and navigates the user back to the root path, effectively logging them out.

---

f. FilmList.js

Purpose: Displays a list of films based on provided film IDs, allowing users to watch films, add them to a watch later list, or mark them as favorites.

Key Functionality

- Filters and displays films from local storage based on provided IDs.
- Provides interactive buttons for each film to perform various actions.
- Navigates to the film watching page upon user interaction.

Main Logic Snippet

```
// Filter films to only include those with IDs in the filmIds prop
const filteredFilms = films.filter(film => filmIds.includes(film.id));

// Set the filtered list of complete film objects
setFilmList(filteredFilms);
```

This code filters the list of films retrieved from local storage to include only those whose IDs match the provided filmIds array, then updates the component's state with this filtered list for rendering.

Design Patterns Used

- **Iterator Pattern**: Iterates over the array of films to filter and display the relevant ones based on user interaction.

---

g. FilmSearchForm.js

Purpose:
Provides the in-navbar search box, dropdown history, and submits new queries.

Key Functionality:

- getTokenData(): Retrieves past search strings from token.
- Toggles dropdown of past queries.
- Calls back-end search endpoint.
- Stores searchtoken & results in localStorage, then navigate('/searchresults/…').
- Utilizes the FilmList component to provide the user with a list of films to watch and enjoy.
- Utilizes the GradientBackground component for consistent background styling.
- Utilizes the ActionButton component for consistent button styling.
- Utilizes the Navbar component for organized look for searching.

Main Logic Snippet:

```
const handleSubmit = async (e) => {
    e.preventDefault();
    try {
        // get searchtoken from api
        const searchResponse = await search(profileId, searchText);
        // get the result from the searchResponse
        localStorage.setItem('searchtoken', searchResponse);
        // get token from memory to make sure proper data is used
        let searchToken = JSON.parse(localStorage.getItem('searchtoken'));

        // set the memory with proper values
        localStorage.setItem('authToken', '"' + searchToken.token + '"');
        localStorage.setItem('searchResults', searchToken.results);

        // update the actual component
        loadSearchHistory();

        // remove the text from the search
        setSearchText('');

        // if there are results navigate to the page and show the results
        if (localStorage.getItem('searchResults') != '') {
            navigate(`/searchresults/${profileId}`);
        }
```

On form submit: calls search(), stores returned IDs, and navigates to results.

Design Pattern Used:
Command pattern: Encapsulates the "search" action and its side-effects (storage, navigation) in one handler.

---

h. Navbar.js

Purpose:
Renders the top navigation bar, optionally including the search form.

Key Functionality:

- Displays the application name.
- Displays buttons to different pages and the search bar.
- Utilizes the Navbar component for organized look for searching.

Main Logic Snippet:

```jsx
const Navbar = ({ profileId, hasSearch }) => {
  return (
    <nav className="navbar">
      <div className="navbar-brand">
        <h1>Minflix</h1>
      </div>
      {hasSearch && (
        <div className="navbar-search">
          <FilmSearchForm profileId={profileId} />
        </div>
      )}
    </nav>
  );
};
```

Conditionally injects the search form based on the hasSearch prop.

Design Pattern Used:
Decorator pattern: Wraps optional search functionality around the base nav bar.

---

5. Styling

a. GradientBackground.css

Purpose: Applies a consistent blue-to-black gradient background across pages where it's used, creating visual consistency.

Key Functionality:

- Establishes a full-screen background layout.
- Provides flexible padding and text alignment for content inside the gradient.
- Ensures all child elements use box-sizing: border-box for better layout control.

Main Logic Snippet:

```
.gradient-background,
.gradient-background * {
  box-sizing: border-box;
}

.gradient-background {
    display: flex;
    flex-direction: column;
    min-height: 100vh;
    overflow: hidden;
    background: linear-gradient(to bottom,  blue, #000000);
    color: white;
    text-align: center;
  }
```

This sets a column layout with a vertical gradient and prevents scrollbars from appearing unexpectedly. The * selector ensures all elements inside follow consistent sizing behavior.

---

b. ActionButton.css

Purpose: Provides styling for all reusable buttons that share the same shape, size, and hover behavior.

Key Functionality:

- Gives a consistent blue look with rounded corners.
- Adds a transition effect on hover.
- Inherits box sizing for layout stability.

Main Logic Snippet:

```css
.action-button,
.action-button * {
  box-sizing: border-box;
}
/*Styling of buttons*/
.action-button {
    background-color: #1e3a5f;
    border: 2px solid #00bfff;
    color: white;
    border-radius: 10px;
    padding: 10px 20px;
    margin: 10px;
    font-size: 1rem;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

.action-button:hover {
    background-color: #1e90ff;
}
```

Creates visually appealing, accessible buttons that are used throughout the project in components like LogoutButton, EditProfileForm, and FilmList.

---

c. ProfilePickerPage.css

Purpose: Handles layout and visual styling specific to the Profile Picker page, including the profile buttons, layout spacing, and form styling.

Key Functionality:

- Styles circular profile buttons.
- Centers the dashboard content and buttons.
- Styles profile forms with contrast borders and inputs.

