FINAL PROJECT REPORT

SEMESTER 1, ACADEMIC YEAR: 2025-2026

CT312H: MOBILE PROGRAMMING

- **Project/Application name: Music App**
- **GitHub link: https://github.com/25-26Sem1-Courses/ct312hm01-project-GiaVinh2404.git**
- **Link Video Demo Feature: https://www.youtube.com/watch?v=7k5cDgvDjXk&t=12s**
- **Student ID 1: B2206023**
- **Student Name 1: Dương Thế Vĩ**
- **Student ID 2: B2207579**
- **Student Name 2: Ngô Gia Vinh**
- **Class/Group Number: CT312HM01/Group 19**

I. Introduction

Project Title: Music App (Flutter & PocketBase)

Overview: The Music App is a robust, cross-platform mobile application designed for seamless music streaming and personal library management. Built with Flutter, the application delivers a high-performance. It features a modern Material Design interface that supports dynamic theming (Dark/Light mode) and responsive layouts for various screen sizes.

Technical Architecture: The project is engineered using Clean Architecture, ensuring a strict separation of concerns between the Domain, Data, and Presentation layers. This structure guarantees scalability, maintainability, and testability.

- **State Management:** Implements the BLoC (Business Logic Component) pattern for reactive UI updates and Hydrated BLoC for persisting

application states locally (e.g., keeping user theme preferences or cached data).

- **Dependency Injection:** Utilizes GetIt for efficient service location and dependency management across the app lifecycle.
- **Backend Integration:** Integrates PocketBase as a real-time backend solution to handle authentication, database storage, and file hosting.
- **Error Handling:** Adopts functional programming concepts using the dartz library (Either<Failure, Success>) to manage exceptions and data flow securely.

Key Features:

- **Advanced Audio Engine:** Powered by just_audio and just_audio_background, the app supports background playback, lock screen controls, notification management, and complex queue handling (Shuffle, Loop One, Loop All).
- **User Authentication:** Secure Sign-up, Sign-in, Profile management, and Password reset functionality.
- **Smart Discovery:** Users can browse "News Songs", view Artist details, and search for both tracks and artists via an optimized search logic.

Personalization & Social:

- **Playlist Management:** Full CRUD capabilities allow users to create custom playlists, and add/remove tracks in this.
- **Interactions:** Features "Favorite Songs" and "Follow Artists" functionality to tailor the listening experience.
- **Smooth Navigation:** Implements go_router for deep linking and managing navigation stacks effectively.

A task assignment sheet for each member.

No.	Feature & Page	Estimated Complexity	Responsible Member	Basis for Assignment
I	Core Architecture & Infrastructure	High	Ngô Gia Vinh	Setting up the foundational architecture, DI, and core backend integration.
1	Architectural Overview (Clean Architecture, BLoC, GetIt)	High	Ngô Gia Vinh	Building the initial framework, ensuring scalability, maintainability, and testability.
2	Backend: PocketBase Integration (Auth, DB, File Hosting)	High	Ngô Gia Vinh	Integrating the real-time backend solution.
3	Error Handling with dartz (Either<F, S>)	High	Ngô Gia Vinh	Implementing functional programming concepts for secure data flow.
II	User & Library Management Features		Dương Thế Vĩ	Focus on user data modification, personal library management, and core playback engine.
4	Profile Page	Medium	Dương Thế Vĩ	Displaying user info, followed artists, and favorite songs.
5	Edit Profile Page	High	Dương Thế Vĩ	Handling file upload (Avatar), updating name, and complex state interaction between EditProfileCubit and ProfileInfoCubit.
6	Change Password Page	Medium	Dương Thế Vĩ	Secure password update with form validation and dedicated Cubit.
7	Playlist Detail Page (View & CRUD)	High	Dương Thế Vĩ	Complex logic: Ownership check, full CRUD capabilities, song removal, and managing two Cubits (DetailsCubit, MyPlaylistsCubit).
8	Edit Playlist Dialog	Medium	Dương Thế Vĩ	Local state management for input, file selection (Cover), and dispatching updates to the shared MyPlaylistsCubit.
9	Advanced Audio Engine	High	Dương Thế Vĩ	Configuring and deploying just_audio and just_audio_background for background playback, lock screen controls, and queue management.
III	Authentication & Discovery Features		Ngô Gia Vinh	Focus on the core authentication flow, thematic control, and music discovery pages.
10	Choose Mode Screen (Theme Control)	Low	Ngô Gia Vinh	Managing theme switching using the BLoC pattern and Hydrated BLoC for persistence.
11	Sign In Screen	Medium	Ngô Gia Vinh	Implementing user login with PocketBase Authentication (authWithPassword).
12	Sign Up Screen	Medium	Ngô Gia Vinh	Implementing user registration with PocketBase Authentication (create user).
13	Reset Password Page	Medium	Ngô Gia Vinh	Handling password reset request asynchronously, providing feedback via SnackBar, and using BlocListener.
14	Artist Detail Page	High	Ngô Gia Vinh	Implementing the Follow/Unfollow toggle feature with instant local state update and remote persistence.
15	Album Detail Page	Medium	Ngô Gia Vinh	Managing multiple data fetches concurrently using Future.wait for metadata and tracks.

16	Search Page	High	Ngô Gia Vinh	Implementing the Debouncer mechanism for optimized real-time search, using a tabbed interface for Songs and Artists results.
17	Home Page (Including NewsSongs/ArtistsList)	High	Ngô Gia Vinh	Hybrid State management, synchronizing TabBar/TabBarView, and using UniqueKey() to force playlist refresh.

Summary of Division:

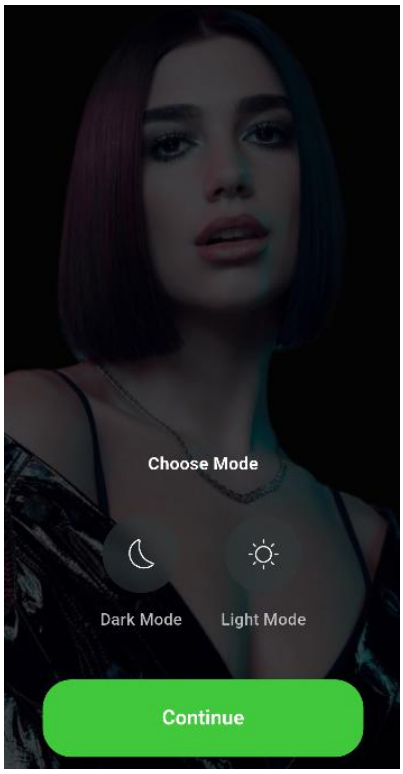
- **Ngô Gia Vinh (B2207579):** Focuses on Architectural Foundation, User Authentication (Sign In/Up, Reset, Theme Control), and Discovery Features (Home Page, Search, Artist Detail, Album Detail).
- **Dương Thế Vĩ (B2206023):** Focuses on Account Management (Profile, Edit Profile, Change Password), Personal Library Management (Playlist Detail, Edit Playlist), and the Core Audio Playback System (Just Audio Engine).

II. Details of implemented features

1. Feature / Application page 3: Choose Mode Screen

- **Description:** The Sign Up Screen allows new users to create an account in the Music App by providing their full name, email address, and password. This feature is essential for personalizing the music experience, enabling users to create playlists, save favorite songs, and maintain playback history tied to their account. The screen uses a simple and responsive design, with a clear form layout, input validation (in future integration), and navigation to the Sign In page for returning users.

- **Screenshots:** some screenshots of this feature/application page.



- **Implementation details:** students should answer the following questions:
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scafford	Defines the page layout, containing an app bar, body, and footer navigation.
Stack	Overlays multiple layers: background image, opacity layer, and content.
SafeArea	Make sure the content (Logo, Text, Button) is not obscured by the notch, status bar or bottom navigation bar on new phone models.

Container	Displays the background image and adds padding.
BoxDecoration & DecorationImage	Used to apply and style the background image.
BlocBuilder	This is the most important widget to connect UI with Logic. It listens to changes from ThemeCubit to know if the user is choosing Dark or Light mode, then redraws the interface (changes border color, icons) accordingly.
ClipOval	Crop the blur effect to a circle. Without this Widget, the BackdropFilter blur effect would spread across the entire screen or rectangle, not be confined to the circle of the button.
BackdropFilter	Creates the blurred glass effect for the mode selection buttons
SvgPicture.asset	Renders SVG icons (sun and moon) and logo using the flutter_svg plugin.
Column, Row, SizedBox, Padding, Align, Spacer	Organize and position UI elements with proper spacing.
GestureDetector	Detects user taps on each mode option to update the theme.
Text	Displays titles and labels like “Choose Mode”, “Dark Mode”, and “Light Mode”.

BasicAppButton	Custom reusable button widget used for the “Continue” action.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins.

Library / Plugin	Purpose
flutter_svg	Load and display SVG vector graphics for the app logo.
flutter/material.dart	Core Flutter UI library for building Material Design components.
flutter_bloc	Provides the BLoC architecture; used here to manage theme switching through ThemeCubit.
dart:ui	Enables the use of ImageFilter.blur for the blur effect (BackdropFilter).

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This feature uses **BLoC (Cubit) state management** for theme control:

  - The app defines a ThemeCubit that extends Cubit<ThemeMode>.
  - When the user taps on a mode icon, context.read<ThemeCubit>().updateTheme(ThemeMode.dark) or ThemeMode.light is triggered.

