FINAL PROJECT REPORT

SEMESTER 1, ACADEMIC YEAR: 2025-2026

CT312H: MOBILE PROGRAMMING

- **Project/Application name:** Movie Trailer Application
- **GitHub link:** https://github.com/25-26Sem1-Courses/ct312hm01-project-Gabu1909
- **Youtube link:** https://youtu.be/cO0LFYV68qs
- **Student ID 1:** B2205999
- **Student Name 1:** Huỳnh Thanh Nhuận
- **Student ID 2:** B2206004
- **Student Name 2:** Trần Thị Kim Phụng
- **Class/Group Number:** CT312HM01

I. Introduction

This Mobile Movie Trailer Application is a modern, cross-platform solution built on the Flutter framework, designed to deliver a smooth and responsive user experience while serving as a comprehensive client for The Movie Database (TMDB) API, offering real-time data for Now Playing, Popular, and Top Rated movies, alongside detailed Movie and Genre information. The application utilizes a clean, layered architecture where state management is handled robustly by the Provider package, with the MovieProvider and FavoritesProvider efficiently managing data flow using concurrent fetching (Future.wait) for optimal performance. GoRouter provides declarative navigation, enabling complex routing structures like ShellRoute for the main Bottom Navigation and dynamic paths for details (/movie/:id) and comprehensive list views (/see-all). Furthermore, core user data persistence is achieved using a local SQLite database via the SQFlite package, abstracted through the DatabaseHelper to securely store and manage the user's favorite movies across application sessions, thereby meeting all core project requirements for modern mobile development.

- A task assignment sheet for each member if working in groups.

No.	Task Description	Member(s)
1	Setup Core Architecture & Models	Tran Thi Kim Phung Huynh Thanh Nhuan
2	Implement API Service Layer	Tran Thi Kim Phung
3	Implement State Management with Provider	Huynh Thanh Nhuan
4	Setup Local Database for Favorites	Huynh Thanh Nhuan
5	Configure App Navigation with GoRouter	Huynh Thanh Nhuan Tran Thi Kim Phung
6	Design & Build the Movie Discovery UI	Tran Thi Kim Phung
7	Design & Build the Movie Detail & Search Pages	Tran Thi Kim Phung
8	Implement Favorites Feature	Huynh Thanh Nhuan

II. Details of implemented features

1. Home Screen:

Description: The Home Screen is the central page for movie discovery. It fetches and displays categorized lists of movies: **Popular**, **Now Playing**, and **Top Rated**. It features a navigation **Drawer** for supplementary options (like "Category" and "Settings") and an **AppBar** action button to navigate to Search. It supports **pull-to-refresh** to update data.

- **Screenshots:**

- **Implementation details:**

  - **Widgets:**

    + Scaffold, AppBar, Drawer, ListView, DrawerHeader, IconButton, Consumer<MovieProvider>, RefreshIndicator, SingleChildScrollView, Column, Center, CircularProgressIndicator.
    + **Special Widgets: MovieList** (Custom Widget, assumed to handle the title and horizontal scrolling of movies) is used three times. _buildDrawerItem is a helper method returning a ListTile for the Drawer.

  - **Libraries/Plugins**:

    + **provider**: Used for state management via Consumer<MovieProvider> to listen to movie lists and loading state.
    + **go_router**: Used for navigation, specifically context.go('/settings') (from AppBar) and context.push('/search') (from Drawer). It also uses

context.go('/') and context.push() for other Drawer items (though the specific destination logic is simplified in the provided snippet).

- **Shared State Management**:

  + **MovieProvider**. The screen is wrapped in a Consumer<MovieProvider>. Upon initialization (initState), it checks if nowPlayingMovies is empty and calls provider.fetchAllMovies() (which, based on the MovieProvider code, fetches three movie lists). The UI then listens to the provider.isLoading flag to show a CircularProgressIndicator and updates automatically when the movie lists are populated via notifyListeners().

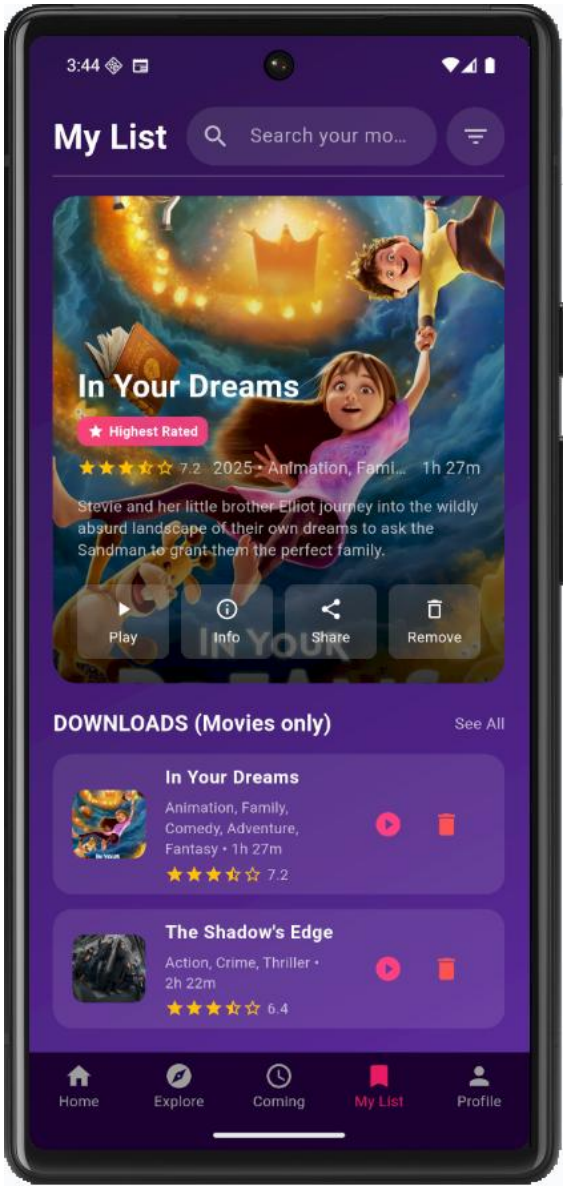
- **Data Usage (Local/Remote)**:

  + **Remote**. It reads three lists of Movie objects (nowPlayingMovies, popularMovies, topRatedMovies).
  + **REST API:** All data is fetched from the TMDB API using GET requests through the ApiService.
  + **API Calls:** GET /movie/now_playing, GET /movie/popular, GET /movie/top_rated.
  + **Input:** Only the api_key is required in the URL query parameters.
  + **Output:** A JSON response containing a list of movie objects, which are mapped to the local Movie model.

2. My List Screen

Description: The Favorites Screen displays a list of all movies locally saved by the user. It shows a list of movie posters in a grid layout and informs the user if the favorites list is empty.

- **Screenshots:**



- **Implementation details:**

- Widgets:

+ Scaffold, AppBar, Consumer<FavoritesProvider>, Center, Text, GridView.builder, SliverGridDelegateWithFixedCrossAxisCount.
+ **Special Widgets:** None beyond standard widgets and the custom MovieCard for display.

- Libraries/Plugins:

+ **provider:** Used to listen to the FavoritesProvider state, specifically the favorites list.

- Shared State Management:

+ **FavoritesProvider (using ChangeNotifier)**. The screen uses Consumer<FavoritesProvider> to react to updates in the **favorites** list. The provider is instantiated in main.dart and calls **loadFavorites()** on creation to populate the list from the database. Any changes (add/remove from detail screen) trigger notifyListeners() which rebuilds this screen's GridView.

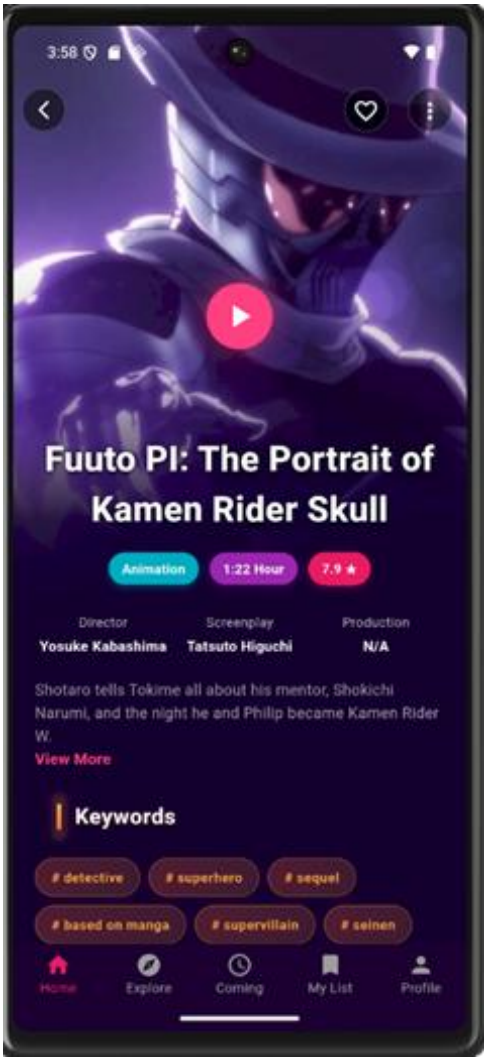
- Data Usage (Local/Remote):

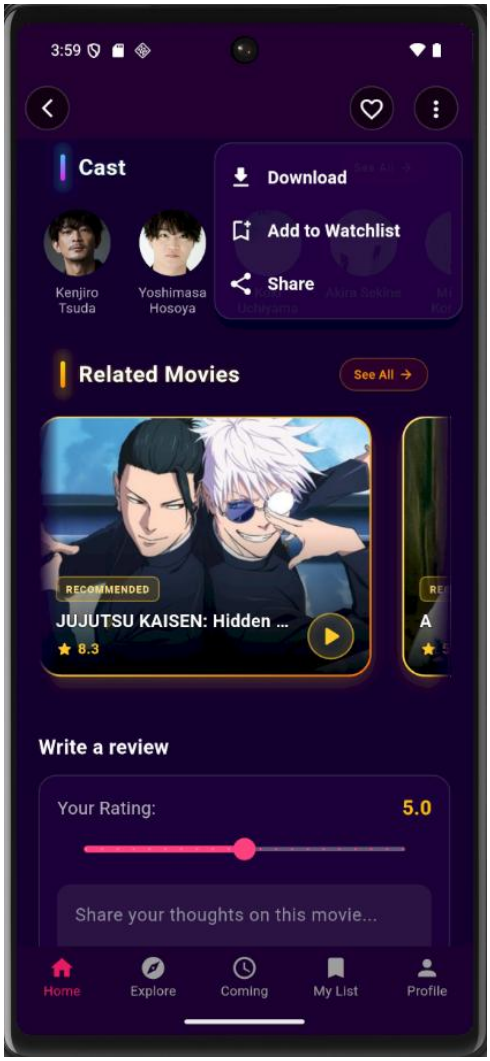
+ **Local Persistence (SQLite)**. Data is read from the local database via DatabaseHelper.
+ **Data Table Structure:** The DatabaseHelper defines the favorites table structure as follows: CREATE TABLE favorites (id INTEGER PRIMARY KEY, title TEXT NOT NULL, overview TEXT NOT NULL, posterPath TEXT, voteAverage REAL NOT NULL). Data is read by querying this table (db.query('favorites')).