Main Logic Snippet:

```
#overlay #profileButton {
    border: 2px solid #00bfff;
    border-radius: 50%;
    width: 120px;
    height: 120px;
    background: transparent;
    color: white;
    font-size: 16px;
    cursor: pointer;
    transition: transform 0.2s, background-color 0.3s;
}
```

This is what gives the aesthetic elements like profile circles and balanced forms, giving them a unique look compared to the ActionButton.

---

d. LoginPage.css

Purpose: Enhances the user experience during login by adding visual depth, central alignment, and responsive form controls.

Key Functionality:

- Styles input fields with rounded corners and transitions.
- Controls visibility of password inputs and checkboxes.

Main Logic Snippet:

```css
.wrapper {
    width : 420px;
    background: transparent;
    border: 2px solid rgba(255, 255, 255, 0.2);
    color: white;
    border-radius: 10px;
    padding: 30px 20px;
    backdrop-filter: blur(40px);
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.2);
    position: center;
    margin: 0 auto;
    position: relative;
}

*{
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    font-family: 'Poppins', sans-serif;
}

.input-box{
    width: 100%;
    height: 50px;
    margin: 30px 0;

}
```

The .wrapper and .input-box classes offer a clean form layout with rounded inputs, subtle borders, and a centered layout.

---

e. ProfileHomePage.css

Purpose: Styles the header and navigation buttons on the Profile Home Page, offering clarity and consistent spacing.

Key Functionality:

- Enlarges and centers the page title text.

- Adds margin spacing to navigation buttons for visual separation.

Main Logic Snippet:

```
.profile-home-page{
    font-size: 50px;
    color: #FFF;
    text-align: center;

}

.back-to-profile{
    margin: 10px;
    font-size: 1rem;
}
```

The .profile-home-page class is used for the main heading text, while .back-to-profile ensures buttons below are spaced appropriately and maintain consistent sizing.

---

f. FilmBrowserPage.css

Purpose:
Applies layout and responsive sizing to film thumbnails when browsing lists of films.

Key Functionality:

- Displays film images with fixed sizing.
- Images are aligned towards the center.

Main Logic Snippet:

```css
.thumbnail{
    max-width: 100%; /* Image scales within its container */
    height: auto; /* Maintain aspect ratio */
}

.content {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    padding: 35px;
}
```

These rules ensure each film's poster (.thumbnail) never overflows its container, and .content centers all inner elements with comfortable padding.

---

g. FilmSearchForm.css

Purpose:
Styles the search input, dropdown history, and search button within the search form component.

Key Functionality:

- Made sure that the container for the form was fixed and looked nice.
- Styled the input for good aesthetics.
- Added a drop down menu to display a professional and modern search bar.

Main Snippet:

```css
.search-container {
    position: relative;
    width: 100%;
}

.search-form {
    width: 100%;
}

.search-input-container {
    display: flex;
    width: 100%;
}

.search-input {
    flex-grow: 1;
    padding: 0.5rem 1rem;
    border: none;
    border-radius: 10px;
    /*border-radius: 4px 0 0 4px;*/
    font-size: 0.9rem;
}

.search-button {
    border-radius: 0 4px 4px 0;
    margin-left: 0;
}

/* Dropdown styles */
.dropdown-container {
    position: absolute;
    width: 64%;
    z-index: 10;
```

This flex layout keeps the text field and button aligned; the dropdown sits on top when open, styled as a white card with shadow.

---

h. FilmWatcher.css

Purpose:
Defines the full-screen video player container, gradient background, and responsive behavior for the film-watching page.

Key Functionality:

- Displays the movie watcher in the middle of the screen for a more symmetrical look.
- Adapted for mobile users.

Main Logic Snippet:

```css
#filmWatcher {
  width: 100%;
  max-width: 1200px;
  text-align: center;
}

#filmWatcher h2 {
  font-size: 2rem;
  margin-bottom: 20px;
}

#filmWatcher video {
  width: 100%;
  height: auto;
  border-radius: 10px;
}


/* Responsive for screens 600px and below */
@media (max-width: 600px) {
  #filmWatcher h2 {
    font-size: 1.5rem;
  }

  #filmWatcher video {
    max-width: 100%;
  }
}
```

Ensures the video scales fluidly on both desktop and small screens, with a responsive heading size under 600 px width.

---

i. Navbar.css

Purpose:
Styles the top navigation bar—its layout, colors, and sticky positioning.

Key Functionality:

- The navbar is placed on top and given a very simple look, but very distinct so users know where to search movies.

Main Logic Snippet:

```css
.navbar {
    display: flex;
    align-items: center;
    justify-content: space-between;
    padding: 1rem 2rem;
    background-color: #1e3a5f;
    border-radius: 10px;
    border: 2px solid #00bfff;
    color: white;
    position: sticky;
    top: 0;
    z-index: 100;
    width: 100%;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

.navbar-brand h1 {
    margin: 0;
    font-size: 1.5rem;
}
```

Creates a two-column flex layout, fixes the bar to the top of the viewport, and uses your dark-blue and cyan accent colors for consistent branding.

6.6 Backend Main and Core

a. main.py

Purpose: This is the central file of the backend. It sets up the FastAPI app, initializes the database, configures CORS, and defines the main endpoints for user authentication, profile management, and film handling.