- The Cubit updates the global app theme instantly, allowing consistent theme appearance across all pages.

Workflow:

- User opens the Choose Mode page.
- User taps on Dark Mode or Light Mode → ThemeCubit updates the theme.
- The app's theme changes visually in real time.
- User presses Continue → navigates to the Sign In / Sign Up screen.

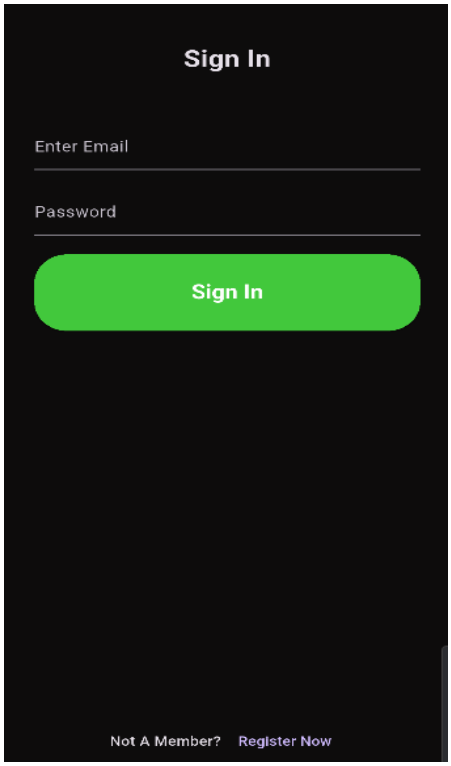
- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  This feature does not read or store user data remotely.

2. Feature / Application page 1: Sign in Screen

- **Description:** The Sign In Screen allows existing users to securely log into the Music App using their registered email and password. It provides a clean, minimal, and user-friendly interface following Material Design guidelines, with fields for credentials and navigation to the registration page for new users. This page acts as the authentication gateway, ensuring that only authorized users can access personalized features such as playlists, favorites, and playback history.

- **Screenshots:** some screenshots of this feature/application page.



- **Implementation details:** students should answer the following questions:
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Align	Aligns the "Forgot Password" button to the right side of the screen.
Text	Displays static text labels like "Sign In", "Not A Member?", and button titles.
SnackBar	Displays error messages (e.g., invalid credentials) at the bottom of the screen.

Scafford	Provides the overall page layout, including the AppBar and BottomNavigationBar.
BasicAppbar	Displays the app logo and provides consistent styling across screens.
SvgPicture.asset	Loads and renders the vector-based app logo from assets using the flutter_svg plugin.
SingleChildScrollView	Allows the page to scroll when the keyboard appears.
Column, SizedBox, Padding	Layout organization and spacing.
TextField	Used for entering the user's email and password.
BasicAppButton	Reusable button widget used for the Sign In action.
Row and TextButton	Used in the footer for navigation to the Sign Up screen.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins.

Library / Plugin	Purpose
flutter_svg	Load and display SVG vector graphics for the app logo.
flutter/material.dart	Core Flutter UI library for building Material Design components.
go_router or Navigator API	Handles navigation between the Sign In and Sign Up pages.

firebase_auth	Used to authenticate user credentials with the backend (email + password).

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This page currently uses local state via TextEditingController for managing input fields (_email, _password).
  When integrated into the full authentication flow, the Sign In page will connect to a AuthBLOC that handles:

  - Dispatching a LoginEvent(email, password) when the user presses Sign In.
  - Listening to the result from Firebase/MySQL and emitting either AuthSuccess or AuthFailure.
  - Navigating to the Home Page upon successful login.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  This feature use Pocketbase Authentication with Method:

  client.pb.collection('users').authWithPassword(email, password)

  Input:

  email (String)

  password (String)

  Output (on successful):

{

"uid": "4R3tG9sM7v2...",

"email": "user@example.com"

}

Output (on failure):

{

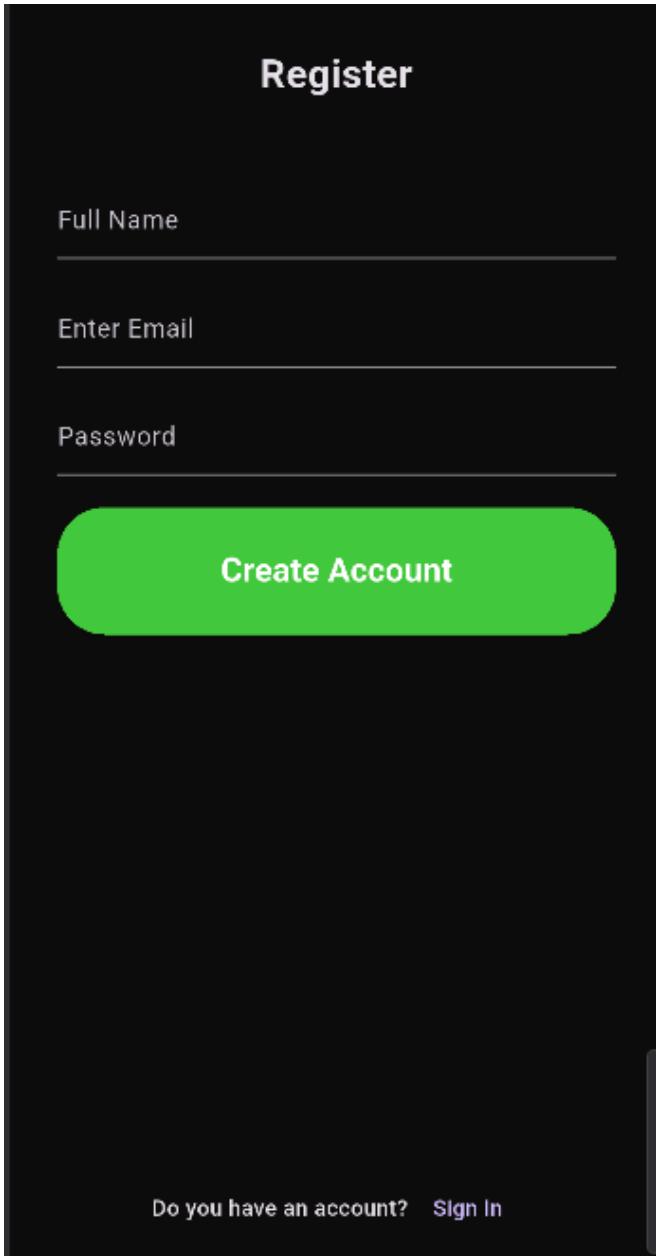
"error": "Invalid credentials"

}

3. **Feature / Application page 2: Sign up Screen**

- **Description:** The Sign Up Screen allows new users to create an account in the Music App by providing their full name, email address, and password. This feature is essential for personalizing the music experience, enabling users to create playlists, save favorite songs, and maintain playback history tied to their account. The screen uses a simple and responsive design, with a clear form layout, input validation (in future integration), and navigation to the Sign In page for returning users.

- **Screenshots:** some screenshots of this feature/application page.



- **Implementation details:** students should answer the following questions:
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose

Scafford	Defines the page layout, containing an app bar, body, and footer navigation.
BasicAppbar	Displays the app logo and provides consistent styling across screens.
SvgPicture.asset	Loads and renders the vector-based app logo from assets using the flutter_svg plugin.
SingleChildScrollView	Allows the page to scroll when the keyboard appears.
Column, SizedBox, Padding	Layout organization and spacing.
TextField	Used for entering the user's email and password.
BasicAppButton	Reusable button widget used for the Sign In action.
Row and TextButton	Used in the footer for navigation to the Sign Up screen.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins.

Library / Plugin	Purpose
flutter_svg	Load and display SVG vector graphics for the app logo.

flutter/material.dart	Core Flutter UI library for building Material Design components.
go_router or Navigator API	Handles navigation between the Sign In and Sign Up pages
firebase_auth	Used for creating a new user account in Firebase

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen currently uses local state through TextEditingController for handling form inputs.
  In the complete authentication flow, it can be integrated with a BLoC or Cubit structure such as AuthBloc, which manages registration logic.
  Workflow:

  - User enters full name, email, and password.
  - When “Create Account” is pressed, the BLoC dispatches a RegisterEvent.
  - The BLoC calls the authentication repository to send the data to the backend.
  - The state is updated based on the result (RegisterSuccess, RegisterFailure).