3. Movie Detail Screen

Description: The Movie Detail Screen displays detailed information for a single movie, dynamically fetched upon navigation. It features an expanding SliverAppBar with the movie poster and includes a button to toggle the movie's favorite status, which is instantly reflected in the UI.

- **Screenshots:**





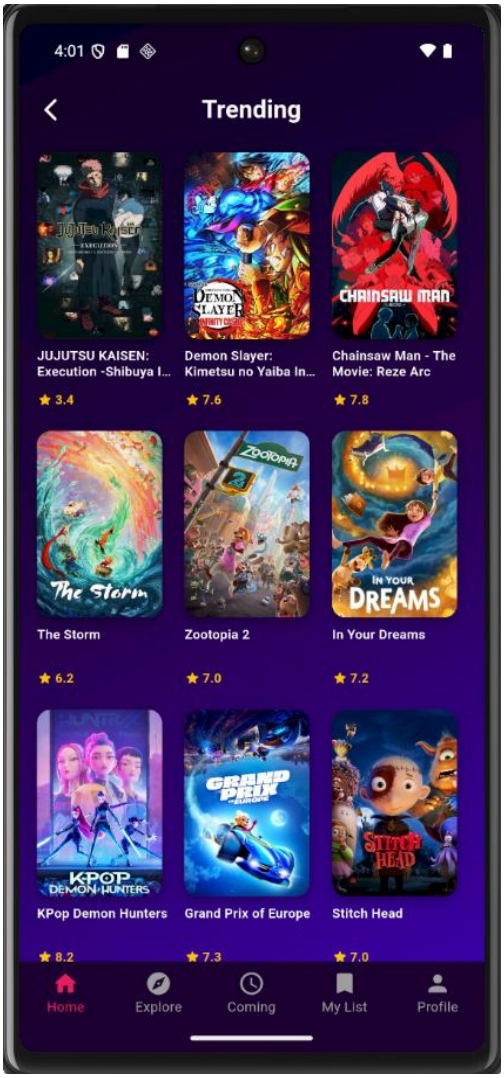
- **Implementation details:**
  - **Widgets:**
    + Scaffold,FutureBuilder<Movie>,CustomScrollView, SliverAppBar,SliverList,Padding,Column,Text,Consumer<FavoritesProvider>,IconButton,CircularProgressIndicator.
    + **Special Widgets: CachedNetworkImage** (from the external library cached_network_image) is used for efficient and cached loading of the remote movie poster. SliverAppBar and CustomScrollView are used to create the collapsing/expanding header effect.
  - **Libraries/Plugins**:

    + **cached_network_image**: An external library used for loading and caching network images.
    + **provider**: Used to check and update the movie's favorite status via FavoritesProvider.
    + **http**: Used indirectly via ApiService to fetch the movie's details.
- **Shared State Management**:
    + **FavoritesProvider**. The IconButton is wrapped in a Consumer<FavoritesProvider>. This ensures only the button (and its icon/color) re-renders when the favorite status changes. It checks provider.isFavorite(movie.id) to determine the icon and calls provider.toggleFavorite(movie) on press, which triggers the database update and state change.
- **Data Usage (Local/Remote)**:
    + **Remote (Detail)** and **Local (Favorite Status)**.
    + **Remote:** Fetches specific movie details using a **GET** request.
    + **API Call:** GET /movie/$movieId?api_key=....
    + **Input:** The movieId path parameter.
    + **Output:** A single detailed Movie object.
    + **Local:** The FavoritesProvider interacts with the local SQLite database to check and update the favorite status of the movie using the id as the primary key.

4. See All Screen

Description: The See All Screen is a general-purpose screen used to display a full list of movies from any category (e.g., all "Popular" movies) in a scrollable grid format. It is designed to receive the list of movies and a title dynamically during navigation.

- **Screenshots:**



- **Implementation details:**

  - **Widgets:**
    + Scaffold,AppBar,GridView.builder,SliverGridDelegateWithFixedCrossAxisCount.
    + **Special Widgets:** None beyond standard widgets and the custom MovieCard.
  - **Libraries/Plugins**:
    + **provider**: **go_router**: Used indirectly by AppRouter to pass required arguments (title and movies) to this screen using the state.extra mechanism.
  - **Shared State Management**: **No**. This screen is a StatelessWidget and receives all necessary data (title and movies) directly as arguments upon

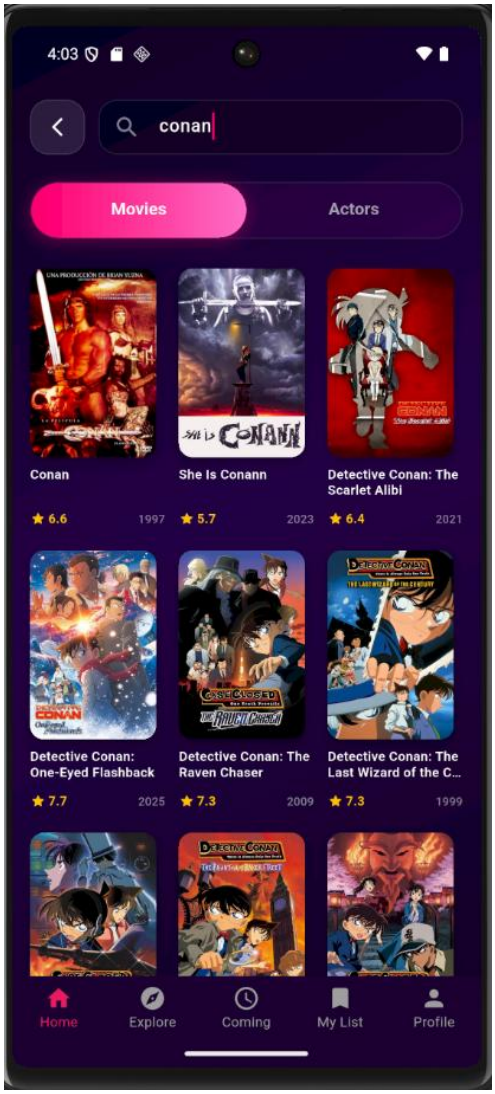
creation (from AppRouter), so it does not need to use Provider to listen for global state changes.

- **Data Usage (Local/Remote)**: **None directly**. It receives a pre-fetched list of Movie objects (movies) passed as an argument from the preceding screen or the router. The data was originally fetched remotely by the MovieProvider.

5. Search Screen

Description: The Search Screen provides an interactive interface for users to search for movies by title. It features a controlled TextField for input and dynamically updates a GridView of results based on the search query. It handles loading and displays a "No results found" message when the search yields no matches.

- **Screenshots:**



- **Implementation details:**

- Widgets:

+ Scaffold, AppBar, Column, Padding, TextField, InputDecoration, IconButton, Expanded, Consumer<MovieProvider>, GridView.builder, SliverGridDelegateWithFixedCrossAxisCount.
+ **Special Widgets:** None beyond standard and custom UI widgets (MovieCard). It uses a TextEditingController to manage the TextField input state.

- Libraries/Plugins:

+ **provider**: **provider**: Used to invoke the searchMovies method and listen for the searchedMovies results and isLoading flag.

- Shared State Management:

+ **MovieProvider**. The _onSearch method retrieves the MovieProvider instance (listen: false) and calls provider.searchMovies(query). This provider updates its internal _searchedMovies list and sets _isLoading to false before calling notifyListeners(). The screen's Consumer then rebuilds the GridView.builder using the provider.searchedMovies list.

- Data Usage (Local/Remote): It performs on-demand searches.

+ **REST API:** Data is fetched from the TMDB API via the ApiService.
+ **API Call:** GET /search/movie?query=$query&api_key=....
+ **Input:** The movie title (query) is passed in the URL query string.
+ **Output:** A JSON response containing a list of Movie objects matching the query, which populate the searchedMovies list.

6. Categories Screen

- Description:

This feature is a comprehensive navigation and content filtering drawer, accessible from the main HomeScreen by tapping the menu icon (a "three-bar" or "hamburger" icon).