Key Functionality:

- Registers and logs in users
- Adds/edits user profiles
- Adds favorites and watch-later films
- Streams films and serves image files
- Configures initial DB with either example or production data

Main Logic Snippet:

```python
@app.post("/login")
async def login(session: SessionDep, form_data: OAuth2PasswordRequestForm = Depends()) -> str:
    statement = select(FilmUser).where(FilmUser.username == form_data.username)
    current_user = session.exec(statement).first()

    if current_user is None:
        raise HTTPException(status_code=404, detail="User not found")

    if not settings.pwd_context.verify(form_data.password, current_user.password):
        raise HTTPException(status_code=404, detail="Wrong Password")

    return create_jwt_token(TokenModel.model_validate(current_user).model_dump())
```

This snippet authenticates a user by checking their username and password, then generates a JWT token if successful.

Design Pattern Used:

- **Command Pattern**: User actions like login and registration are abstracted as endpoint "commands" handled by FastAPI.

- **Decorator Pattern**: FastAPI uses decorators like @app.post() and @Depends() to wrap core logic around HTTP methods and dependency injection.

---

b. config.py

Purpose: Holds all configuration and environmental settings, like database connection strings, secrets, and FastAPI security setup.

Key Functionality:

- Reads environment variables for configuration.
- Stores reusable security objects (e.g., pwd_context, oauth2_scheme).
- Defines static media and image directories.

Main Logic Snippet:

```
class Settings(BaseSettings):
    db_setup: str = os.getenv("DATABASE_SETUP", "Dynamic")
    db_url: str = os.getenv(
```

This class defines application-wide configuration variables using Pydantic and environment variables.

---

c. db.py

Purpose: Manages database engine setup, session creation, and utility functions for setting up tables and loading test data.

Key Functionality:

- Establishes SQLModel engine.
- Provides DB session dependencies.
- Drops, creates, and seeds the database.

Main Logic Snippet:

```python
def add_films(session: SessionDep):
    for film in FILMS:
        session.add(film)
    session.commit()
```

Once tables are initialized, add_films(session) populates the database with production film data from film_data.py, ensuring the app has real content for browsing and streaming.

Design Patterns Used:

- **Command Pattern** — Each method encapsulates a specific DB setup command.

---

d. jwt.py

Purpose: Handles creation, decoding, and validation of JSON Web Tokens (JWTs) used for authentication.

Key Functionality:

- Generates tokens using jose.
- Verifies token integrity and expiration.
- Extracts user identity for secure endpoints.

Main Logic Snippet:

```python
def create_jwt_token(data: dict) -> str:
    to_encode = data.copy()
    expire = datetime.datetime.now(
        datetime.timezone.utc) + datetime.timedelta(minutes=settings.access_token_expire_minutes)
    to_encode.update({"exp": expire, "token_type": "bearer"})
    encoded_jwt = jwt.encode(to_encode, settings.secret_key, algorithm=settings.algorithm)
    return encoded_jwt
```

Encodes a payload with an expiration timestamp into a JWT, using a secret key and algorithm.

---

e. log.py

Purpose:
Configures logging for the application and reduces SQLModel noise.

Key Functionality:

- Logs app events into a file.
- Sets SQLModel logging level to WARNING.

Main Logic Snippet:

```
logging.basicConfig(
    filename='app.log',
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  # Log format
)
```

Initializes logging configuration to output timestamped logs into app.log.

6.7 Data

a. film_data.py

Purpose: Stores a predefined list of classic public-domain films with metadata for seeding the database.

Key Functionality:

- Defines FILMS, a list of Film objects.
- Each film includes title, image, filename, and metadata.

Main Logic Snippet:

```
FILMS = [
    Film(
        title="Assignment: Outer space",
        length=1,
        image_name="Assignment_Outer_Space.jpg",
        file_name="Assignment_Outer_Space.mp4",
        producer="NA",
        film_cast=[],
        production_team=[]
    ),
```

Declares a hardcoded list of film objects for import into the app database.

6.8 Models

a. film_models.py

Purpose: This file defines the data structure for films, including cast and production team members. These models directly map to database tables using SQLModel.

Key Functionality:

- Film class stores metadata about each film (title, length, image, etc.)
- FilmCast links actors to a film
- FilmProductionTeam links behind-the-scenes members to a film
- All models support relationship mapping via SQLModel

Main Logic Snippet:

```python
class Film(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    title: str
    length: int
    image_name: str
    file_name: str
    producer: str
    name: Optional[str] = None  # Added for backward compatibility

    # Relationships
    film_cast: List["FilmCast"] = Relationship(back_populates="film")
    production_team: List["FilmProductionTeam"] = Relationship(back_populates="film")
```

This code creates a Film table that stores film metadata and also defines relationships to cast and production teams.

---

b. film_token_models.py

Purpose: Defines a lightweight representation of film data used for sending to the frontend as part of a JWT or API response.

Key Functionality

- FilmToken class is a minimal BaseModel containing only public-facing fields of a film.

Main Logic Snippet

```python
class FilmToken(BaseModel):
    id: int
    title: str
    image_name: str
    file_name: str
```

This class allows only the necessary film data to be passed along to clients without exposing relationships or internal attributes.