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Pocketbase Authentication

  - Method: await client.pb.collection('users').create(
    body: {
    'email': email,
    'password': password,
    'passwordConfirm': password
    'name': fullName,
    'emailVisibility': true,
    },
    );
  - Input Fields:

    full_name (String)

    email (String)

    password (String)
  - Output (on success):

    {

    "uid": "t7Ys9ZK4rFvW...",

    "email": "newuser@example.com"

    }
  - Ouput (on failure):

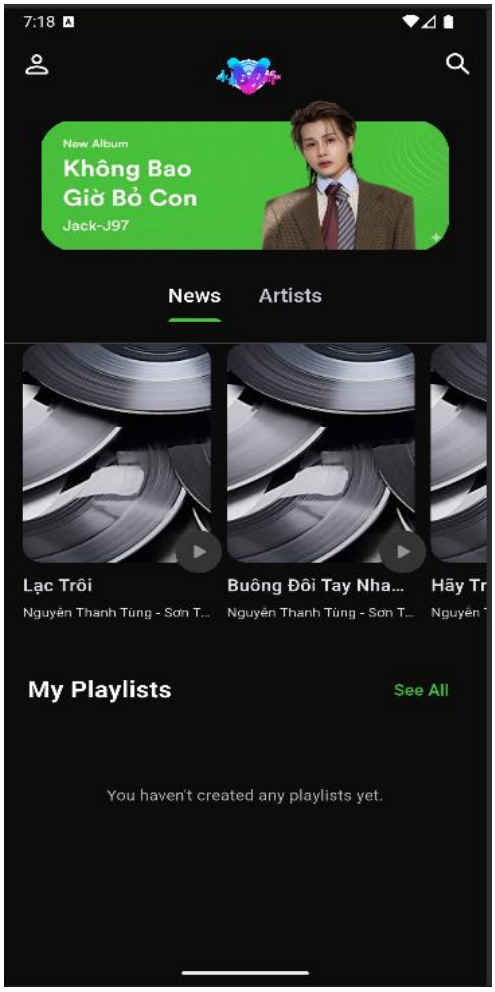
    {

    "error": "Email already exist"

}

4. **Feature / Application page: HomePage**

- **Description:** The Home Page is the central hub of the Music App, offering a streamlined interface for users to discover new music, explore artists, and access their personal library. It is designed with a vertical scrollable layout (SingleChildScrollView) containing dynamic sections that adapt to the user's theme preference (Light/Dark mode)
- **Screenshots:**



- **Implementation details:** students should answer the following questions:
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure (AppBar + Body).
AppBar	The top navigation bar containing the Profile icon, App Logo, and Search button.
SingleChildScrollView	Allows the entire page body to scroll vertically to fit content on smaller screens.
TabBar	Displays the tab switches ("News" and "Artists") allowing users to toggle views.
TabBarView	Renders the dynamic content (NewsSongs or ArtistsList) corresponding to the selected tab.
Stack & Align	Overlays multiple layers (Background Vector + Artist Image) to create the _homeTopCard banner.
SvgPicture.asset	Renders vector-based graphics (Logo, Top Card background) using flutter_svg.
NewsSongs/ ArtistsList	Custom Widgets used to display the list of songs and artists inside the tabs.
PlaylistsPreview	Custom Widget used to display the user's personal playlists.
IconButton	Clickable icons used for navigation (Profile, Search).
Column/Row/SizedBox	Layout organization, spacing, and alignment of elements.

There are special widgets/techniques used in this page that might be considered advanced or external:

TabBar & TabBarView (with TabController):

- **Why special:** Unlike basic navigation, this implements a synchronized tab system. It requires the class to mix in SingleTickerProviderStateMixin to manage the animation state of the tabs.

SvgPicture (External Package):

- **Why special:** This is not a core Flutter widget but part of the flutter_svg package, used to render high-quality vector graphics instead of standard bitmap images.

UniqueKey (State Management Technique):

- **Why special:** In the _refreshPlaylistPreview function, UniqueKey() is assigned to the PlaylistsPreview widget. This is a specific technique to force Flutter to destroy and rebuild the widget entirely (refreshing the data) when the user returns from the "See All" screen.
    - Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_svg	Loads and displays SVG vector graphics for the App Logo and the Home Top Card background.
flutter/material.dart	Core Flutter UI library for building Material Design components like Scaffold, AppBar, and TabBar.
go_router	Handles navigation to the Search and Profile pages using context.push.
pocketbase	(Replacement for firebase_auth) Used to fetch data for Songs, Artists, and User Playlists from the backend database.

flutter_bloc	Manages the state of the UI, specifically for adapting the interface theme (Light/Dark mode) and handling data in sub-widgets.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses a **Hybrid State Management** approach. It utilizes **Local State** (via TabController and setState) for UI animations and manual refreshing, while relying on **Shared State Management** (BLoC/Cubit) for handling dynamic data fetching.

  It is integrated with multiple Cubits such as NewsSongsCubit and ArtistsListCubit, which manage the logic for loading songs and artists respectively.

  Workflow:

  - **Initialization:** When the Home Page loads, the integrated Cubits (NewsSongsCubit, ArtistsListCubit) automatically trigger events to fetch data.
  - **Data Request:** The Cubits call the respective Repositories (SongRepository, ArtistRepository) to retrieve the latest songs and artists from the PocketBase backend.
  - **State Update:** The UI listens to state changes from these Cubits to update the interface accordingly (e.g., showing a loading spinner, displaying the list of items upon Loaded state, or showing an error message upon Failure).
  - **User Interaction:** Local interactions like switching tabs are handled by TabController, while refreshing the playlist preview uses setState to rebuild the widget with a new UniqueKey.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Read): The Home Page fetches data for **Songs** (New Releases), **Artists**, and **User Playlists** from the PocketBase backend.

  Locally (Store/Read): It reads and persists the user's preferred **Theme Mode** (Dark/Light) using HydratedBloc storage.

  The application interacts with the backend via the **PocketBase Dart SDK**, which acts as a wrapper for the REST API.

  Fetching New Songs

  - SDK Call: client.pb.collection('songs').getList(sort: '-created', limit: 5).
  - Purpose: Retrieves the most recently added songs.
  - Input:
    - sort: '-created' (Sorts by creation date descending to get the newest songs).
    - limit: 5 (Retrieves only the top 3 newest tracks).
  - Output: A list of JSON records mapped to SongEntity (fields: id, title, artist, duration, audioUrl, coverUrl).

  Fetching Artists

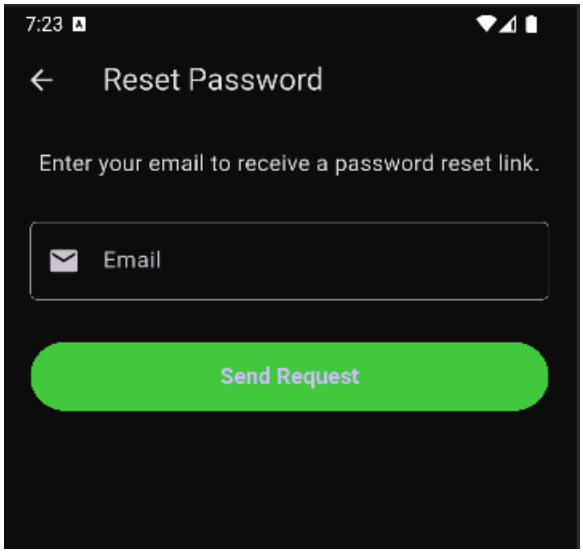
  - SDK Call: client.pb.collection('artists').getList().
  - Purpose: Retrieves the list of available artists.
  - Input: None
  - Output: A list of JSON records mapped to ArtistEntity (fields: id, name, bio, image).

  Fetching User Playlists

- SDK Call: client.pb.collection('playlists').getList(filter: 'user_id = "CURRENT_USER_ID"').
- Purpose: Retrieves playlists created by the logged-in user.
- filter: A string query matches the user_id field with the currently logged-in user's ID.
- Output: A list of PlaylistEntity records.

5. **Feature / Application page: Reset Password Page**

- **Description:** The Reset Password Page allows users to request a password reset link by entering their email address. It is a simple, stateless-looking page built around a Form and uses the Bloc/Cubit state management pattern to handle the asynchronous process of sending the request and providing immediate feedback (success/failure) via a SnackBar
- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the basic page structure (AppBar + Body).
AppBar	The top bar, featuring the title 'Reset Password' and providing the back navigation (via context.pop()).
Padding	Provides spacing around the main form content.
Form	Wraps the input fields to enable validation via _formKey.
Column	Arranges the prompt text, TextFormField, and button vertically.
TextFormField	Input field for the user's email, including input validation logic.
ElevatedButton	The 'Send Request' button, which triggers the password reset logic upon validation.
BlocProvider	Provides the ResetPasswordCubit instance to the widget tree.
BlocListener	Special Widget: Listens to state changes (ResetPasswordSuccess/Failure) to display non-UI-blocking feedback (SnackBar) and navigate back (context.pop()).
BlocBuilder	Listens to state changes (ResetPasswordLoading) to dynamically switch the button with a CircularProgressIndicator.
CircularProgressIndicator	Special Widget: Displays a loading animation when the reset request is being processed.
SnackBar	Special Widget: Displays transient success or error messages at the bottom of the screen.
GoRouter	Special Widget: The context.pop() call uses the go_router package for navigation, returning the user to the previous screen.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter/material.dart	Core Flutter UI library for standard widgets like Scaffold, TextFormField, and ElevatedButton.
flutter_bloc	State management solution used to handle the asynchronous reset request lifecycle (Loading -> Success/Failure) and update the UI accordingly.
go_router	Handles the navigation: used for context.pop() to return to the login screen upon successful request.
dart:async	Used implicitly within the Cubit for asynchronous operations (Future<void>).
service_locator.dart	(External file) Used to resolve the dependency injection (sl<ResetPasswordUseCase>) when creating the ResetPasswordCubit.
pocketbase (Implicit)	The underlying data source/library likely used within the ResetPasswordUseCase to communicate with the backend's authentication API.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses the BLoC/Cubit pattern for state management. It utilizes a dedicated ResetPasswordCubit to manage the UI logic flow.

  Cubit Architecture (reset_password_cubit.dart):

  - **States:**
    - ResetPasswordInitial: Trạng thái ban đầu.

    - ResetPasswordLoading: Hiển thị khi request đang được gửi đi.
    - ResetPasswordSuccess: Yêu cầu gửi email thành công.
    - ResetPasswordFailure: Yêu cầu thất bại, chứa thông báo lỗi (message).
- **Method:**
    - sendResetEmail(String email): Chuyển sang trạng thái Loading, gọi _useCase (sử dụng ResetPasswordUseCase), và sau đó phát ra trạng thái Success hoặc Failure dựa trên kết quả.