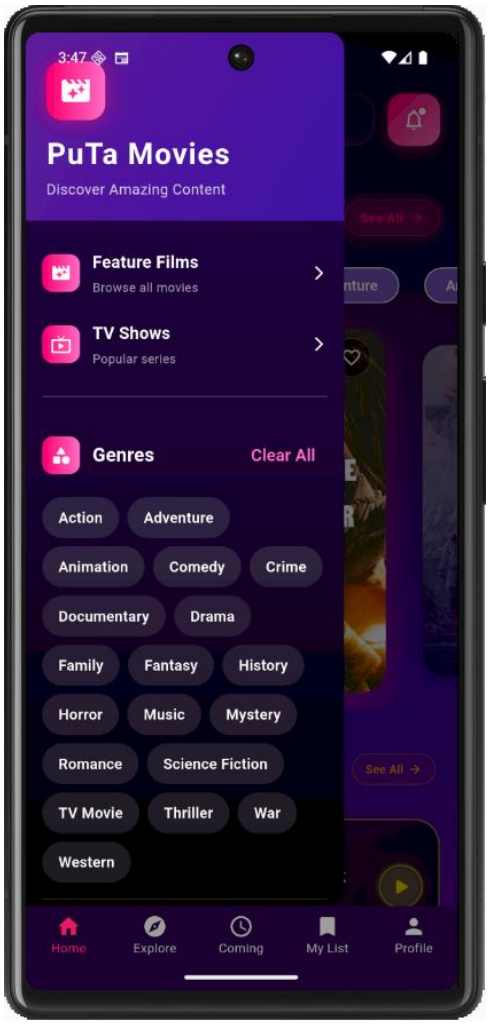
Its primary purpose is to serve as the central hub for users to discover and filter the application's content. It allows users to:

- **Navigate to Broad Categories**: Quickly access pre-defined pages for "Feature Films" and "TV Shows".

- **Filter by Genre**: Select one or more genres (e.g., Action, Comedy, Horror) to find movies that match their interests.
- **Filter by Country**: Select one or more countries of origin to discover international cinema.
- **Apply Filters**: Combine selected genres and countries to generate a specific, filtered list of movies, which is then displayed on a new screen.
- **Clear Selections**: Easily reset all active filters to start a new search.

The drawer provides a rich, interactive experience with dynamic UI updates, reflecting the user's current selections in real-time.

- **Screenshots:**



- **Implementation details:**

  - **Widgets:**

- Scaffold: The fundamental widget that provides the structure for the screen, including the drawer property where our feature is placed. A GlobalKey<ScaffoldState> is used to programmatically open the drawer.
- Drawer: The primary widget that creates the slide-out navigation panel.
- ListView: Used as the main child of the Drawer to ensure its content is scrollable, accommodating various screen sizes and a potentially long list of genres.
- Container, BoxDecoration, LinearGradient: These are used extensively to create the drawer's custom look, including the dark gradient background and the styled header.
- Column, Row, Wrap: Standard layout widgets. Wrap is particularly important for displaying the genre and country filter chips, as it allows them to flow onto multiple lines gracefully if they don't fit in a single row.
- Divider: A thin line used to visually separate the different sections within the drawer (e.g., separating navigation items from filter sections).
- ElevatedButton.icon: Used for the "Apply Filters" button, providing both an icon and text.
- **Special Widgets**: The feature uses a custom-composed "chip" widget created from several standard widgets, which is a common and powerful pattern in Flutter.
    - The _buildDrawerChip and _buildCountryDrawerChip methods create interactive filter chips. They don't use the standard Chip or ChoiceChip widget. Instead, they combine an InkWell (for tap effects and logic), an AnimatedContainer (for smooth visual transitions when selected/deselected), and a BoxDecoration (for styling, color, borders, and rounded corners). This custom composition gives complete control over the appearance and animation of the filter chips.

- Libraries/Plugins:

1. **provider**: This is a state management library.
    - **Role**: It manages the application's shared state through a MovieProvider class. For the drawer feature, MovieProvider holds the lists of selected genres (selectedDrawerGenreIds) and countries

(selectedCountries). When a user taps a chip, a method in the provider is called to update this state. The UI, such as the color of the chips and the state of the "Apply Filters" button, listens to these changes and rebuilds accordingly. It also contains the business logic for fetching data based on the selected filters.

2. **go_router**: This is a declarative routing package.
    - **Role**: It handles all navigation within the app. When the user taps "Apply Filters" or a main navigation item like "TV Shows", go_router is used to push a new route (/see-all). The filtered list of movies and a corresponding title are passed to the new SeeAllScreen as route parameters (extra), ensuring the next screen displays the correct data.

- Shared State Management:

1. **MovieProvider Class**: This class extends ChangeNotifier. It holds the application's data and UI state, such as List<Genre> genres, Set<int> selectedDrawerGenreIds, and Set<String> selectedCountries.
2. **State Modification**: When a user taps a genre chip, the onTap event calls provider.toggleDrawerGenre(genre.id). This method adds or removes the genre's ID from the selectedDrawerGenreIds set and then calls notifyListeners().
3. **UI Updates**:
    - Widgets that need to rebuild when the state changes, like the filter chips (_buildDrawerChip), use context.watch<MovieProvider>() or are wrapped in a Consumer<MovieProvider> widget. When notifyListeners() is called, these widgets automatically rebuild, reflecting the new state (e.g., changing color to show they are selected).
    - For actions that only read the state or call a method without needing to listen for changes (like the onPressed callback of the "Apply Filters" button), context.read<MovieProvider>() or Provider.of<MovieProvider >(context, listen: false) is used for better performance.

- Data Usage (Local/Remote): It performs on-demand searches.

+ API Description: The feature uses the The Movie Database (TMDb) API to fetch movie data. When the "Apply Filters" button is pressed, the getMoviesByFilter() method in MovieProvider is executed.

+ API Call: This method constructs a URL for the TMDb /discover/movie endpoint.

+ Input: The method takes the selected filters from the provider's state (selectedDrawerGenreIds and selectedCountries) and adds them to the API request as URL query parameters.

+ with_genres: A comma-separated string of genre IDs (e.g., 28,878 for Action & Sci-Fi).

+ with_origin_country: A comma-separated string of country codes (e.g., US,KR for USA & Korea).

+ An API key is also included for authentication.

+ Example URL: https://api.themoviedb.org/3/discover/movie?api_key=YOUR_API_KEY&with_genres=28,12&with_origin_country=US

+ Output: The API returns a JSON object containing a list of movies that match the specified criteria. Each movie object in the list is a complex JSON object with details like id, title, poster_path, vote_average, etc. This JSON response is then parsed into a List<Movie> of Dart objects, which is passed to the SeeAllScreen for display.

7. Profile Screen

Description: The Profile Screen serves as the user's personal hub within the PuTa Movies application. It is designed to provide a visually engaging and informative overview of the user's activity and account details.

Key Functions:

- **User Identification:** Displays the user's avatar, full name, and a short bio. The avatar is presented with an animated, glowing border and sparkling effects for a dynamic and modern feel.

- **Statistics Summary:** Presents key user statistics in a "glassmorphism" style card. This includes the total number of movies/shows in their watchlist, the number of reviews they have written, and the year they joined the application.

- **Account Management:** Provides quick access to essential account-related features such as 'Edit Profile', 'Security', and 'Subscription' status.

- **Content Navigation:** Offers shortcuts to user-specific content pages, including 'My List' (the watchlist) and 'My Reviews'.
- **Application Settings:** Contains links to general application pages like 'Settings' and 'Help Center'.
- **Session Control:** Includes a prominent 'Log Out' button to allow the user to securely end their session.

The screen employs a sophisticated UI with layered elements, including gradient backgrounds, blurred "blob" effects, and smooth animations to create a premium user experience.

- **Screenshots:**

- **Implementation details:**

- a. Widgets Used

The screen is built using a combination of standard and specialized Flutter widgets to achieve its complex layout and animations.

- **Layout & Structure:** Scaffold, AppBar, Stack, Positioned, SafeArea, SingleChildScrollView, Column, Row, SizedBox.

- **Styling & Decoration:** Container, DecoratedBox, ClipRRect, ClipOval, Border, LinearGradient, SweepGradient.
- **Text & Icons:** Text, Icon, IconButton.
- **User Interaction:** GestureDetector, InkWell, Material.
- **Asynchronous UI:** FutureBuilder is implicitly used via _isLoading state checks to show a CircularProgressIndicator while data is being fetched.

Special Widgets:

- **BackdropFilter:** This widget is used to create the "glassmorphism" or frosted-glass effect on the statistics card and settings groups. It applies a filter, in this case, ImageFilter.blur, to the area behind the widget.
- **AnimatedBuilder:** This is a core widget for the animations on this screen. It is used in conjunction with an AnimationController to:
    1. Create the continuously rotating gradient border around the user's avatar.
    2. Animate the small "sparkles" that orbit the avatar, which fade in and out as they move.
- **Transform.rotate:** Used within the AnimatedBuilder to apply the rotation transformation for the animated border.

b. Libraries and Plugins

- **go_router:** This library is used for all navigation within the app. On this screen, it handles navigating to other pages like /profile/edit, /settings, and logging out to /login via context.push() and context.go().
- **provider:** This is the primary state management solution. It is used to access the AuthProvider to get the current user's data (authProvider.currentUser) and to trigger the logout action (authProvider.logout()).
- **cached_network_image:** This plugin is used to display the user's profile picture. It efficiently loads the image from a URL and caches it, so it doesn't have to be re-downloaded every time. It also provides placeholder and error widgets.

- **intl:** The Internationalization library is used for date formatting. Specifically, DateFormat('yyyy') is used to extract the year from the user's account creation date.
- **dart:io:** This core Dart library is used to handle file paths, specifically for loading a user avatar from local device storage (Image.file(File(url))).

c. Shared State Management

Yes, this feature uses a shared state management solution via the **provider** package.

- **How it works:** The ProfileScreen depends on AuthProvider.
    - It uses context.watch<AuthProvider>() to listen for changes to the authentication state. This ensures that if the user's data (like their name or profile picture) is updated elsewhere, the UI on the profile screen will automatically rebuild to show the new information.
    - It uses context.read<AuthProvider>() inside the _loadUserStats method. This is a one-time read to get the current user's ID to fetch their stats from the database without subscribing to future updates, which is a performance optimization.
    - When the user taps the logout button, Provider.of<AuthProvider>(context, listen: false).logout() is called. This method within the AuthProvider class handles clearing the user session and notifying all listeners (like the router) that the authentication state has changed, triggering a navigation to the login screen.
- **Code Architecture:** The AuthProvider is likely provided at a high level in the widget tree, for example, wrapping the MaterialApp in main.dart. This makes the AuthProvider instance available to any widget down the tree that needs access to authentication state or user data.

d. Data Storage and API

This feature reads and stores data both locally and remotely.