---

c. token_models.py

Purpose: Models to help encode user profiles, favorites, and watch lists into the JWT token. These are simplified representations designed to serialize database models cleanly into tokens.

Key Functionality:

- TokenModel: Represents a user and their profiles.
- TokenProfileDataModel: Represents a user's profile with watch lists and favorites.
- Lightweight token-safe versions of actual database models.

Main Logic Snippet:

```
class TokenModel(BaseModel):
    '''
        id: this the film user id
        profiles: list of user profiles
    '''
    id: int
    profiles: List["TokenProfileDataModel"]
    model_config = ConfigDict(from_attributes=True)
```

This snippet defines the outermost structure of the JWT payload, encapsulating user ID and all profile info.

---

d. user_models.py

Purpose: Defines all user-related database models including users, profiles, and subcomponents like favorites and watch history.

Key Functionality

- FilmUser: Authenticated user with login credentials and profile list.
- Profile: Each user's customizable profile.

- Favorite, WatchLater, SearchHistory, and WatchHistory: support film. interaction features.
- All models are linked via relationships.

Main Logic Snippet

```python
class FilmUser(SQLModel, table=True):
    # Define the table name explicitly
    __tablename__ = "filmuser"

    id: Optional[int] = Field(default=None, primary_key=True)

    # Use SQLAlchemy Column for explicit naming
    # If your database column is actually named differently, specify it here
    # Common possibilities might be 'user_name', 'email', etc.
    username: str = Field(sa_column=Column("email", String, nullable=False))
    password: str
    date_registered: datetime.datetime
    profiles: List["Profile"] = Relationship(back_populates="filmuser")
```

This creates a filmuser table that stores user credentials and maintains a list of profiles. The username is explicitly stored under a column named "email".

---

e. search_models.py

Purpose:
Defines the shape of the JSON response returned by the search endpoint, bundling both the list of matching film IDs and the JWT search token.

Key Functionality:

- Inherits from Pydantic's BaseModel to enforce type checking and automatic serialization.
- results: List[int] — array of film IDs matching the search query.
- token: str — JWT string containing both the search results and updated authentication.

Main Logic Snippet:

```python
class SearchResponseModel(BaseModel):
    results: List[int]
    token: str
```

Pydantic reads these annotations, ensures both fields are present and correctly typed, and produces the JSON schema for FastAPI's docs.

6.9 Recommender

a. recommender.py

Purpose:
Provides a simple content-based film recommendation service using cosine similarity over genre feature vectors.

Key Functionality:

- **cosine_similarity(vec_a, vec_b)** — computes similarity between two NumPy vectors.
- Builds a **similarity_matrix** at module load time for all films.
- **recommend(film_name, top_n=2)** — returns the top_n most similar films to the given title.

Main Logic Snippet:

```python
def recommend(film_name, top_n=2):
    """
        Recommend a film using the matrix and cosine similarity function
    """
    film_index = film_names.index(film_name)
    similarity_scores = similarity_matrix[film_index]
    similar_films = sorted(zip(film_names, similarity_scores),
                           key=lambda x: x[1], reverse=True)
    result = [film for film, score in similar_films[1:top_n+1]]
    return result
```

It looks up the index of the target film, retrieves its precomputed similarity scores, sorts all films by descending similarity, and returns the next top_n titles.

---

7. Testing

7.1 Testing Approach

Our testing methodology focuses on verifying the correctness and reliability of both frontend and backend components through unit tests, user interaction tests, and use case validation. The primary goals are to:

- Confirm that all critical features behave as expected.
- Ensure full C0 (line) coverage across core functional components.
- Identify regressions early in development.

a. Frontend Testing

We use **Jest** and **React Testing Library** to test user interface components. These tests simulate user interactions such as button clicks and form submissions, verifying that UI elements render correctly and event handlers behave as expected.

Example testing strategies:

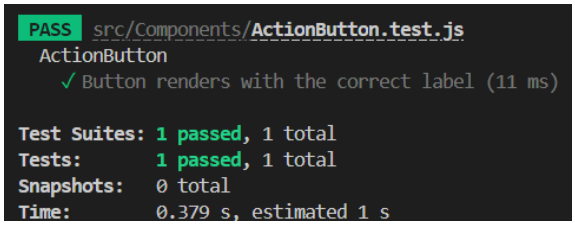
- Verifying component rendering with correct props.
- Testing form validation and navigation logic.
- Ensuring conditional rendering (e.g., showing/hiding elements) works correctly.

Testing snippet:

```
describe(ActionButton, () => {
    // test 1: tests for button rendering with correct label "Click Me"
    it("Button renders with the correct label", () => {
        const {getByText} = render(<ActionButton label="Click Me" onClick={() => {}} />);
        const buttonElement = getByText("Click Me");
        expect(buttonElement).toBeInTheDocument();
    });
```

This test suite for the ActionButton component renders the button with a label prop of "Click Me" (and a no-op onClick handler), then uses **React Testing Library's** getByText to locate an element displaying that exact label. Finally, it asserts that the

button is present in the document, verifying that the component correctly renders its label.



This shows that 1 out of 1 tests have passed.

C0 coverage is prioritized—every function, branch, and conditional in our frontend logic is executed at least once in our tests.