Workflow (reset_password.dart):

- The ResetPasswordPage is wrapped in a BlocProvider to create and provide the ResetPasswordCubit.
- Upon button press, the Form is validated. If valid, context.read<ResetPasswordCubit>().sendResetEmail(...) is called.
- The BlocBuilder listens for ResetPasswordLoading state to show the CircularProgressIndicator instead of the button.
- The BlocListener listens for ResetPasswordSuccess or ResetPasswordFailure to:
    - Show a green SnackBar and navigate back (context.pop()) on Success.
    - Show a red SnackBar with the error message on Failure.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  This feature primarily writes/sends a request remotely and reads the response status.

Remotely (Write/Send): The Reset Password Page sends an email address to the backend to initiate the password reset process. It reads the response status (success or failure) from the API.

Locally (Store/Read): No explicit local data storage (like preferences or cache) is used directly by this feature.

The application interacts with the backend (implicitly using the PocketBase Dart SDK) to request the password reset functionality.

Sending Password Reset Request:

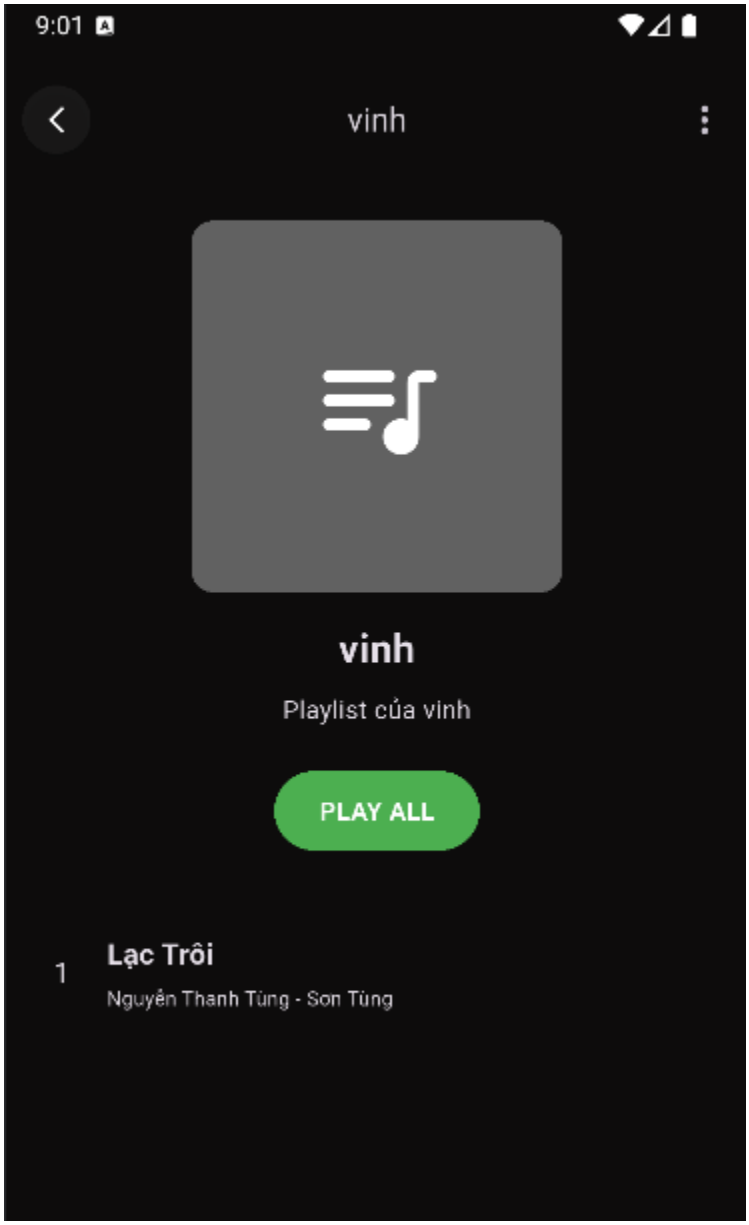
- SDK Call (Abstracted): _useCase(params: email)
- PocketBase Endpoint (Implicit): client.pb.authStore.requestPasswordReset(email)
- Purpose: Sends a request to the server to generate a unique reset token and email it to the user's provided address.
- Input: email (a single string containing the user's Email address).
- Output:
    - Success: The API confirms the email request was processed (often status code 200/204, no data body). Mapped to ResetPasswordSuccess.
    - Failure: The API returns an error message (e.g., "User not found," "Email not verified"). Mapped to ResetPasswordFailure(message).

6. **Feature / Application page: Playlist Detail Page**

- **Description:** The Playlist Details Page is designed to display the comprehensive details of a specific music playlist identified by its ID. It features a large header for the playlist metadata (cover, name, description, owner) and a scrollable list of all contained songs. The page supports core music interactions (Play All, Play

Individual Song) and conditional CRUD (Create, Read, Update, Delete) actions, allowing only the playlist owner to Edit, Delete, or Remove songs from the list.

- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose

Scaffold	Defines the page layout structure (AppBar + Body).
BasicAppbar	Custom widget serving as the top navigation bar, featuring back navigation and a More Options (more_vert) action button.
BlocProvider.value	Provides existing Cubit instances (PlaylistDetailsCubit, MyPlaylistsCubit) to the widget tree.
BlocBuilder	Manages UI rendering based on the PlaylistDetailsState (Loading, Failure, Loaded).
SingleChildScrollView	Allows the body content (Header and Song List) to scroll vertically.
Container	Used to display the playlist cover image using NetworkImage.
GestureDetector	Handles taps on the " Play All " button and individual songs to start playback.
ListView.separated	Special Widget: Renders the list of songs efficiently. It uses shrinkWrap: true and NeverScrollableScrollPhysics to allow it to function smoothly inside the SingleChildScrollView.
IconButton	Used for the More Options menu and for the song removal button (remove_circle_outline_rounded).
CircularProgressIndicator	Displays a loading spinner when the playlist details are being fetched.
showModalBottomSheet	Special Widget: Displays the "More Options" menu at the bottom of the screen.
AlertDialog	Special Widget: Displays a custom confirmation dialog before deleting a playlist.
EditPlaylistDialog	Custom Widget: Renders the content for editing the playlist details within a showDialog.

context.push / context.go	Special Technique: Uses the go_router package for navigation to the Song Player and My Playlists screen.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_bloc	The core State Management solution. PlaylistDetailsCubit handles data fetching and song removal, while MyPlaylistsCubit handles ownership checking, updating, and deleting the playlist.
go_router	Handles all navigation away from the page (e.g., to the Song Player, My Playlists page).
equatable	Used internally by the Cubit states (e.g., PlaylistDetailsState) to simplify value comparison and optimize rebuilds.
service_locator.dart	Used for Dependency Injection (sl<Cubit>) to retrieve pre-configured Cubit instances.
pocketbase (Implicit)	The underlying data access layer used within the UseCases for all remote operations (get details, delete, update, remove song).

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses a Hybrid BLoC/Cubit architecture, relying on two separate Cubit instances:

  - Dedicated State (PlaylistDetailsCubit): Manages the loading and content of the currently viewed playlist.

- Shared State (MyPlaylistsCubit): Manages the list of all user-owned playlists to handle ownership checking and trigger updates/deletes that affect the broader playlist list.

Workflow:

- **Initialization:** In initState, the PlaylistDetailsCubit is created and immediately calls loadPlaylist(playlistId) to fetch the data. The MyPlaylistsCubit is also retrieved and calls _checkOwnership() which runs loadPlaylists() to prime the user's ID and owned playlist list.
- **UI Rendering:** The UI uses BlocBuilder to show loading/error states or the loaded content.
- **Ownership Check:** The view dynamically determines if the current user is the owner using _myPlaylistsCubit.isMyPlaylist(playlist). This controls the visibility of the Edit, Delete, and Remove Song options.
- **Mutations (Owner Actions):**
    - Delete/Update: Calls the respective methods (_myPlaylistsCubit.deletePlaylist, _myPlaylistsCubit.updatePlaylist) which then trigger a reload of all user playlists via loadPlaylists().
    - Remove Song: Calls _detailsCubit.removeSongAt(index). The cubit first updates the local state to give the user immediate feedback, then calls the API (_removeSongFromPlaylistUseCase) to persist the change remotely.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

Remotely (Read/Write/Update/Delete): The page performs CRUD operations on the playlists collection and Read operations on the users and songs collections on the PocketBase backend.

Locally (Store/Read): No explicit local storage is used.

The application interacts with the backend via the PocketBase Dart SDK.

Fetching Playlist Details (Read)

- **SDK Call (Abstracted):** _getPlaylistByIdUseCase(params: playlistId).
- **Purpose:** Retrieves the specific playlist record, usually with the songs list included (e.g., via PocketBase's expand feature).
- **Input:** playlistId (the ID of the playlist to view).
- **Output:** PlaylistEntity (including: id, name, ownerId, coverUrl, and the nested list of SongEntity).

Removing Song from Playlist (Update)

- **SDK Call (Abstracted):** _removeSongFromPlaylistUseCase.call(params: params).
- **Purpose:** Sends an update request to remove the song ID from the playlist's song list on the server.
- **Input:**
    - RemoveSongFromPlaylistReq (containing the playlistId and the songId to remove).
- **Output:** Success/Failure.