- **Local Data (SQLite):** The screen fetches user-specific statistics from a local SQLite database using the DatabaseHelper class.

- DatabaseHelper.instance.getAllUserReviews(userId): This method is called to count the number of reviews a user has made. It likely queries a reviews table.
- DatabaseHelper.instance.getWatchlist(userId): This method is called to count the number of items in a user's watchlist. It queries a watchlist table.

Data Table Structure (assumed):

- **reviews table:**
  - id (INTEGER, PRIMARY KEY)
  - userId (TEXT)
  - movieId (INTEGER)
  - rating (REAL)
  - reviewText (TEXT)
  - createdAt (TEXT)
- **watchlist table:**
  - id (INTEGER, PRIMARY KEY)
  - userId (TEXT)
  - movieId (INTEGER)
  - mediaType (TEXT, e.g., 'movie' or 'tv')
  - addedAt (TEXT)

- **Remote Data (via AuthProvider):** The primary user information (name, email, profile image URL, creation date) is managed by AuthProvider. While the code doesn't show the direct API call, AuthProvider is responsible for fetching this data from a remote backend (like Firebase, Supabase, or a custom REST API) upon login and storing it in memory. The ProfileScreen then reads this data from the provider.

8. Trailer Screen

Description: The YouTube Trailer Screen is a dedicated, full-screen media player designed to provide an immersive and uninterrupted viewing experience for movie and TV show trailers. When a user taps on a trailer thumbnail elsewhere in the application, this screen is launched.

Key Functions:

- **Full-Screen Playback:** The screen immediately forces the device into landscape mode and hides the system status and navigation bars, allowing the video to occupy the entire screen.
- **Automatic Playback:** The video trailer begins playing automatically as soon as the player is ready, providing instant engagement.
- **Playback Control:** It utilizes the standard YouTube player interface, giving users familiar controls for play/pause, seeking, volume, and quality settings.
- **Easy Dismissal:** Users can exit the player in two intuitive ways: by tapping a clear 'close' icon at the top-right corner or by performing a vertical swipe-down gesture on the screen.
- **Automatic Exit:** The screen automatically closes and returns the user to the previous page once the video has finished playing.

This feature is crucial for enhancing user engagement by allowing them to quickly preview content without leaving the application's ecosystem.

- **Screenshots:**

- **Implementation details:**

- a. Widgets Used

The screen is constructed with a focused set of widgets to prioritize the video content.

- **Layout & Structure:** Scaffold, Stack, Center, Positioned, SafeArea.
- **User Interaction:** GestureDetector (to detect the vertical swipe-down to dismiss), IconButton (for the close button), WillPopScope (to intercept the back button press and pause the video).

Special Widgets:

- **YoutubePlayer:** This is the central widget of the screen, provided by the youtube_player_flutter package. It is not a standard Flutter SDK widget. It embeds a native YouTube player view (using a platform view) into the Flutter widget tree, handling all the complexities of video streaming and playback control.

b. Libraries and Plugins

Yes, this feature relies on several key libraries and plugins.

- **youtube_player_flutter:** This is the most critical plugin for this feature. Its role is to provide the YoutubePlayer widget and the YoutubePlayerController. It manages the entire lifecycle of the YouTube video playback, from initialization

with a video ID to handling player states (playing, paused, ended) and user interactions with the player controls.

- **flutter/services.dart:** This is a core Flutter library used for platform-specific interactions. On this screen, it is used for:
    - SystemChrome.setPreferredOrientations(): To lock the screen orientation to landscape (landscapeLeft, landscapeRight) when the player opens and restore all orientations when it closes.
    - SystemChrome.setEnabledSystemUIMode(SystemUiMode.immersiveSticky): To hide the top status bar and bottom navigation bar, creating a true full-screen experience. SystemUiMode.edgeToEdge is used to restore them on exit.
- **go_router (Implicitly):** While not directly imported, the navigation Navigator.of(context).pop() is part of the Flutter framework that works in conjunction with the app's routing solution, which is go_router in this project. It's used to close the screen.

c. Shared State Management

No, this feature **does not use a shared state management solution** like Provider or Riverpod.

- **How it works:** The YouTubePlayerScreen is a StatefulWidget that manages its own local state.
    - The primary state it manages is the YoutubePlayerController. This controller is initialized in initState() with the videoId passed into the widget's constructor.
    - The controller's lifecycle is tied directly to the widget's lifecycle; it is created in initState() and released in dispose().
    - Data (the videoId and title) is passed down from the parent widget via the constructor, not accessed from a shared provider. This is an effective and simple approach for a screen with such a self-contained and temporary function.

d. Data Storage and API

This feature reads data remotely, but not through a direct REST API call written within the widget itself.

- **Input:** The screen receives a videoId (e.g., dQw4w9WgXcQ) as a String parameter when it is created. This ID is the only piece of data the screen needs to function.

- **API Interaction:** The youtube_player_flutter plugin takes this videoId and uses it to communicate with the **YouTube IFrame Player API** under the hood. The plugin handles the API calls required to fetch the video stream and metadata from YouTube's servers.

- **Output:** The "output" is the video stream displayed on the screen. The screen itself does not process or store any data returned from the API. When the video ends (onEnded callback) or the user closes the screen, it simply pops the navigation stack.

There is no local data storage (like SQLite) or custom REST API interaction on this screen.

9. Explore Screen

Description: The Explore Screen serves as the application's news hub, providing users with a curated feed of the latest articles and updates from the world of cinema. It is designed to keep users informed and engaged with current events, movie announcements, and industry news.

Key Functions:

- **News Aggregation:** Fetches and displays a list of recent movie-related news articles from a remote API.

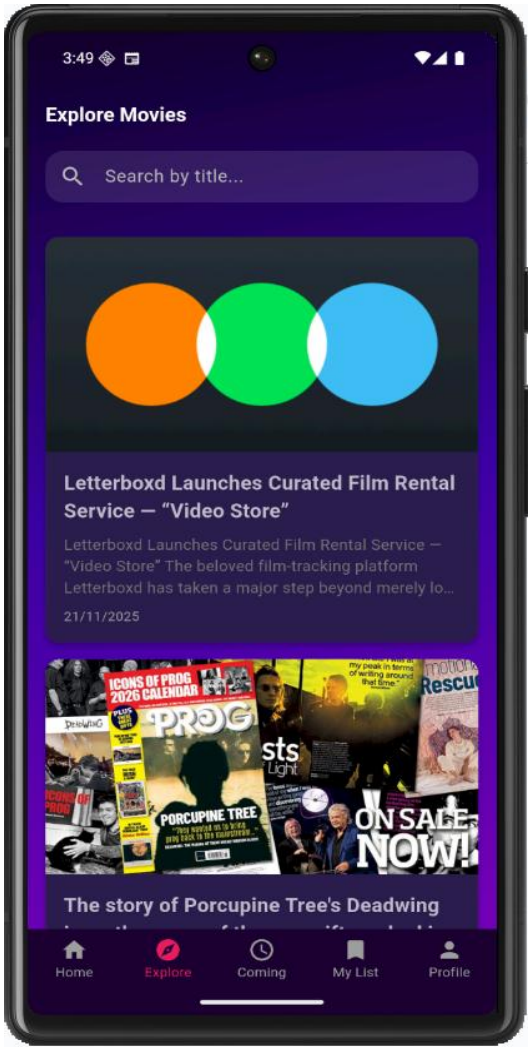
- **Dynamic Search:** Includes a prominent search bar that allows users to instantly filter the articles by title, making it easy to find news on specific topics.

- **Engaging UI:** Presents articles in a visually appealing list format, with each item showing a headline image, title, and source. The screen features an animated gradient background for a modern and fluid user experience.

- **Pull-to-Refresh:** Users can swipe down from the top of the list to manually refresh the news feed and load the latest content.

- **Responsive Layout:** The app bar, containing the title and search field, elegantly collapses and expands as the user scrolls through the news list.

- **Screenshots:**



- **Implementation details:**

- a. Widgets Used

The screen is built using a combination of standard and specialized Flutter widgets to create its dynamic and scrollable interface.

- **Layout &**
  Structure: Scaffold, AnimatedBuilder, Container, RefreshIndicator, CustomScrollView, SliverAppBar, SliverFillRemaining, Padding, Center.
- **Input:** TextField is used for the search functionality.
- **Display:** Text, Icon, CircularProgressIndicator.
- **Custom Widgets:** MovieNewsSection is a custom widget that takes the list of articles and renders them, likely using a SliverList and custom card widgets for each article internally.

Special Widgets:

- **CustomScrollView & SliverAppBar:** These widgets are used together to create the flexible, scrolling app bar effect. The SliverAppBar can float, pin, and expand/collapse as the user scrolls the list of news, providing a modern and space-efficient UI.
- **RefreshIndicator:** This widget wraps the main scroll view to provide the "pull-to-refresh" functionality, allowing users to easily fetch the latest news.
- **AnimatedBuilder:** This widget is used in conjunction with an AnimationController to create the smoothly animating gradient background, making the UI feel more alive and dynamic.

b. Libraries and Plugins

Yes, this feature uses the following libraries:

- **http:** This package is essential for the feature. Its role is to make the HTTP GET request to the News API to fetch the movie news articles.
- **dart:convert:** A core Dart library used to decode the JSON response received from the news API. Specifically, json.decode(utf8.decode(response.bodyBytes)) is used to handle UTF-8 encoded characters in the response body.

c. Shared State Management

No, this feature **does not use a shared state management solution** like Provider.

- **How it works:** The ExploreScreen is a StatefulWidget that manages its own local state internally.
    - **State Variables:** It uses local state variables like _isLoading (to show a loading indicator), _error (to display error messages), _allArticles (to hold the master list of news), and _filteredArticles (to hold the list displayed to the user after searching).
    - **initState():** In initState(), it calls _fetchMovieNews() to load the initial data.
    - **setState():** The state is updated using setState() whenever the data is fetched, an error occurs, or the user types in the search bar.