---

b. Backend Testing

The backend is tested using **Pytest** alongside **FastAPI's TestClient**. These tools allow us to simulate real HTTP requests to API endpoints in a controlled environment. The goal is to validate authentication, data persistence, and endpoint correctness.

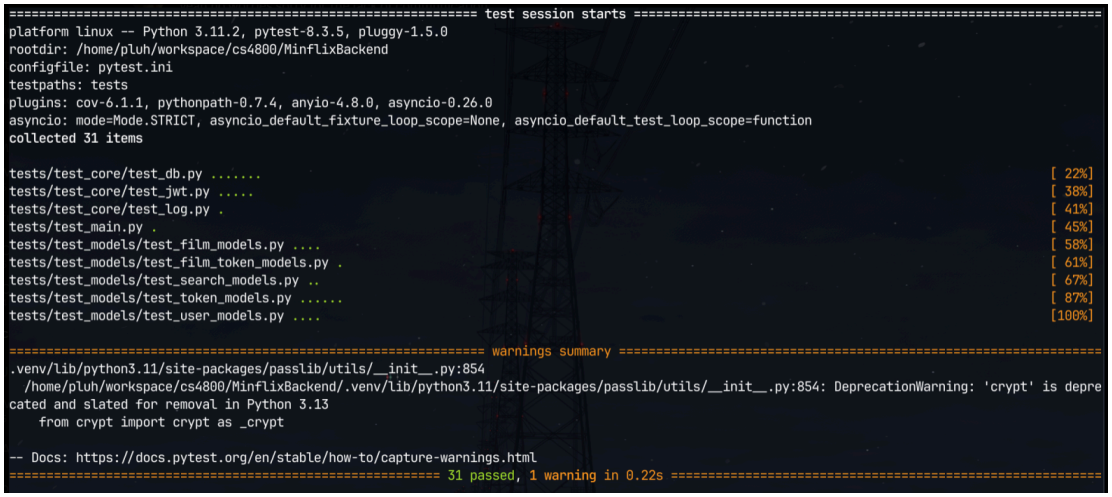
Backend testing includes:

- Registration and login flows.
- Profile creation and editing.
- Watch later, favorites, and watch history management.
- Error handling for invalid input or unauthorized access.

Testing Snippet:

```python
@pytest.fixture
def jwt_test_data():
    """Fixture providing test data for JWT tests."""
    return {
        "id": 1,
        "username": "test@example.com",
        "profiles": [{"id": 1, "displayname": "Test Profile"}]
    }
```

This Pytest fixture generates sample user data to simulate a decoded JWT token. It allows backend tests to inject consistent test identities without relying on real token generation. By using this, we can easily validate route behavior in isolation and simulate scenarios such as profile-based access control, token validation, and user-specific actions.



This shows that 31 tests have passed.

The tests assert expected status codes, token behavior, and database changes, helping us validate both positive and negative cases.

7.2 Coverage

Testing coverage includes:

- **Frontend logic** (via Jest): ActionButton, AuthenticationForm, GradientBackground, EditProfileForm, FilmList, FilmSearchForm, LogoutButton, Navbar, and all page-level components like LoginPage, RegistrationPage, ProfilePickerPage, FavoritePage, FilmBrowserPage, FilmSearchPage, FilmWatcher, ProfileHomePage, SearchResultsPage, WatchHistoryPage, WatchLaterPage.

- **Backend logic** (via Pytest): All routes in main.py including /login, /registration, /addprofile, /editprofile, /watchlater, /favorite, /getfilms, /films, /images, and /recommendations.

We focus primarily on **C0 (line) coverage**, ensuring that all key branches and logic paths are exercised at least once.

We track frontend coverage using Jest and backend coverage using Pytest.

7.3 Conclusion

Our team ensured that each component was tested independently and validated against real-world use cases. The combination of **Jest** for frontend and **Pytest + FastAPI TestClient** for backend ensures that both interfaces and internal APIs are robust, secure, and user-ready. We are confident in the accuracy and completeness of our test coverage across the entire stack.

---

8. Conclusion

8.1 Difficulties

Given the scale of the project, we were forced to learn many new things and use tools that we never used before. Some people were a lot more experienced, but there were some people who needed more time to learn. We all did our best to work around our strengths and weaknesses, but we still had hurdles to jump over. We wished to have implemented things, but due to time we just weren't able to. For example, we have many great styling ideas and proofs of concepts, but unfortunately opted to cut them to have a presentable application that was optimized.

8.2 Future Suggestions

As stated before, we would want to add and improve on things such as styling. We have the proof of concept, but dropped due to time. Also things like adding more films to our catalog would be a nice addition, but be the last things on our list. We here at MinSystems value performance and optimization to our application. Hopefully in the future we are able to create improvements in things like style or

movie selection when we feel confident on doing it in a way that doesn't hinder user experience.

8.3 Closing Statement

That being said, we still worked well and implemented many of the clients desires throughout these following months. We had a working application that can have multiple users enjoy our catalog of films. The experience was very educational and helpful for all members here at MinSystems. We will use this experience to improve our current application and any other future projects that we take on; striving to increase user enjoyment and experience.