Deleting Playlist (Delete)

- **SDK Call (Abstracted):** _deletePlaylistUseCase.call(params: playlistId).
- **Purpose:** Completely deletes the playlist record from the database.
- **Input:**
    - playlistId: The ID of the playlist to delete.

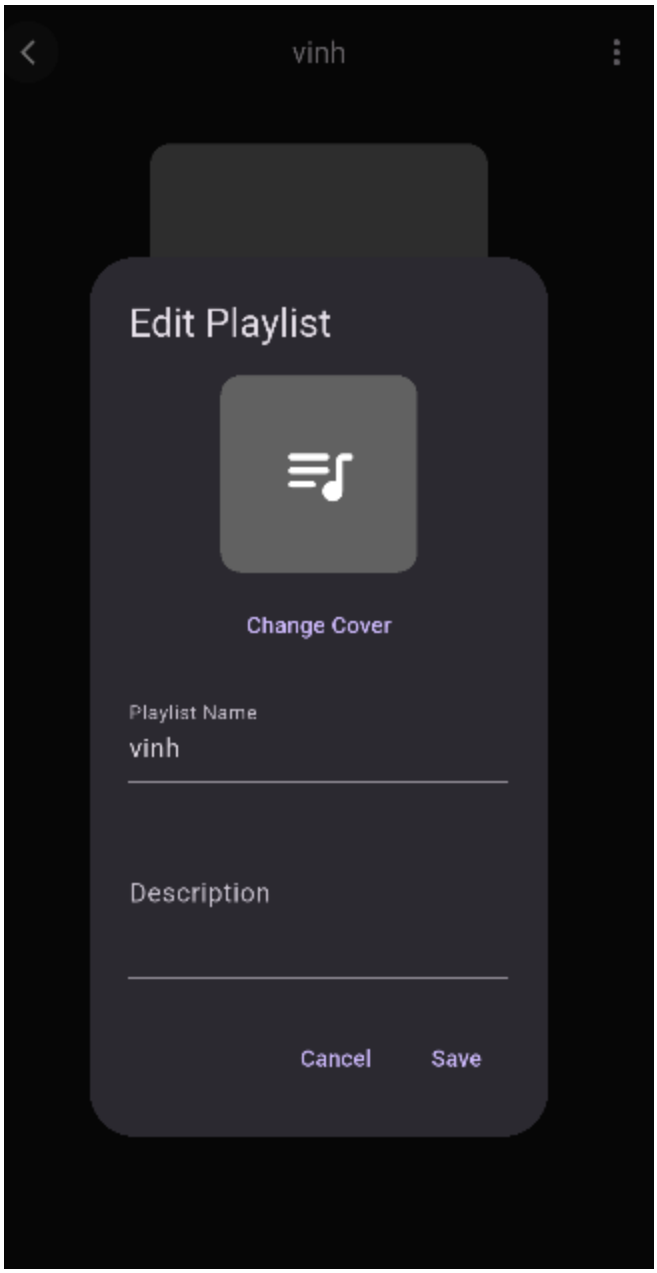
- **Output:** Success/Failure.

Data Table Structures (Implicit)

- **playlists table:** Contains playlist records (fields: id, name, ownerId (linked to users), coverUrl, songs (linked to songs)).
- **songs table:** Provides detailed data for the songs displayed in the list.
- **users table:** Implicitly queried to retrieve the _cachedUserId for ownership checking (ownerId == _cachedUserId).

7. **Feature / Application page: Edit Playlist**

- **Description:** The Edit Playlist Dialog is a modal window that allows a playlist owner to modify the name, description, and cover image of an existing playlist. It uses a StatefulWidget to manage user input and file selection locally, and utilizes flutter_bloc to dispatch the update request to the backend upon saving.
- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
AlertDialog	The main structure widget that displays the content as a modal popup over the current screen.
SingleChildScrollView	Allows the content within the dialog to scroll if it exceeds the screen height.

Column	Arranges the image selector, text fields, and spacing vertically.
GestureDetector	Handles taps on the cover image container to trigger the image picking function (_pickImage).
Container	Used to display the current or newly selected playlist cover image.
TextButton	Used for the dialog actions ("Cancel" and "Save") and for the "Change Cover Image" button.
TextField	Used for user input for the playlist Name and Description.
FileImage / NetworkImage	Special Widget: Used within the DecorationImage to display either a newly selected local file (_coverFile) or the existing remote cover URL.
ImagePicker	Special Technique: The image selection logic relies on the external image_picker package to access the device's gallery.
context.read<MyPlaylistsCubit>()	Special Technique: Accesses the MyPlaylistsCubit to dispatch the updatePlaylist event.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter/material.dart	Core Flutter UI library for standard widgets like AlertDialog, TextField, and TextButton.
image_picker	External plugin used to open the device's gallery and allow the user to select a new cover image file.
dart:io	Used to handle the selected image file as a File object before it's sent to the backend.
flutter_bloc	Used to access the shared state manager (MyPlaylistsCubit) to call the updatePlaylist function when the user saves changes.

pocketbase (Implicit)	The underlying data access layer used within the updatePlaylist UseCase to send the update request, potentially including the image file, to the backend.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This feature uses a combination of Local State and Shared State Management (Cubit):

  - **Local State (StatefulWidget)**: The dialog itself is a StatefulWidget (_EditPlaylistDialogState) which locally manages:
    - **Text Controllers**: _nameController and _descController hold the text input values.
    - **Image File**: _coverFile holds the temporarily selected image file (File?).
    - The dialog uses setState to update the cover image preview when a new file is picked.
  - **Shared State (MyPlaylistsCubit)**: The dialog does **not** provide the MyPlaylistsCubit; it expects it to be provided by its parent (the PlaylistDetailsPage).
    - **Workflow**: When the **Save** button is pressed, the dialog reads the updated values and the selected image file, packages them into a UpdatePlaylistReq object, and dispatches it by calling context.read<MyPlaylistsCubit>().updatePlaylist(req).
    - The MyPlaylistsCubit handles the remote API call and ensures the main list of playlists is reloaded upon success.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

Remotely (Read/Update): The dialog performs an initial Read of the playlist data (via widget.playlist) to pre-fill the fields, and a Write/Update operation upon saving.

Locally (Store/Read): It temporarily stores the selected cover image file (File?) in memory until the user presses save.

The application interacts with the PocketBase backend via the UpdatePlaylistUseCase (which is called by MyPlaylistsCubit).

Updating Playlist Details (Update/Write)

- **SDK Call (Abstracted):** _updatePlaylistUseCase.call(params: req).
- **Purpose:** Updates the metadata (name, description) and optionally uploads a new image file to replace the existing cover for the specified playlist ID.
- **Input:**
    - UpdatePlaylistReq: An object containing:
        - id: Playlist ID (String).
        - name: New playlist name (String).
        - description: New description (String).
        - cover: Optional image file (File?) to be uploaded as the new cover.
- **Output:** Success/Failure. A successful update triggers a reload of the main playlist list by the Cubit.

Data Table Structure (Implicit)

- **playlists table:** The record corresponding to the widget.playlist.id is targeted for the update operation. Fields being updated are name,

description, and coverUrl (the URL is updated after a successful file upload).

8. **Feature / Application page: Artist Detail Page**

- **Description:** The Artist Details Page provides a dedicated view for a specific music artist, identified by their ID. The page features a central header with the artist's name and avatar/cover image. Its primary functions include: Play All the artist's songs, Toggle Follow status for the artist, and displaying a list of all their available songs for individual playback. The data is managed using the Bloc/Cubit pattern to handle loading states and dynamic follow status updates.
- **Screenshots:**

- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure (AppBar + Body).
BasicAppbar	Custom widget serving as the top navigation bar, featuring back navigation and displaying the artist's name as the title.
BlocProvider	Provides the ArtistDetailsCubit instance and triggers the initial data load.
BlocBuilder	Manages UI rendering based on the ArtistDetailsState (Loading, Failure, Loaded).
SingleChildScrollView	Allows the body content (Header and Song List) to scroll vertically.
CircleAvatar	Displays the artist's cover image in a circular format, using NetworkImage if a URL is available.
GestureDetector	Handles taps on the "PLAY ALL" button and the "FOLLOW/FOLLOWING" button to dispatch corresponding actions.
ListView.separated	Special Widget: Renders the list of songs efficiently. It uses shrinkWrap: true and NeverScrollableScrollPhysics to allow it to function smoothly inside the SingleChildScrollView.
CircularProgressIndicator	Displays a loading spinner during the data fetching process.
context.push / context.go	Special Technique: Uses the go_router package for navigation (to the Song Player and the Home screen).

  - Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose

flutter_bloc	The core State Management solution. ArtistDetailsCubit handles data fetching, loading states, and managing the follow status.
go_router	Handles all navigation away from the page, routing to the Song Player (context.push) or back to the Home page (context.go).
equatable	Used internally by the Cubit states (e.g., ArtistDetailsState) to simplify value comparison and optimize rebuilds.
service_locator.dart	Used for Dependency Injection (sl<ArtistDetailsCubit>) to retrieve the Cubit instance.
pocketbase (Implicit)	The underlying data access layer used within the UseCases for all remote operations (get artist details, get songs, toggle follow status).

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen utilizes a Dedicated BLoC/Cubit pattern with the ArtistDetailsCubit.