The _searchController has a listener that calls _filterArticles, which in turn calls setState() to rebuild the UI with the filtered list.

- **Code Architecture:** This self-contained state management is appropriate for a screen whose data is not needed by other parts of the application.

d. Data Storage and API

This feature reads data remotely from a REST API. It does not store any data locally.

- **REST API:** The screen communicates with the **NewsAPI**.
    - **API Call:** It makes an HTTP GET request to the everything endpoint.
    - **Endpoint URL:** https://newsapi.org/v2/everything
    - **Input (Query Parameters):**
        - q=movie: Specifies that the search query is "movie".
        - language=en: Requests articles in English.
        - sortBy=publishedAt: Sorts the results by the most recently published.
        - apiKey: The secret API key for authentication.
    - **Output (JSON Response):** The API returns a JSON object. The primary data is within the articles key, which contains a list of article objects.
- **Data Structure (for each article in the list):** Each article is a JSON object

10. Coming Soon Screen

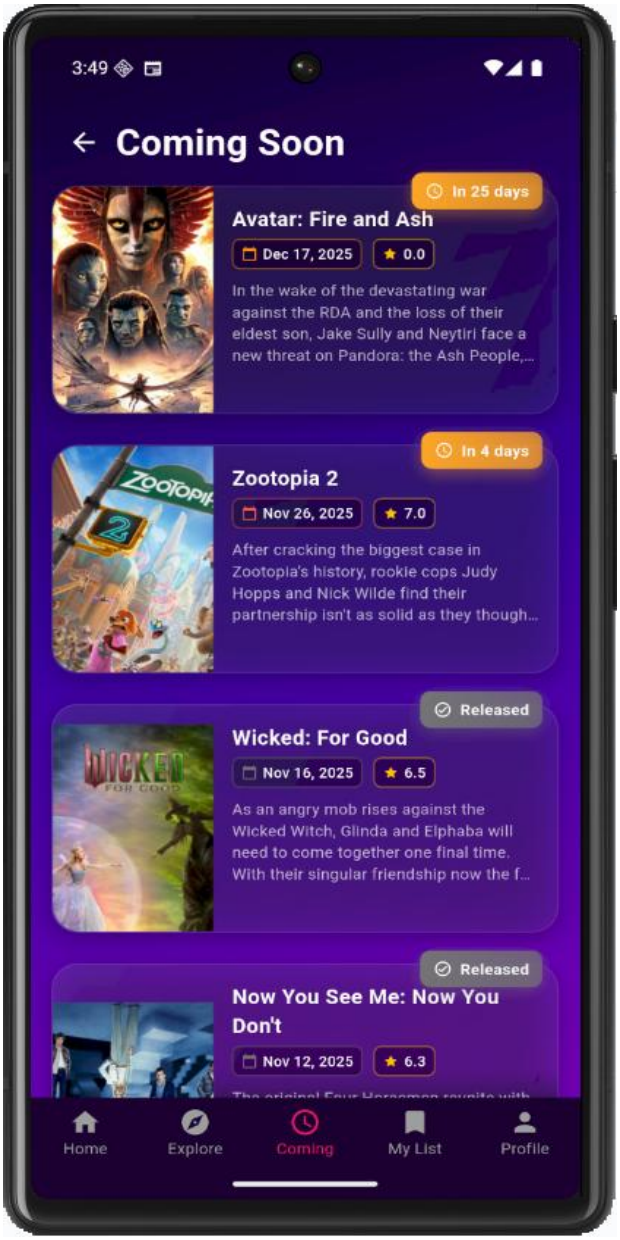
Description: The "Coming Soon" screen is designed to build anticipation by showcasing a list of movies that are scheduled for future release. It provides users with an elegant and interactive way to discover and track upcoming films.

Key Functions:

- **Upcoming Movie Feed:** Displays a vertically scrollable list of movies fetched from an external API, ordered by their release date.

- **Infinite Scrolling:** As the user scrolls towards the bottom of the list, the app automatically fetches and appends the next page of results, creating a seamless browsing experience.
- **Pull-to-Refresh:** Users can pull down from the top of the screen to refresh the entire list, ensuring they have the most up-to-date information.
- **Parallax Effect:** Each movie poster features a subtle parallax scrolling effect, where the image moves at a different speed than the card, adding a sense of depth and premium quality to the UI.
- **Release Countdown:** A prominent badge on each movie card indicates the time until its release (e.g., "In 15 days", "Tomorrow", "Today!").
- **Context Menu:** A long-press on any movie card reveals a context menu with quick actions, such as adding/removing from the watchlist, sharing the movie with friends, or viewing more details.

- **Screenshots:**



- **Implementation details:**

- a. Widgets Used

The screen is built with a rich set of widgets to achieve its interactive and visually appealing design.

- **Layout & Structure:** Scaffold, AnimatedBuilder, RefreshIndicator, CustomScrollView, SliverList, SliverFillRemaining, SliverToBoxAdapter, Stack, Positioned, Row, Column, Expanded, SizedBox, Padding, SafeArea.
- **Interaction:** GestureDetector (for tap and long-press), IconButton.

- **Display:** Text, Icon, CircularProgressIndicator, CachedNetworkImage, Hero.
- **Styling & Effects:** Container, DecoratedBox, ClipRRect, BackdropFilter (for the frosted glass effect on cards), LinearGradient.

Special Widgets:

- **Flow with a custom _ParallaxFlowDelegate:** This is a highly specialized widget combination used to create the parallax effect on the movie posters. The Flow widget allows for custom layout and painting logic. The _ParallaxFlowDelegate calculates the scroll position of the list item relative to the viewport and applies a Matrix4.translation to the poster image, making it move at a different rate than the list itself.
- **AnimatedContainer:** Used on the _EnhancedMovieCard to provide a subtle scale-down animation when the card is long-pressed, giving the user tactile feedback.
- **PopupMenuItem / showMenu:** These are used together to implement the custom context menu that appears on a long-press gesture.

b. Libraries and Plugins

Yes, this feature relies on several important libraries and plugins.

- **provider:** This is the core state management library. It is used to access:
    - MovieProvider: To fetch the list of upcoming movies, handle pagination (infinite scroll), and manage loading/error states.
    - WatchlistProvider: To check if a movie is in the user's watchlist and to add or remove it via the context menu.
- **go_router:** Used for declarative navigation. When a user taps a movie card, context.push('/movie/${movie.id}') is called to navigate to the movie details screen.
- **cached_network_image:** This plugin is crucial for efficiently loading and caching movie poster images from URLs. It also provides placeholder widgets (Shimmer) while the image is loading.
- **shimmer:** Used as a placeholder within CachedNetworkImage to display a loading animation before the poster image appears.

- **intl:** The Internationalization library is used for date formatting. DateFormat.yMMMd() formats the movie's release date into a readable string.
- **share_plus:** This plugin is used to invoke the native platform sharing UI, allowing users to share movie details through other apps.

c. Shared State Management

Yes, this feature uses the **provider** package for shared state management.

- **How it works:**
    - The ComingSoonScreen is a Consumer of MovieProvider. It listens for changes to the upcomingMovies list and the isLoading/isFetchingMoreUpcoming flags to rebuild the UI accordingly.
    - The _onScroll listener calls context.read<MovieProvider>().fetchMoreUpcomingMovies() to trigger the fetching of the next page of data when the user nears the end of the list. This is a "read" because it's a one-time action, not a subscription.
    - The RefreshIndicator's onRefresh callback calls movieProvider.fetchAllData() to reload all data from scratch.
    - In the context menu, context.read<WatchlistProvider>().toggleWatchlist(movie) is called to add or remove a movie from the user's watchlist. The WatchlistProvider handles the underlying database logic and notifies its listeners.
- **Code Architecture:** MovieProvider and WatchlistProvider are provided at a high level in the widget tree (likely above MaterialApp). This makes them accessible to any screen that needs movie data or watchlist functionality, promoting a clean separation of UI and business logic.

d. Data Storage and API

This feature reads data remotely and interacts with local storage via a provider.

- **Remote Data (REST API):** The screen gets its primary data from The Movie Database (TMDB) API, orchestrated by the MovieProvider.

- **API Call:** The provider makes an HTTP GET request to the /movie/upcoming endpoint.
- **Endpoint URL:** https://api.themoviedb.org/3/movie/upcoming
- **Input (Query Parameters):** The API call includes the apiKey for authentication, language for localization, and a page parameter for pagination, which is incremented by fetchMoreUpcomingMovies() for infinite scrolling.
- **Output (JSON Response):** The API returns a JSON object containing a page number, a list of movie objects under the results key, total_pages, and total_results. The Movie model class is used to parse each object in the results list.

- **Local Data (via WatchlistProvider):** While this screen doesn't directly query the database, the WatchlistProvider does. When a user adds a movie to their watchlist, the provider saves the movie's ID and other relevant details to a local SQLite database. This data persists between app sessions. The structure is likely a watchlist table containing columns like id, userId, movieId, title, posterPath, etc.