---

9. Appendices

9.1 Use Case Testing

We created a use case test table to validate real user flows on the deployed app. Each row lists the interface, the input scenario, expected outcome, actual result, and pass/fail status.

Loc	Use cases	Expected	Actual	Success/fail
Login Page	"User enters an email and password that is not registered yet."	"Authentication failed: User not found"	"Authentication failed: User not found"	Success
Login Page	"User presses the register button"	Redirects the user to the registration page.	Redirects the user to the registration page.	Success
Register Page	"User tries to register with a blank text box."	"Please fill this field out."	"Please fill this field out."	Success
Register Page	"User writes two different passwords for the password and the confirm	"Passwords do not match."	"Passwords do not match."	Success

	password text fields.”			
Register Page	“User enters their information correctly on the email, password, and confirm password text fields”	The user is redirected to the profile picker page.	The user is redirected to the profile picker page	Success
Login Page	“User enters their information correctly on the email, and password text fields”	The user is redirected to the profile picker page.	The user is redirected to the profile picker page	Success
Login/Register Page	“User presses the show password box to unhash the password”	Shows password.	Shows password.	Success
Register Page	“User presses the login button”	Redirects the user to the login page.	Redirects user to the login page.	Success
Register Page	“User adds an email that is already registered”	“Authentication failed: User already exists. Please login instead.”	“Authentication failed: User already exists. Please login instead.”	Success
Login Page	“User adds the right email, but wrong password”	“Authentication failed: Wrong Password”	“Authentication failed: Wrong Password”	Success
Profile Picker Page	“User enter the profile picker page without having profiles.”	It will state that there are currently no profiles.	It will state that there are currently no profiles.	Success
Profile Picker Page	“User enter the profile picker page with having profiles.”	Profiles will display to the user.	Profiles will display to the user.	Success
Profile Picker Page	“User presses the 'Add Profile’ button.”	A form will open prompting the user to add a display name.	A form will open prompting the user to add a display name.	Success
Profile Picker Page	“User presses	“Please fill this field	“Please fill this field	Success

	submit on the ‘Add Profile’ text field with no display name.”	out”	out”	
Profile Picker Page	“User presses submit on the ‘Add Profile’ text field with a display name.”	A profile is created with a button associated with its name	A profile is created with a button associated with its name	Success
Profile Picker Page	“User presses close on the ‘Add Profile’ text field with a display name.”	The form closes.	The form closes.	Success
Profile Picker Page	“User presses the 'Edit Profile’ button.”	A form will open prompting the user to edit the display name.	A form will open prompting the user to edit the display name.	Success
Profile Picker Page	“User presses submit on the ‘Edit Profile’ with no display name or new display name.”	“Please fill this field out”	“Please fill this field out”	Success
Profile Picker Page	“User presses submit on the ‘Edit Profile’ with a display name that doesn’t exist.”	No change happens	No change happens	Success
Profile Picker Page	“User presses submit on the ‘Edit Profile’ with a display name and a new display name.”	The old profile’s name is edited and it is displayed on the profile button.	The old profile’s name is edited and it is displayed on the profile button.	Success
Profile Picker Page	“User presses the logout button”	The user is redirected to the login page.	The user is redirected to the login page.	Success
Profile Picker Page	“User presses a profile button”	User is redirected to the profile home page.	User is redirected to the profile home page.	Success
Profile Home Page	“User enters the home page without ever watching a movie.”	There will be no recommendations available.	There will be no recommendations available.	Success

Profile Home Page	“User enters the home page and has watched a movie before.”	Films will be displayed in the “Recommended For You” section of the homescreen.	Films will be displayed in the “Recommended For You” section of the homescreen.	Success
Profile Home Page	“User enters the home page after watching a film from the browse films section for the first time.”	Films will be displayed in the “Recommended For You” section of the homescreen.	Films will be displayed in the “Recommended For You” section of the homescreen.	Success
Profile Home Page	“User presses Browse film button”	User is redirected to the Browse film page.	User is redirected to the Browse film page.	Success
Profile Home Page	“User presses the favorites button and hasn’t favorite movies.”	Users are shown there are no favorite films to display.	Users are shown there are no favorite films to display.	Success
Favorites Page	“User presses the return to profile button.”	Users are redirected back to Profile Home Page.	Users are redirected back to Profile Home Page.	Success
Profile Home Page	“User presses the watch later tab and hasn’t movies to watch later.”	Users are shown that there are no films to watch later.	Users are shown that there are no films to watch later.	Success
Watch Later Page	“User presses the return to profile button.”	Users are redirected back to Profile Home Page.	Users are redirected back to Profile Home Page.	Success
Profile Home Page	“User presses the history tab and hasn’t watched movies before.”	Users are shown that there are no films to display in history.	Users are shown that there are no films to display in history.	Success
Watch History Page	“User presses the return to profile button.”	Users are redirected back to Profile Home Page.	Users are redirected back to Profile Home Page.	Success
Profile Home Page	“User presses the favorites button and has favorite movies.”	Users are shown all their liked films in the favorites tab.	Users are shown all their liked films in the favorites tab.	Success
Profile Home Page	“User presses the watch later button	Users are shown all the films they	Users are shown all the films they	Success