  Architecture: The ArtistDetailsCubit combines multiple Use Cases (GetArtistUseCase, GetArtistSongsUseCase, IsArtistFollowedUseCase) into a single loadArtistDetails method, ensuring all necessary data is fetched concurrently or sequentially before emitting the final ArtistDetailsLoaded state.

  Workflow:

  - The BlocProvider creates the ArtistDetailsCubit and calls loadArtistDetails(artistId).

- The Cubit emits ArtistDetailsLoading, then sequentially fetches the ArtistEntity, checks the isFollowed status, and fetches the songsList.
- If successful, it emits ArtistDetailsLoaded containing the artist, songs, and isFollowed boolean.
- Follow Toggle: When the user taps the follow button, toggleFollowArtist is called. The Cubit immediately updates the local state (isFollowed: !currentIsFollowed) for instant UI feedback, and then calls the remote Use Case (followArtistUseCase or unfollowArtistUseCase). If the remote call fails, the state is reverted.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Read/Write): The page performs Read operations to fetch artist and song data, and a Write/Update operation to toggle the user's follow status for the artist.

  Locally (Store/Read): No explicit local storage is used.

  The application interacts with the backend via a suite of Use Cases.

Data Fetching and Mutation

1. **Fetching Artist Details & Songs (Read):**

- SDK Call (Abstracted): getArtistUseCase(artistId), getArtistSongsUseCase(artistId).
- Purpose: Retrieves the specific artist record and all associated song records.
- Input: artistId (String).
- Output: ArtistEntity and a list of SongEntity (including nested album details).

2. **Toggling Follow Status (Write/Update):**

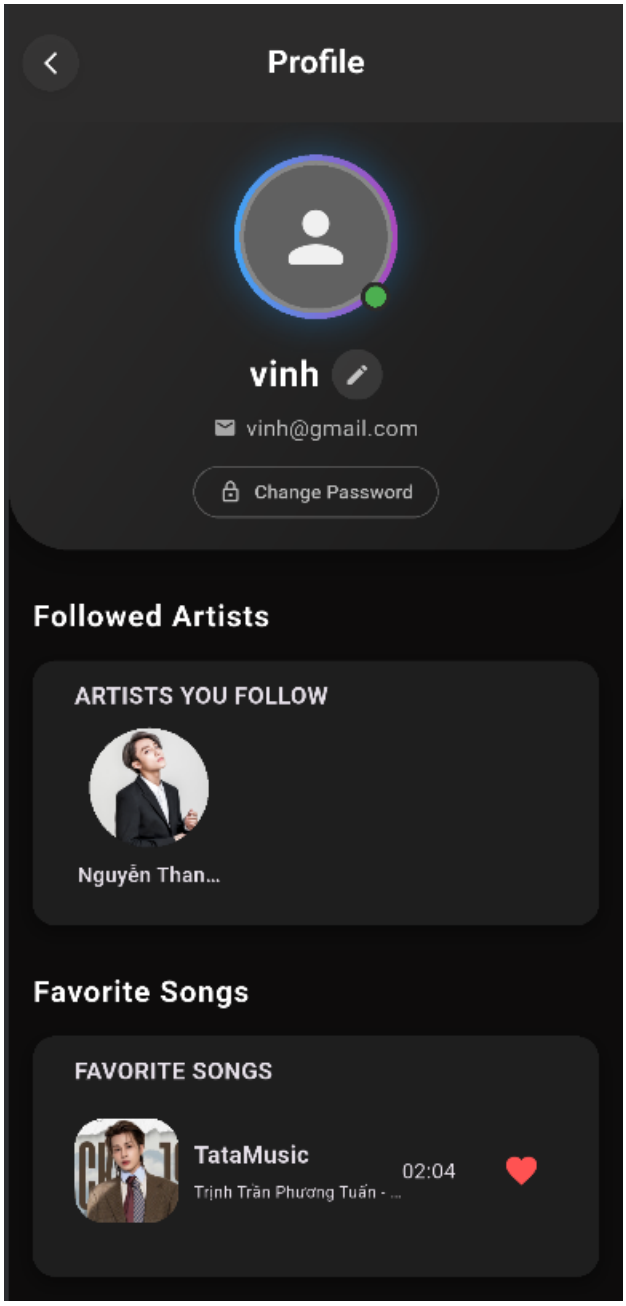
- SDK Call (Abstracted): Calls followArtistUseCase(artistId) or unfollowArtistUseCase(artistId).
- Purpose: Adds or removes the current user's ID from the artist's follower list on the server.
- Input: artistId (String).
- Output: Success/Failure.

Data Table Structures (Implicit)

- **playlists table:** Contains playlist records (fields: id, name, ownerId (linked to users), coverUrl, songs (linked to songs)).
- **songs table:** Provides detailed data for the songs displayed in the list.
- **users table:** Implicitly queried to retrieve the _cachedUserId for ownership checking (ownerId == _cachedUserId).

9. **Feature / Application page: Profile Page**

- **Description:** The Profile Page is the personalized hub for the authenticated user, designed to display key user information and provide access to saved content. The layout is vertically scrollable, adapts to the user's theme (Light/Dark Mode) by dynamically adjusting colors, and is composed of distinct sections for user details, followed artists, and favorite songs. The page also includes the crucial Log Out functionality.
- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure, adapting its background color based on the current theme.

BasicAppbar	Custom widget for the top navigation bar, displaying the "Profile" title.
BlocProvider	Provides the ProfileInfoCubit instance and triggers the initial user data fetch (getUser()).
SingleChildScrollView	Allows the entire body content to scroll vertically.
ElevatedButton	The distinct Log Out button, styled with a prominent red color.
ProfileHeader	Custom Widget: Displays the user's name and avatar (manages its own state via the Cubit).
FollowedArtistsSection	Custom Widget: Displays a list or preview of the artists the user is following.
FavoriteSongsSection	Custom Widget: Displays a list or preview of the user's favorite songs.
_buildSectionWrapper	Custom Builder Method: A private method used to create consistent, themed, card-like containers for the "Followed Artists" and "Favorite Songs" sections, including box shadows that adapt to Light/Dark Mode.
context.isDarkMode	Special Technique: An extension method (imported from common/helpers/is_dark_mode.dart) used to check the current theme mode for dynamic color assignments.
AudioPlayer	Special Library: The Log Out function uses sl<AudioPlayer>().stop() to ensure any currently playing audio stops before navigation.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_bloc	The core State Management solution. ProfileInfoCubit handles fetching and providing the user's details.

go_router	Handles navigation, specifically routing the user to the sign-in/sign-up page (context.go('/signup-or-signin')) after logging out.
just_audio	Used to stop audio playback (sl<AudioPlayer>().stop()) immediately before the user logs out.
pocketbase_client.dart	Provides the PocketBaseClient instance used to call the logout() method, clearing the authentication token.
service_locator.dart	Used for Dependency Injection (sl<Cubit>, sl<AudioPlayer>, sl<PocketBaseClient>) to retrieve necessary service instances.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses a Dedicated BLoC/Cubit pattern with ProfileInfoCubit.

  Architecture: The ProfileInfoCubit is a simple layer dedicated to retrieving the current user's entity.

  Workflow:

  - The ProfilePage is initialized with BlocProvider, calling cubit.getUser().
  - The Cubit emits ProfileInfoLoading, then calls sl<GetUserUseCase>().call().
  - If successful, the Cubit emits ProfileInfoLoaded containing the UserEntity.
  - The ProfileHeader (a consumer of this Cubit, though not shown) rebuilds to display the user's name and related info.

- The Log Out button handles actions that are outside of the Cubit's state management scope (stopping audio and clearing the remote session).

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Read/Write): The page performs Read operations to fetch the user's data and Write operations (implicitly) when logging out. The subsequent widgets (FollowedArtistsSection, FavoriteSongsSection) perform their own Read operations on the remote backend.
  Locally (Store/Read): No explicit local storage is directly managed by this page.

Data Fetching and Mutation

1. **Fetching User Data (Read):**
   - SDK Call (Abstracted): sl<GetUserUseCase>().call().
   - Purpose: Retrieves the user's entity based on the active authentication token stored in the client.
   - Input: None (relies on the active session/token).
   - Output: UserEntity (fields: id, name, email, etc.).
2. **Logging Out (Write/Delete Session):**
   - SDK Call (Abstracted): sl<PocketBaseClient>().logout().
   - Purpose: Clears the local authentication token/session data held by the PocketBase client, effectively logging the user out.
   - Input: None.
   - Output: Success (clears token, triggers navigation).