11. Login Screen

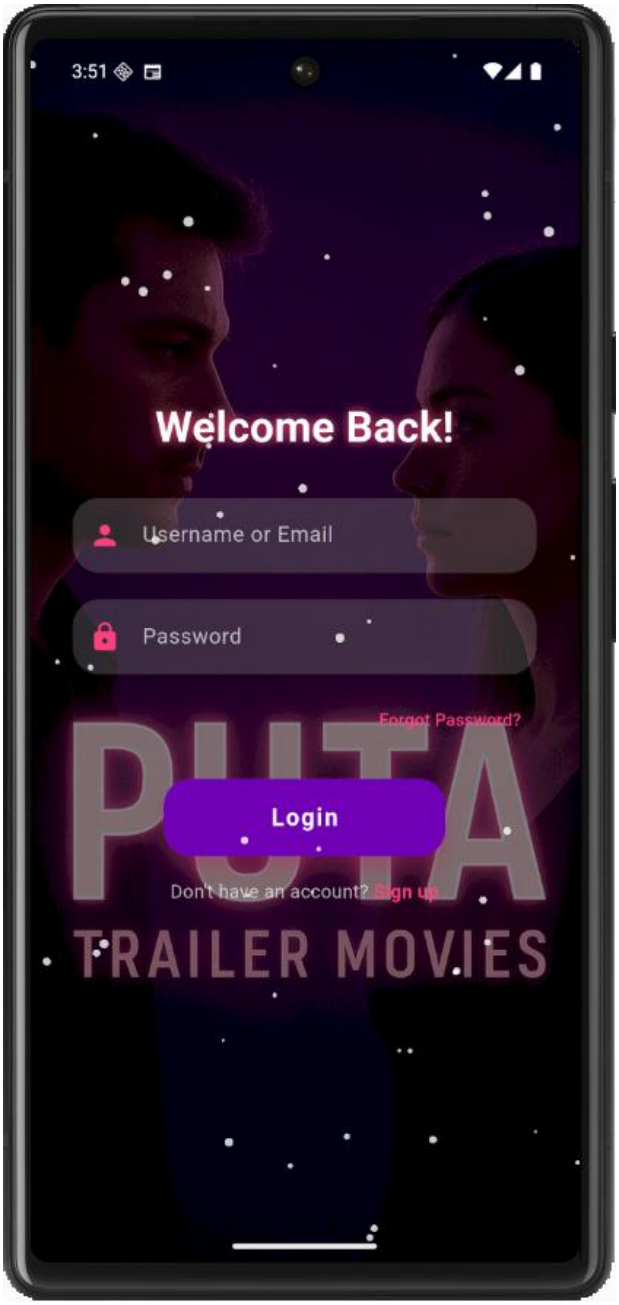
Description: The Login Screen is the primary authentication gateway for existing users of the PuTa Movies application. It provides a secure and visually engaging interface for users to access their accounts.

Key Functions:

- **User Authentication:** Allows users to log in using their registered username or email address and their password.
- **Credential Persistence:** Remembers the user's email and password from the last successful login session and pre-fills the input fields for a faster login experience.
- **Navigation:** Provides clear, tappable links for users who don't have an account ("Sign up") to navigate to the registration screen, and for those who have forgotten their password ("Forgot Password?") to navigate to the password recovery flow.

- **Feedback and State Handling:** Displays a loading indicator while the authentication request is in progress and shows clear error messages (via a snackbar) if the login attempt fails.
- **Immersive UI:** Features a full-screen background image with a dark overlay and a subtle, animated snow effect to create a cinematic and polished atmosphere. The entire form fades in smoothly when the screen is first loaded.

- **Screenshots:**



- **Implementation details:**

a. Widgets Used

The screen is built using a variety of Flutter widgets to create its animated and interactive layout.

- **Layout & Structure:** Scaffold, Stack, FadeTransition, Center, SingleChildScrollView, Form, Column, Row, SizedBox, Align.
- **Display & Graphics:** Image.asset, Container (for overlays and gradients), Text, Icon, CircularProgressIndicator.
- **Input & Interaction:** TextFormField, ElevatedButton, TextButton, GestureDetector.

Special Widgets:

- **FadeTransition & AnimationController:** These are used together to create the smooth fade-in effect for the entire login form when the screen first appears, enhancing the user experience.
- **SnowEffect:** This is a custom widget that renders an animated overlay of falling snowflakes, adding to the screen's aesthetic appeal. It is not a standard Flutter widget.

b. Libraries and Plugins

Yes, this feature uses several key libraries and plugins.

- **go_router:** This library is used for all navigation. On this screen, it handles:
    - Navigating to the home screen upon successful login (context.go('/home')).
    - Navigating to the registration screen (context.go('/register')).
    - Navigating to the password recovery screen (context.push('/forgot-password')).
- **provider:** This is the shared state management solution. It is used to access the AuthProvider to perform the login action by calling Provider.of<AuthProvider>(context, listen: false).login(...).
- **shared_preferences:** This plugin is used for simple local data persistence. Its role on this screen is to read the previously saved email and password from the

device's local storage to pre-populate the input fields, making it more convenient for returning users.

c. Shared State Management

Yes, this feature uses a shared state management solution via the **provider** package.

- **How it works:** The LoginScreen interacts with a central AuthProvider.
    - It uses Provider.of<AuthProvider>(context, listen: false) to get an instance of the provider and call the login() method. The listen: false argument is a performance optimization, as this screen only needs to trigger an action, not react to state changes within the AuthProvider.
    - The login() method within AuthProvider contains the business logic for authenticating the user (e.g., making an API call to a server).
    - If the authentication is successful, the AuthProvider updates its own state (e.g., stores the user object and token) and the login() method completes. The LoginScreen then manually navigates to the home screen using context.go('/home').
- **Code Architecture:** The AuthProvider is provided at the root of the application's widget tree (likely wrapping the MaterialApp). This makes a single instance of the provider available to any widget that needs to access authentication status or perform auth-related actions, ensuring a single source of truth for user session data.

d. Data Storage and API

This feature reads data locally and sends data to a remote API.

- **Local Data (Read):** The screen reads data from the device's local storage using the shared_preferences plugin.
    - **Data Structure:** It reads two key-value pairs:
        - 'saved_email': (String) The user's email from the last login.
        - 'saved_password': (String) The user's password from the last login.
    - This data is loaded in initState() to pre-fill the _usernameOrEmailController and _passwordController.

- **Remote Data (Write/API Call):** The actual authentication is handled by a remote API, which is called from within the AuthProvider.
    - **API Description (Inferred):** The authProvider.login(email, password) method likely makes a POST request to a backend authentication endpoint.
    - **Endpoint (Example):** https://api.yourapp.com/auth/login
    - **Input (Request Body):** A JSON object containing the user's credentials.
    - **Output (Success Response):** A JSON object containing a user object and an authentication token.
    - **Output (Error Response):** A JSON object with an error message and a corresponding HTTP status code (e.g., 401 Unauthorized).

12. Signup Screen

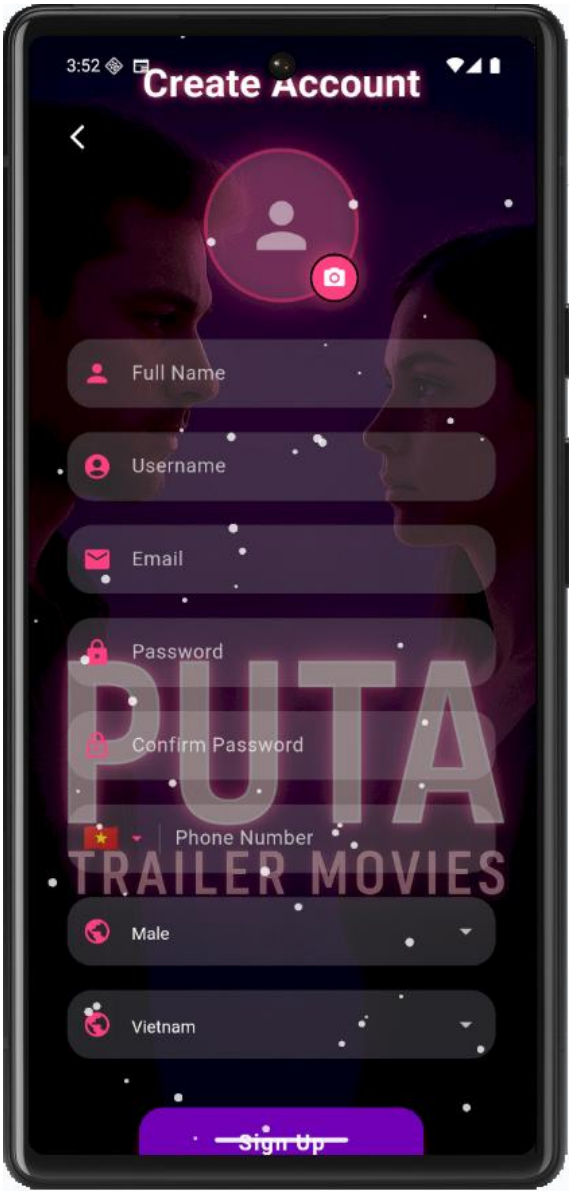
Description: The Sign Up Screen is the entry point for new users to join the PuTa Movies application. It is designed to be a simple, secure, and visually appealing page for creating a new user account.

Key Functions:

- **New User Registration:** Allows a new user to create an account by providing their full name, a valid email address, and a secure password.
- **Input Validation:** The form includes validation to ensure that all fields are filled out correctly before the registration request is sent.
- **Secure Authentication:** The screen captures user credentials and passes them to a central authentication provider, which handles the secure creation of the user account on the backend.
- **Feedback and State Handling:** Provides immediate visual feedback to the user. It displays a loading indicator during the registration process and shows clear error messages (e.g., "Email already in use") if the sign-up attempt fails.
- **Navigation to Login:** Includes a convenient link for users who already have an account to quickly navigate back to the Login Screen.

- **Immersive UI:** Like the Login Screen, it features a full-screen background image, a dark overlay, and an animated snow effect to create a consistent and engaging user experience. The form elements fade in smoothly on screen load.

- **Screenshots:**



- **Implementation details:**

- a. Widgets Used

The screen is constructed using a similar set of widgets as the LoginScreen to maintain a consistent look and feel.

- **Layout & Structure:** Scaffold, Stack, FadeTransition, Center, SingleChildScrollView, Form, Column, Row, SizedBox.
- **Display & Graphics:** Image.asset, Container (for overlays), Text, Icon, CircularProgressIndicator.
- **Input & Interaction:** TextFormField, ElevatedButton, GestureDetector, TextButton.

Special Widgets:

- **FadeTransition & AnimationController:** This combination is used to animate the opacity of the main form content, creating a smooth fade-in effect when the screen is first displayed.
- **SnowEffect:** This is a custom widget (also used on the LoginScreen) that renders an animated overlay of falling snowflakes, contributing to the app's immersive, cinematic theme. It is not a standard Flutter widget.

b. Libraries and Plugins

Yes, this feature would use the following key libraries and plugins:

- **go_router:** This library manages all navigation. On this screen, it is used for:
    - Navigating to the home screen (context.go('/home')) after a successful registration.
    - Navigating back to the login screen (context.go('/login')) if the user taps the "Log in" link.
- **provider:** This is the application's shared state management solution. It is used to access the AuthProvider and call the signUp() method: Provider.of<AuthProvider>(context, listen: false).signUp(...).
- **shared_preferences:** This plugin would be used after a successful sign-up to save the new user's email and password. This allows the app to pre-fill the login form for the user's convenience in future sessions.

c. Shared State Management

Yes, this feature uses a shared state management solution via the **provider** package.