	and has movies to watch later.”	marked to watch later.	marked to watch later.	
Profile Home Page	“User presses the history button and has watched movies before.”	Users are shown the history of all the films they have watched.	Users are shown the history of all the films they have watched.	Success
Profile Home Page	“User presses the back to profile button”	Users are redirected back to profile picking page.	Users are redirected back to profile picking page.	Success
Profile Home Page	“User presses the 'Edit Profile’ button.”	A form will open prompting the user to edit the display name.	A form will open prompting the user to edit the display name.	Success
Profile Home Page	“User presses submit on the ‘Edit Profile’ with no display name or new display name.”	“Please fill this field out”	“Please fill this field out”	Success
Profile Home Page	“User presses submit on the ‘Edit Profile’ with a display name that doesn’t exist.”	No change happens	No change happens	Success
Profile Home Page	“User presses submit on the ‘Edit Profile’ with a display name and a new display name.”	The old profile’s name is edited and it is displayed on the profile button.	The old profile’s name is edited and it is displayed on the profile button.	Success
Profile Home Page	“User presses close on the ‘Edit Profile’ button”	The form closes	Thes form closes	Success
Profile Home Page	“User presses the Search button with no entry”	It states “Please fill this field out”	It states “Please fill this field out”	Success
Profile Home Page	“User presses the Search button with a single letter”	It redirects the user to the search page with any movie with that letter.	It redirects the user to the search page with any movie with that letter.	Success
Search Result Page	“User’s search results after	All movies with that letter are shown.	All movies with that letter are shown.	Success

	searching with a single letter”			
Search Result Page	“User presses the return to profile button.”	Users are redirected back to Profile Home Page.	Users are redirected back to Profile Home Page.	Success
Profile Home Page	“User presses the Search button with a name of a movie”	It redirects the user to the search page with the film they search for.	It redirects the user to the search page with the film they search for.	Success
Search Results Page	“User’s search results after searching with the name of a movie”	The movie of that name is shown.	The movie of that name is shown.	Success
Profile Home Page	“User presses the Search button with a random assortment of letters”	The user is not redirected anywhere.	The user is not redirected anywhere.	Success
Profile Home Page	“User presses the Search bar with no previous searches.”	There is no drop down that shows previous search history.	There is no drop down that shows previous search history.	Success
Profile Home Page	“User presses the Search bar with previous searches.”	There is a drop down that shows previous search history.	There is a drop down that shows previous search history.	Success
Profile Home Page	“User presses the Search bar with more than 3 previous searches.”	There is a drop down that shows 3 most recent previous searches.	There is a drop down that shows 3 most recent previous searches.	Success
Film Browse Page	“User presses on the movie button.”	User is redirected to the film watching page to watch the movie.	User is redirected to the film watching page to watch the movie.	Success
Film Brose Page	“User presses on the favorite button”	The movie will be found in the favorite tab.	The movie will be found in the favorite tab.	Success
Film Browse Page	“User presses on the watch later button”	The movie will be found in the watch later tab.	The movie will be found in the watch later tab.	Success

Film Browse Page	“User presses the back to profile button”	Redirects user back to the profile home page	Redirects user back to the profile home page	Success
Film Watcher	“User presses the full screen button while the film isn’t playing”	The film becomes full screen without playing the film.	The film becomes full screen without playing the film.	Success
Film Watcher	“User presses the windowed button while the screen button while the film isn’t playing”	The film exits full screen and the film continues to not be playing.	The film exits full screen and the film continues to not be playing.	Success
Film Watcher	“User presses the play button while not full screened”	The film plays while not full screen.	The film plays while not full screen.	Success
Film Watcher	“User presses the pause button while not full screened”	The film pauses while not full screen.	The film pauses while not full screen.	Success
Film Watcher	“User presses the screen to play the film while not full screened”	The film plays while not full screen.	The film plays while not full screen.	Success
Film Watcher	“User presses the screen to pause the film while not full screened”	The film pauses while not full screen.	The film pauses while not full screen.	Success
Film Watcher	“User presses the play button while full screened”	The film plays while full screen.	The film plays while full screen.	Success
Film Watcher	“User presses the pause button while full screened”	The film pauses while full screen.	The film pauses while full screen.	Success
Film Watcher	“User presses the screen to play the film while full screened”	The film plays while full screen.	The film plays while full screen.	Success
Film Watcher	“User presses the full screen button while the film is playing”	The film becomes full screen and continues to play the film.	The film becomes full screen and continues to play the film.	Success