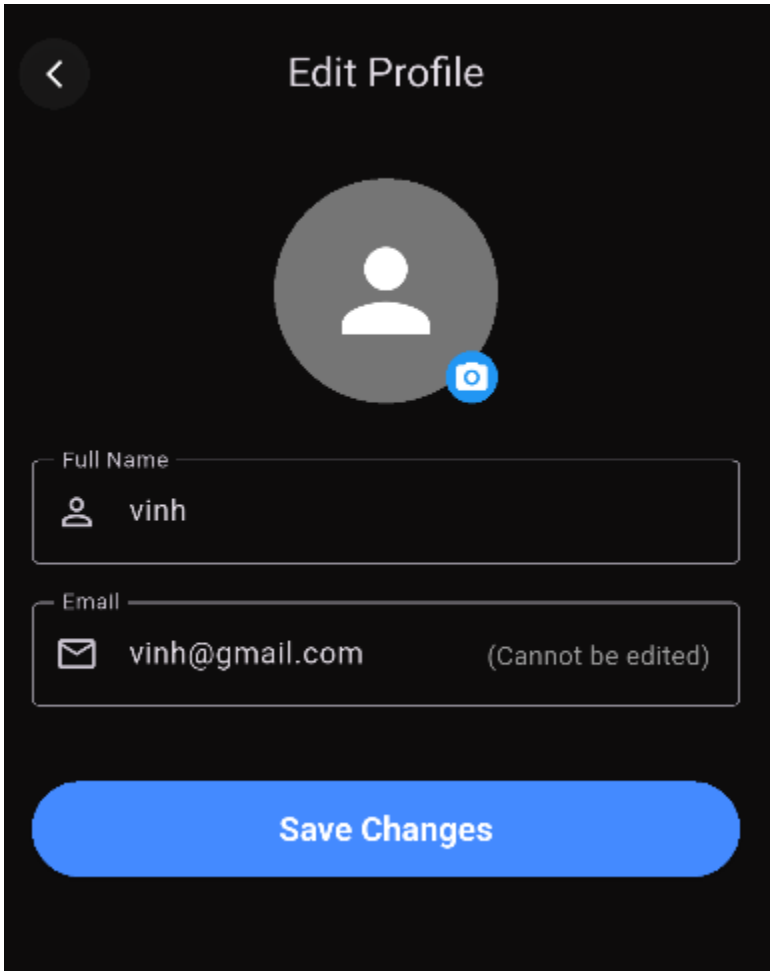
Data Table Structures (Implicit)

- **users table**: The primary table queried to retrieve the UserEntity data displayed in the ProfileHeader.

- **artists table:** Queried by the FollowedArtistsSection to fetch the artists the user is following.
- **songs table:** Queried by the FavoriteSongsSection to fetch the user's favorited songs.

10. **Feature / Application page: Edit Profile Page**

- **Description:** The Edit Profile Page is a dedicated screen for authenticated users to update their profile information. It allows the user to modify their Full Name and change their Avatar image. The user's email is displayed but is read-only. The page uses a StatefulWidget to manage local file selection and input text, and relies on the EditProfileCubit to handle the asynchronous upload and update process, providing feedback via SnackBar alerts.
- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure with a simple AppBar.
BasicAppbar	Custom widget used for the top navigation bar.
BlocProvider	Provides the EditProfileCubit instance to the page.
BlocListener	Special Widget: Listens for EditProfileSuccess or EditProfileFailure states to display a SnackBar and trigger navigation (context.pop()).
BlocBuilder	Listens for the EditProfileLoading state to dynamically show a CircularProgressIndicator instead of the "Save Changes" button.
Form	Wraps the input fields to enable validation via _formKey.
TextFormField	Input fields for the Full Name (editable) and Email (read-only).
CircleAvatar	Displays the user's current or newly selected avatar image.
Stack & Positioned	Used to layer the avatar image and position the camera icon button in the bottom right corner.
InkWell	Provides a tap target around the camera icon to trigger the avatar selection.
FileImage / NetworkImage	Special Widget: Used within the CircleAvatar to display either the newly selected local file (_newAvatarFile) or the existing remote cover URL.
ImagePicker	Special Plugin: The pickNewAvatar method within the Cubit relies on the external image_picker package to access the gallery.

  - Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_bloc	The core State Management solution. Used to dispatch the update event (updateProfile) and listen for success/failure.
go_router	Handles navigation, specifically routing the user back to the Profile Page (context.pop()) after a successful save.
image_picker	External plugin used to allow the user to select a new image file from the device's gallery.
dart:io	Used to handle the selected avatar file as a File object for display and upload.
service_locator.dart	Used for Dependency Injection (sl<EditProfileCubit>).

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses a Hybrid BLoC/Cubit pattern with two Cubit instances interacting:

  Dedicated State (EditProfileCubit): Manages the loading and result state (Loading, Success, Failure) specifically for the update operation.

  Shared State (ProfileInfoCubit): The EditProfileCubit holds a reference to the main ProfileInfoCubit (which manages the user's displayed profile data).

  - Crucial Workflow: Upon a successful update (EditProfileSuccess), the EditProfileCubit immediately calls _profileInfoCubit.getUser(). This forces the main Profile Page (which is listening to ProfileInfoCubit) to reload the updated user data, ensuring the avatar and name are refreshed globally without manual state passing.

    - Local State: The _EditProfilePageState manages the form input (_fullNameController) and the temporary _newAvatarFile using setState.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Read/Write): The page is initialized by Reading the user data passed from the parent page. It performs a Write/Update operation to persist the changes to the backend.

  Locally (Store/Read): It temporarily stores the selected avatar file (File?) in memory (_newAvatarFile) until the user presses save.

  Data Fetching and Mutation

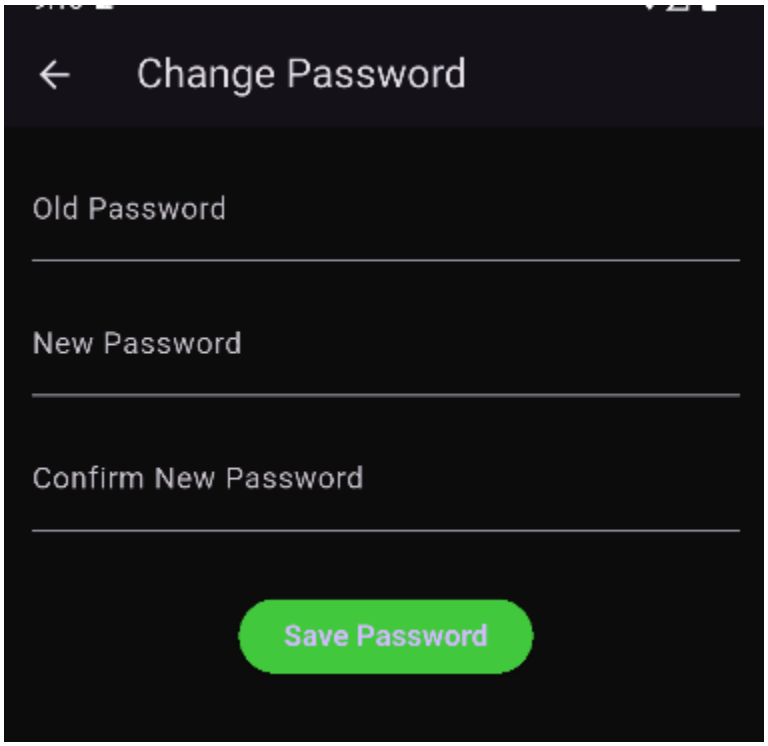
  1. Updating Profile (Update/Write):
     - SDK Call (Abstracted): _authRepository.updateProfile(req).
     - Purpose: Sends a request to update the user's full name and/or upload a new avatar file.
     - Input:
       - UpdateUserReq: An object containing newFullName (String) and newAvatarFile (File?).
     - Output: Success (returns updated UserEntity), or Failure (returns error message string).

  Data Table Structures (Implicit)

  - users table: The user's record is updated. The fields being modified are:
    - fullName: Updated with newFullName.
    - avatarUrl: Updated with the URL of the newly uploaded file (if newAvatarFile was provided).

11. **Feature / Application page: Change Password Page**

- **Description:** The Change Password Page provides a secure interface for an authenticated user to update their password. It requires the user to input their current password, a new password, and confirmation of the new password. The page utilizes form validation to enforce minimum password length and matching confirmation, and employs the Bloc/Cubit pattern to manage the asynchronous update process, providing feedback via a SnackBar.
- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure with a simple AppBar.
BlocProvider	Provides the ChangePasswordCubit instance to the widget tree.

BlocListener	Special Widget: Listens for ChangePasswordSuccess or ChangePasswordFailure states to display a SnackBar notification and trigger navigation (context.pop()).
BlocBuilder	Listens for the ChangePasswordLoading state to dynamically replace the "Save Password" button with a CircularProgressIndicator.
Form	Wraps the input fields and uses _formKey to enable batch validation before submitting the password change request.
TextFormField	Three separate input fields for Old Password, New Password, and Confirm New Password. Uses obscureText: true for security.
CircularProgressIndicator	Displays a loading spinner while the API call is being processed.
context.pop()	Special Technique: Uses the go_router package to navigate back to the previous screen (usually the Profile Page) upon success.
Validation Logic	Special Technique: Includes built-in form validation rules for minimum length (8 characters for New Password) and password matching (Confirm Password must equal New Password).

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_bloc	The core State Management solution. Used to dispatch the changePassword event and manage the UI based on ChangePasswordState.
go_router	Handles navigation, specifically routing the user back to the previous page (context.pop()) upon successful password change.

service_locator.dart	Used for Dependency Injection (sl<ChangePasswordUseCase>()) to retrieve the necessary Use Case for the backend operation.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses a Dedicated BLoC/Cubit approach with ChangePasswordCubit:

  Architecture: The ChangePasswordCubit is a standalone Cubit dedicated solely to handling the password change process. It takes the user's input and formats it into a request object for the backend.

  Workflow:

  - The user validates the form inputs and presses "Save Password".
  - context.read<ChangePasswordCubit>().changePassword(...) is called with the old and new passwords.
  - The Cubit immediately emits ChangePasswordLoading.
  - The Cubit calls the ChangePasswordUseCase with the ChangePasswordReq object.
  - The result is processed: If successful, ChangePasswordSuccess is emitted (triggering pop/snackbar). If an error occurs (e.g., incorrect old password), ChangePasswordFailure is emitted (triggering a red error snackbar).