- **How it works:** The SignUpScreen interacts with the central AuthProvider.
    - When the user submits the form, the screen calls the signUp(name, email, password) method from the AuthProvider instance obtained via Provider.of<AuthProvider>(context, listen: false). The listen: false flag is used because the screen is only dispatching an action and does not need to rebuild in response to changes in the AuthProvider.
    - The signUp() method within AuthProvider contains the business logic for creating a new user. This typically involves making an API call to a backend server.
    - Upon successful account creation, the AuthProvider updates its internal state to reflect the newly authenticated user. The SignUpScreen then navigates the user to the main part of the app (e.g., the home screen).
- **Code Architecture:** The AuthProvider is provided at the root of the widget tree (likely wrapping the MaterialApp). This ensures a single, consistent instance of the provider is available throughout the app for any widget that needs to perform authentication actions or check the user's session status.

d. Data Storage and API

This feature sends data to a remote API and writes data locally upon success.

- **Local Data (Write):** After a successful registration, the screen uses the shared_preferences plugin to store the new user's credentials locally on the device.
    - **Data Structure:** It writes two key-value pairs:
        - 'saved_email': (String) The new user's email.
        - 'saved_password': (String) The new user's password.
- **Remote Data (Write/API Call):** The core registration logic is handled by a remote API, which is called from within the AuthProvider.

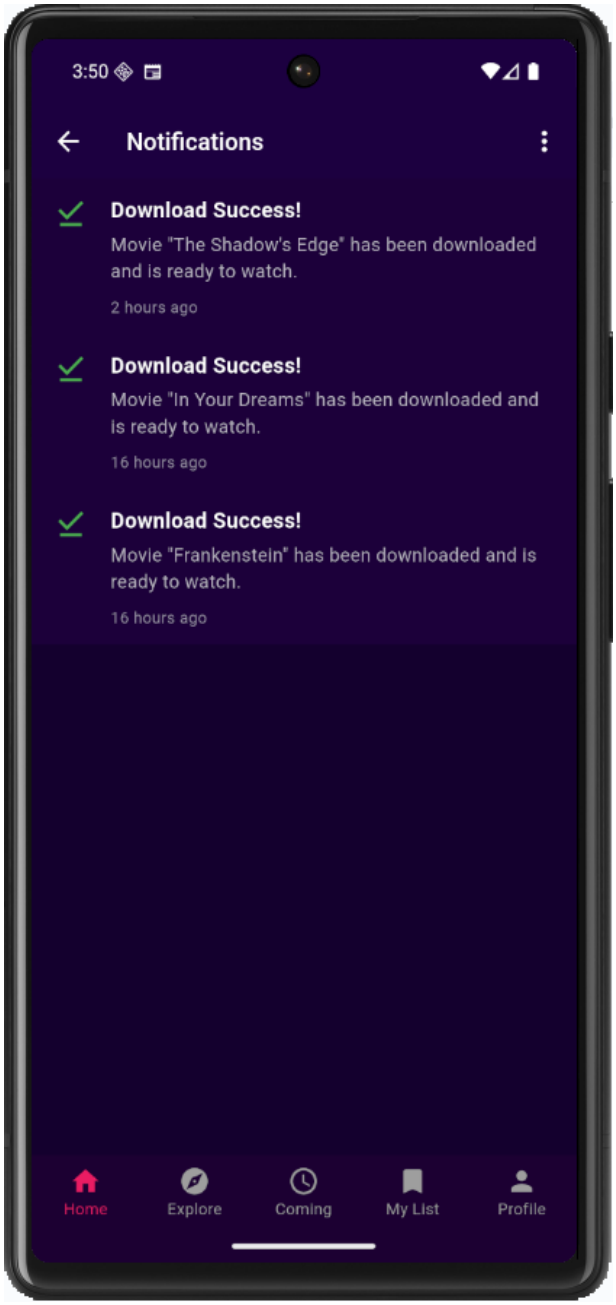
13. Notification Screen

Description: The Notification Screen acts as a centralized inbox for all alerts and updates within the PuTa Movies application. It is designed to keep users informed about relevant activities, new content, and system messages in a clear and organized manner.

Key Functions:

- **Notification Aggregation:** Displays a chronological list of all notifications received by the user.
- **Categorization:** Each notification is visually distinguished by a unique icon and color corresponding to its type (e.g., new release, download complete, system alert, comment reply).
- **Interaction and Navigation:** Tapping a notification marks it as read and navigates the user to the relevant content within the app (e.g., a movie details page).
- **Notification Management:** Provides powerful management tools, including:
    - **Swipe-to-Delete:** Users can dismiss individual notifications by swiping them to the left.
    - **Mark All as Read:** A menu option allows users to mark all unread notifications as read in a single action.
    - **Clear All:** A menu option, with a confirmation dialog, allows users to permanently delete all notifications.
- **Read/Unread State:** Unread notifications have a distinct background color to make them stand out, which disappears once they are read.
- **Relative Timestamps:** Displays how long ago a notification was received in a human-readable format (e.g., "5 minutes ago," "2 days ago").

- **Screenshots:**



- **Implementation details:**

a. Widgets Used

The screen is built using a combination of standard and interactive Flutter widgets.

- **Layout &**
  Structure: Scaffold, AppBar, ListView.builder, Center, Row, Column, Expand ed, SizedBox.

- **Interaction & Menus:** PopupMenuButton, PopupMenuItem, AlertDialog, TextButton, InkWell.
- **Display:** Text, Icon.

Special Widgets:

- **Dismissible:** This is a key widget for this screen. It wraps each notification list item, allowing the user to swipe it away to delete it. It provides the onDismissed callback to trigger the removal logic and a background property to show the red delete confirmation UI during the swipe gesture.

b. Libraries and Plugins

Yes, this feature relies on several important libraries and plugins.

- **provider:** This is the core state management library. It is used to access the NotificationProvider, which holds the list of notifications and the business logic for managing them (adding, removing, marking as read, etc.).
- **go_router:** This library handles all navigation. When a user taps a notification, context.push('/movie/${notification.movieId}') is called to navigate to the corresponding movie details screen.
- **timeago:** This plugin is used to format the DateTime timestamp of each notification into a user-friendly, relative string (e.g., "a moment ago", "2 hours ago").

c. Shared State Management

Yes, this feature uses a shared state management solution via the **provider** package.

- **How it works:**
    - The NotificationListScreen uses context.watch<NotificationProvider>() to listen for changes to the list of notifications. Whenever a notification is added, removed, or its read status changes in the provider, the screen automatically rebuilds to reflect the new state.
    - User actions trigger methods on the provider using context.read<NotificationProvider>(). For example:

        - Swiping a Dismissible calls provider.removeNotification(id).
        - Tapping an item calls provider.markAsRead(id).
        - Selecting from the menu calls provider.markAllAsRead() or provider.clearAllNotifications().

- **Code Architecture:** The NotificationProvider is provided at a high level in the application's widget tree (likely above the MaterialApp). This makes a single instance of the provider available to any part of the app. Other features can add notifications to this provider (e.g., a download service, a background fetch service), and the NotificationListScreen will automatically display them, creating a decoupled and scalable architecture.

d. Data Storage and API

This feature reads and stores data locally, managed entirely by the NotificationProvider. It does not make any direct REST API calls.

- **Local Data:** The NotificationProvider is responsible for persisting notifications so they are not lost when the app closes. While the code for the screen doesn't show the implementation, this persistence is typically achieved using:
    - **shared_preferences:** Storing the list of notifications as a single JSON string. This is simple but can be inefficient for large lists.
    - **A local database (e.g., SQLite):** A more robust solution where each notification is a row in a notifications table. This is better for performance and querying.
- **Data Table Structure (assumed for SQLite):** A notifications table would likely have the following structure:
    - id (TEXT, PRIMARY KEY) - A unique identifier for the notification.
    - title (TEXT) - The main title of the notification.
    - body (TEXT) - The descriptive body text.
    - timestamp (TEXT/INTEGER) - The date and time the notification was created.
    - isRead (INTEGER, 0 or 1) - A flag to track if the user has read it.

    - type (TEXT) - A string representing the notification type (e.g., 'download', 'reply').
    - movieId (INTEGER, NULLABLE) - An optional ID to link the notification to a specific movie or TV show.

- **API Interaction:** This screen does not directly call any REST APIs. Notifications are generated by other services within the app and are added to the NotificationProvider. The screen's only job is to display the data held by the provider.

14. Cast Detail Screen

Description: The Actor Detail Screen provides a comprehensive and visually rich profile for any actor or actress in the application's database. It serves as a deep-dive into their career, offering users detailed biographical information and a complete filmography.

Key Functions:

- **Detailed Profile:** Displays the actor's name and large profile picture in a collapsing app bar for a cinematic effect.
- **Key Statistics:** Presents important stats at a glance, including their popularity score, gender, and current age (calculated from their birthday).
- **Biography:** Shows the actor's full biography. For longer bios, the text is initially truncated with a "Read More" button to expand it, ensuring a clean layout.
- **Filmography:** Organizes the actor's work into "Movies" and "TV Shows" using a tabbed interface. Each tab contains a horizontally scrollable list of relevant movie/show posters.
- **Navigation:** Users can tap on any movie or TV show in the filmography to navigate directly to its respective detail page.
- **Loading & Error States:** Provides a smooth user experience by showing an animated "shimmer" placeholder while data is being fetched and a clear error message with a "Try Again" button if the data fails to load.

- **Screenshots:**

- **Implementation details:**

- a. Widgets Used

The screen is built using a combination of standard and advanced layout widgets to create its dynamic scrolling effect.