Film Watcher	“User presses the windowed button while the screen button while the film is playing”	The film exits full screen and the film continues to play.	The film exits full screen and the film continues to play.	Success
Film Watcher	“User presses the 3 dotted bar while not full screened”	The download, playback speed, and picture in picture option popup.	The download, playback speed, and picture in picture option popup.	Success
Film Watcher	“User presses the 3 dotted bar while full screened”	The download, playback speed, and picture in picture option popup.	The download, playback speed, and picture in picture option popup.	Success
Film Watcher	“User presses the download option while not full screened”	The film gets downloaded in the user's system.	The film gets downloaded in the user's system.	Success
Film Watcher	“User presses the download option while full screened”	The film gets downloaded in the user's system.	The film gets downloaded in the user's system.	Success
Film Watcher	“User presses the playback speed option while not full screened”	The user can see speeds 0.25 to 2.0. The user can also press the back option if they don’t wish to see the speed options.	The user can see speeds 0.25 to 2.0. The user can also press the back option if they don’t wish to see the speed options.	Success
Film Watcher	“User presses the playback speed option while full screened”	The user can see speeds 0.25 to 2.0. The user can also press the back option if they don’t wish to see the speed options.	The user can see speeds 0.25 to 2.0. The user can also press the back option if they don’t wish to see the speed options.	Success
Film Watcher	“User presses the the back ‘<- options’ option of the playback speed while noy full screened”	The user goes back to seeing the download, playback speed, and Picture in picture option.	The user goes back to seeing the download, playback speed, and Picture in picture option.	Success
Film Watcher	“User presses the	The user goes back	The user goes	Success

	the back '<- options' option of the playback speed while full screened"	to seeing the download, playback speed, and Picture in picture option.	back to seeing the download, playback speed, and Picture in picture option.	
Film Watcher	"User presses 0.25 speed in playback speed option while not full screened"	The film moves at 0.25 speed.	The film moves at 0.25 speed.	Success
Film Watcher	"User presses 0.25 speed in playback speed option while full screened"	The film moves at 0.25 speed.	The film moves at 0.25 speed.	Success
Film Watcher	"User presses 0.60 speed in playback speed option while not full screened"	The film moves at 0.50 speed.	The film moves at 0.50 speed.	Success
Film Watcher	"User presses 0.50 speed in playback speed option while full screened"	The film moves at 0.50 speed.	The film moves at 0.50 speed.	Success
Film Watcher	"User presses 0.75 speed in playback speed option while not full screened"	The film moves at 0.75 speed.	The film moves at 0.75 speed.	Success
Film Watcher	"User presses 0.75 speed in playback speed option while full screened"	The film moves at 0.75 speed.	The film moves at 0.75 speed.	Success
Film Watcher	"User presses normal speed in playback speed option while not full screened"	The film moves at normal speed.	The film moves at normal speed.	Success
Film Watcher	"User presses normal speed in playback speed option while full screened"	The film moves at normal speed.	The film moves at normal speed.	Success
Film Watcher	"User presses 1.25 speed in playback speed option while	The film moves at 1.25 speed.	The film moves at 1.25 speed.	Success

	not full screened”			
Film Watcher	“User presses 1.25 speed in playback speed option while full screened”	The film moves at 1.25 speed.	The film moves at 1.25 speed.	Success
Film Watcher	“User presses 1.50 speed in playback speed option while not full screened”	The film moves at 1.50 speed.	The film moves at 1.50 speed.	Success
Film Watcher	“User presses 1.50 speed in playback speed option while full screened”	The film moves at 1.50 speed.	The film moves at 1.50 speed.	Success
Film Watcher	“User presses 1.75 speed in playback speed option while not full screened”	The film moves at 1.75 speed.	The film moves at 1.75 speed.	Success
Film Watcher	“User presses 1.75 speed in playback speed option while full screened”	The film moves at 1.75 speed.	The film moves at 1.75 speed.	Success
Film Watcher	“User presses 2 speed in playback speed option while not full screened”	The film moves at 2 speed.	The film moves at 2 speed.	Success
Film Watcher	“User presses 2 speed in playback speed option while full screened”	The film moves at 2 speed.	The film moves at 2 speed.	Success
Film Watcher	“User presses ‘Picture in picture’ option while not full screened”	The film now shows a miniature player on the corner of the screen.	The film now shows a miniature player on the corner of the screen.	Success
Film Watcher	“User presses ‘Picture in picture’ option while full screened”	The film now shows a miniature player on the corner of the screen.	The film now shows a miniature player on the corner of the screen.	Success
Film Watcher	“User presses play while in mini player”	The film plays in the mini player.	The film plays in the mini player.	Success

Film Watcher	“User presses pause while in mini player”	The film pauses in the mini player.	The film pauses in the mini player.	Success
Film Watcher	“User presses ‘x’ while in mini player”	The film pauses in the mini player and exits the mini player	The film pauses in the mini player and exits the mini player	Success
Film Watcher	“User presses ‘Back to tab’ while in mini player”	The film exits the mini player	The film exits the mini player	Success
Film Watcher	“User drags the mini player screen”	The mini player is dragged to the user's desired location.	The mini player is dragged to the user's desired location.	Success
Film Watcher	“User presses ‘Return to films’ button to return to browsing films”	The user redirects to the movie browsing page.	The user redirects to the movie browsing page.	Success
Film Watcher	“User slides the movie bar to the right”	The film skips to a later part of the film.	The film skips to a later part of the film.	Success
Film Watcher	“User slides the movie bar to the left”	The film skips to an earlier part of the film.	The film skips to an earlier part of the film.	Success
Film Watcher	“User raises the volume up”	The film will raise in volume.	The film will raise in volume.	Success
Film Watcher	“User lowers the volume up”	The film will lower in volume.	The film will lower in volume.	Success
Film Watcher	“User mutes the film”	The film will be muted.	The film will be muted.	Success