- **Layout &**
  Structure: Scaffold, CustomScrollView, SliverAppBar, FlexibleSpaceBar, Sli verToBoxAdapter, SliverFillRemaining, Column, Row, Padding, Center.
- **Display:** Text, Icon, CachedNetworkImage, Divider.
- **Interaction &**
  Navigation: IconButton, ElevatedButton, TextButton, TabBar, TabBarView, T abController.

- **Custom Widgets:** RelatedMoviesList (a reusable widget to display a horizontal list of movies/shows), ActorDetailPlaceholder (a custom placeholder widget for the loading state).

Special Widgets:

- **CustomScrollView with SliverAppBar:** This is the core widget combination that creates the signature collapsing header effect. The SliverAppBar holds the actor's image and name in its flexibleSpace and smoothly transitions from a large, expanded header to a standard, pinned app bar as the user scrolls.
- **Shimmer.fromColors:** This widget, from the shimmer package, is used within the ActorDetailPlaceholder to create an animated, shimmering effect over the placeholder layout, providing a more polished loading experience than a simple spinner.

b. Libraries and Plugins

Yes, this feature relies on several key libraries and plugins.

- **provider:** This is the primary state management library. It is used to access the ActorDetailProvider, which is responsible for fetching, caching, and providing the actor's detailed information.
- **go_router:** This library is used for navigation. context.pop() is called to close the screen, and tapping on a movie in the filmography list will use context.push() to navigate to that movie's detail page.
- **cached_network_image:** This plugin is used to efficiently load and cache the actor's profile picture from a URL, improving performance and reducing network usage.
- **shimmer:** This plugin provides the Shimmer widget used to create the elegant loading placeholder effect.
- **intl:** The Internationalization library is used for date formatting. DateFormat.yMMMMd() is used to format the actor's birthday into a readable string (e.g., "January 1, 1980").

c. Shared State Management

Yes, this feature uses a shared state management solution via the **provider** package.

- **How it works:**

- The screen depends on an ActorDetailProvider. In initState, it dispatches an action to fetch the data: context.read<ActorDetailProvider>().fetchActorDetails(widget.actorId).
- The main body of the build method is wrapped in a Consumer<ActorDetailProvider>. This allows the widget to listen for changes in the provider. When the provider's state changes (e.g., from loading to loaded, or to an error state), the Consumer rebuilds the UI to display the appropriate content (placeholder, error message, or actor details).

- **Code Architecture:** The ActorDetailProvider is likely provided at a high level in the widget tree (e.g., above the MaterialApp). It maintains an internal cache (likely a Map<int, ActorDetail>) of actor details. When fetchActorDetails is called, it first checks if the data for the given actorId is already in the cache. If so, it returns the cached data immediately. If not, it fetches the data from the API, stores it in the cache, and then notifies its listeners (like the ActorDetailScreen) to rebuild.

d. Data Storage and API

This feature reads data remotely from The Movie Database (TMDB) API. It does not store data locally beyond the in-memory cache of the provider.

- **REST API:** The ActorDetailProvider communicates with the TMDB API.
  - **API Call:** It makes an HTTP GET request to the /person/{person_id} endpoint.
  - **Endpoint URL (Example):** https://api.themoviedb.org/3/person/1234
  - **Input (Query Parameters):**
    - api_key: The secret API key for authentication.
    - append_to_response=movie_credits,tv_credits: This is a crucial optimization. It tells the TMDB API to include the actor's movie and TV show credits directly within the main response, avoiding the need for separate API calls.
  - **Output (JSON Response):** The API returns a JSON object that is parsed into the ActorDetail model. The structure includes:

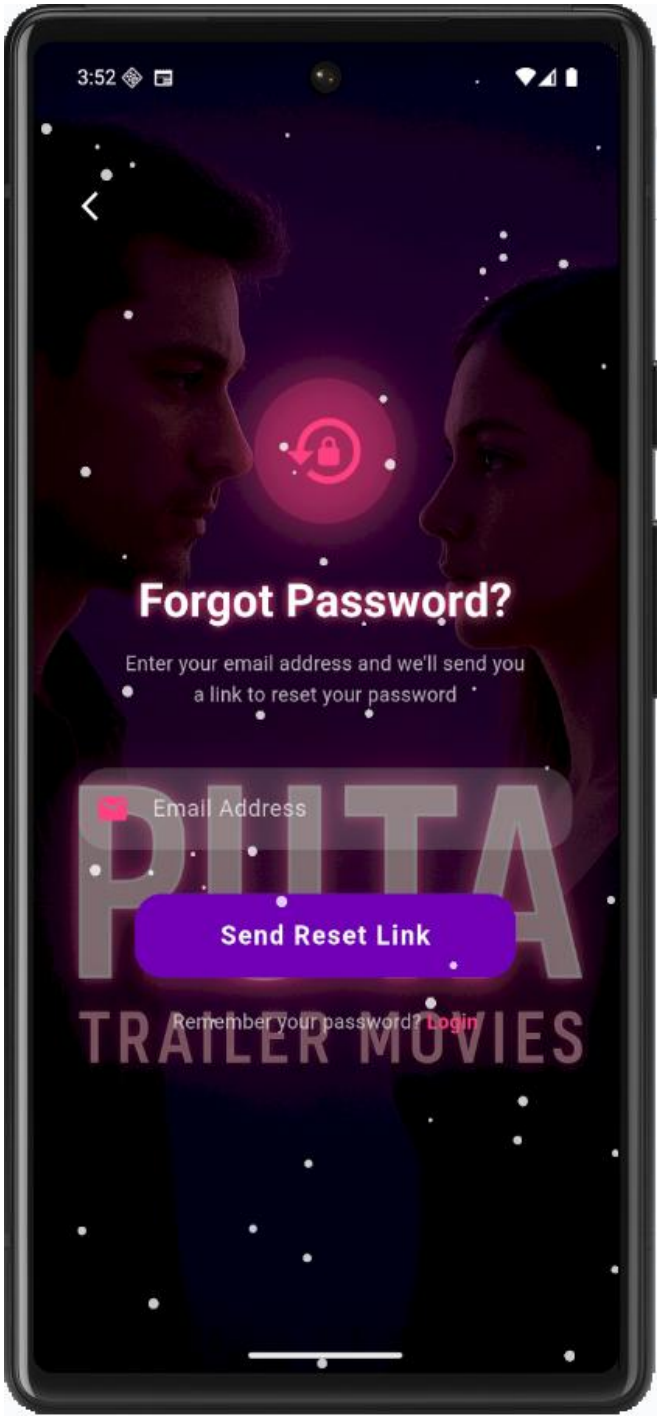
15. Forgot Password Screen:

Description: The "Forgot Password" screen provides a crucial utility for users who have lost access to their accounts. It offers a simple and secure way to initiate the password recovery process.

Key Functions:

- **Password Reset Initiation:** Allows users to enter their registered email address to receive a password reset link.
- **Input Validation:** The email field includes validation to ensure the user enters a correctly formatted email address before submitting the request.
- **User Feedback:** The screen provides clear visual feedback throughout the process. It shows a loading indicator while the request is being processed and displays a success message via a snackbar upon completion.
- **Seamless Navigation:** An easily accessible back button allows users to return to the previous screen. It also includes a link to navigate directly to the "Login" screen for users who might have remembered their password.
- **Immersive UI:** The screen maintains the application's cinematic theme with a full-screen background image, a dark overlay, and an animated snow effect. The form elements fade in smoothly when the screen loads.

- **Screenshots:**



- **Implementation details:**

a. Widgets Used

The screen is built using a combination of standard and custom widgets to create its animated and user-friendly interface.

- **Layout & Structure:** Scaffold, Stack, Center, SingleChildScrollView, Form, Column, Row, SizedBox, Align, SafeArea.
- **Display & Graphics:** Image.asset, Container (for overlays and decoration), Text, Icon, CircularProgressIndicator.
- **Input & Interaction:** TextFormField, ElevatedButton, IconButton, GestureDetector, TextButton.

Special Widgets:

- **FadeTransition:** This widget is used in conjunction with an AnimationController to create a smooth fade-in animation for the entire screen content when it first appears.
- **SnowEffect:** This is a custom, non-standard widget used to render an animated overlay of falling snowflakes, enhancing the screen's visual theme and creating a more dynamic user experience.

b. Libraries and Plugins

Yes, this feature uses the following libraries:

- **go_router:** This library is used for all navigation on this screen. It handles:
    - Navigating back to the previous screen when the back button is pressed (context.pop()).
    - Navigating to the login screen after a successful password reset request or when the "Login" link is tapped (context.go('/login')).
- **ui_helpers.dart (Internal Utility):** This is a custom utility file within the project. Its role on this screen is to provide a standardized way to show user feedback, specifically by calling UIHelpers.showSuccessSnackBar() to display the confirmation message.

c. Shared State Management

No, this feature **does not use a shared state management solution** like Provider or Riverpod.

- **How it works:** The ForgotPasswordScreen is a StatefulWidget that manages its own state locally.
    - It uses a local boolean variable _isLoading to control the visibility of the loading indicator.
    - The state is updated exclusively through setState(). For example, setState(() { _isLoading = true; }) is called at the beginning of the _resetPassword method, and setState(() { _isLoading = false; }) is called when it completes.
- **Code Architecture:** This self-contained approach is suitable for this screen because its functionality is simple and its state (like the loading status) does not need to be accessed or modified by any other part of the application.

d. Data Storage and API

This feature simulates sending data to a remote API but does not actually perform a network request in its current implementation.

- **API Interaction (Simulated):**
    - The _resetPassword function is designed to mimic an API call. It takes the email from the _emailController as input.
    - Instead of making a real HTTP request, it uses await Future.delayed(const Duration(seconds: 2)); to simulate the network latency of a server request.
- **API Description (Hypothetical):** If this were a real feature, the _resetPassword method would make a POST request to a backend endpoint.
    - **Endpoint (Example):** https://api.yourapp.com/auth/forgot-password
    - **Input (Request Body):** A JSON object containing the user's email.