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Write/Update): The page performs a Write/Update operation to modify the user's password hash stored on the backend.

  Locally (Store/Read): No explicit local storage is used.

The application interacts with the backend via the ChangePasswordUseCase.

Data Fetching and Mutation

1. **Updating Password Hash (Write/Update)**
    - SDK Call (Abstracted): sl<ChangePasswordUseCase>().call(params: ChangePasswordReq(...)).
    - Purpose: Authenticates the user using the oldPassword and updates the stored password hash to the newPassword.
    - Input:
        - ChangePasswordReq: An object containing oldPassword (String) and newPassword (String).
        - Output: Success (empty result on update) or Failure (error message if old password is wrong or update fails).

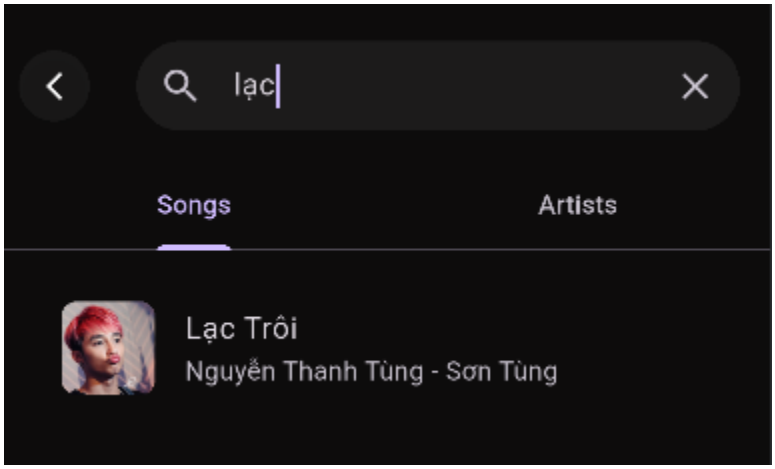
Data Table Structures (Implicit)

- **users table:** The record corresponding to the currently authenticated user is updated. The primary field modified is the stored passwordHash.

12. Feature / Application page: Search

- **Description:** The Search Page provides users with real-time, integrated search functionality across the music application's database. It is designed to immediately capture user input upon navigation, process the query using a debouncing mechanism for efficiency, and display results categorized into Songs and Artists using a tabbed interface. The page dynamically displays states such as Initial, Loading, Empty, and Loaded.

- **Screenshots:**



- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure.
BasicAppbar	Custom widget for the top bar, hosting the main TextField for search input.
TextField	The main search input field. It has autofocus: true and uses onChanged to trigger the search logic.
BlocProvider	Provides the SearchCubit instance to the widget tree.
BlocBuilder	Manages the main body content, switching between SearchInitial, SearchLoading, SearchFailure, SearchEmpty, and SearchLoaded states.
TabBar & TabBarView	Special Widget: Used to segment the results into "Songs" and "Artists" tabs. Requires the SingleTickerProviderStateMixin mixin.
ListView.builder	Renders the scrollable list of search results within each tab.
ListTile	Used to format individual search results for both songs and artists, including covers/avatars and titles.

Debouncer	Special Technique: A helper class (not a Flutter widget) used in the SearchCubit to delay the search execution until the user pauses typing for 500 milliseconds. This prevents excessive API calls.
context.push	Special Technique: Routes the user to '/song-player' or '/artist-detail' when an item is tapped.

- Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_bloc	The core State Management solution. SearchCubit manages the entire search lifecycle.
go_router	Handles navigation from the search results to the Song Player or Artist Details pages.
dart:async	Used by the Debouncer class to manage timers and delay execution.
service_locator.dart	Used for Dependency Injection (sl<SearchCubit>(), sl<SearchUseCase>()) to retrieve necessary service instances.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen uses a Dedicated BLoC/Cubit pattern with SearchCubit:

  Architecture: The SearchCubit is an independent component that encapsulates the search logic, including the debouncing mechanism, which is vital for performance.

  Workflow:

  - The user types in the TextField, triggering the onChanged callback.
  - context.read<SearchCubit>().search(query) is called.

- If the query is empty, the state resets to SearchInitial.
- If the query is valid, the Cubit delegates the action to the Debouncer.
- After 500ms of no further typing, the Debouncer runs the actual search logic:
    - It emits SearchLoading.
    - It calls sl<SearchUseCase>().call(params: query).
    - Based on the result, it emits SearchLoaded, SearchEmpty, or SearchFailure.
- The BlocBuilder in the UI reacts to these states to display the search results or status messages.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Read): The page performs Read operations to query the backend database for songs and artists matching the user's input.

  Locally (Store/Read): No explicit local storage is used.

  The application interacts with the backend via the SearchUseCase.

Data Fetching and Mutation

1. **Search API Interaction (Read)**
    - SDK Call (Abstracted): sl<SearchUseCase>().call(params: query).
    - Purpose: Performs a multi-collection search across the backend (likely using PocketBase's API's filtering capabilities across multiple tables or views).
    - Input:
        - query: The search term entered by the user (String).

- Output: SearchResultEntity (an object containing a list of SongEntity records and a list of ArtistEntity records).

Data Table Structures (Implicit)

- **songs table:** Queried to find songs where the title, artist name, or album name matches the query.
- **artists table:** Queried to find artists where the name matches the query.
- **albums table:** Implicitly joined or queried to retrieve song cover URLs for display in the search results.

13. Feature / Application page: Album

- **Description:** The Album Details Page displays comprehensive information about a specific music album, identified by its ID. The page features a prominent header displaying the album cover, title, and primary artist. Its core function is to list all the tracks belonging to the album, enabling playback for the entire album or individual songs via a "PLAY ALL" button. Data is managed by the AlbumDetailsCubit, which efficiently fetches album metadata and its tracklist concurrently.
- **Screenshots:**

Sơn Tùng MTP

Sơn Tùng MTP

Album by Nguyễn Thanh Tùng - Sơn Tùng

PLAY ALL

1	**SigmaMusicArt** Nguyễn Thanh Tùng - Sơn Tùng	01:26
2	**Lạc Trôi** Nguyễn Thanh Tùng - Sơn Tùng	03:53
3	**Buông Đôi Tay Nhau Ra** Nguyễn Thanh Tùng - Sơn Tùng	03:46
4	**Hãy Trao Cho Anh** Nguyễn Thanh Tùng - Sơn Tùng	04:05
5	**Âm Thầm Bên Em** Nguyễn Thanh Tùng - Sơn Tùng	04:51
6	**Nơi Này Có Anh**	

- **Implementation details:**
  - List the widgets used for this feature/page. Is there any special widget (not introduced in the lesson) used? If so, state them.

Widget	Purpose
Scaffold	Defines the page layout structure.
BasicAppbar	Custom widget serving as the top navigation bar, displaying the album title.
BlocProvider	Provides the AlbumDetailsCubit instance and triggers the initial data load (loadAlbumDetails).
BlocBuilder	Manages UI rendering, switching between AlbumDetailsLoading, AlbumDetailsFailure, and AlbumDetailsLoaded states.
SingleChildScrollView	Allows the body content (Header and Song List) to scroll vertically.
Container	Used to display the large album cover image using NetworkImage.
GestureDetector	Handles taps on the "PLAY ALL" button and individual songs to initiate playback.
ListView.separated	Special Widget: Renders the album tracklist efficiently. It uses shrinkWrap: true and NeverScrollableScrollPhysics to embed it within the SingleChildScrollView.
CircularProgressIndicator	Displays a loading spinner during the data fetching process.
context.push / context.go	Special Technique: Uses the go_router package for navigation, routing to the Song Player (context.push) or back to the Home page (context.go).

  - Does the feature use any libraries or plugins? If so, state them and state the role of those libraries/plugins

Library / Plugin	Purpose
flutter_bloc	The core State Management solution. AlbumDetailsCubit manages data fetching and loading states.

go_router	Handles all navigation, routing to the Song Player (/song-player) or back to the Home page.
dartz	Special Library: Used by the Use Cases to handle functional error results (returning Either<String, T>) for robust error handling.
dart:async	Used implicitly within the Cubit for concurrent operations (Future.wait).
service_locator.dart	Used for Dependency Injection (sl<AlbumDetailsCubit>) to retrieve the Cubit instance.

- Does this feature use a shared state management solution? If so, state briefly how your solution works and describe the code architecture.

  This screen utilizes a Dedicated BLoC/Cubit pattern with the AlbumDetailsCubitz:

  Architecture: The AlbumDetailsCubit encapsulates two distinct Use Cases: GetAlbumDetailsUseCase (for metadata) and GetAlbumSongsUseCase (for the tracklist).

  Workflow:

  - The BlocProvider creates the AlbumDetailsCubit and calls loadAlbumDetails(albumId).
  - The Cubit emits AlbumDetailsLoading.
  - It uses Future.wait to call both _getAlbumDetailsUseCase and _getAlbumSongsUseCase concurrently to improve load time.
  - It processes the results using nested fold methods (due to the Either return type from dartz).

- If both calls succeed, it emits AlbumDetailsLoaded with the AlbumEntity and the list of SongEntity.
- The BlocBuilder in the UI renders the album header and the song list based on the loaded data.

- Does this feature read or store any data? Locally or remotely? If so, state the data table structure. If you are using a REST API, briefly describe that API (how to call, input, output).

  Remotely (Read): The page performs Read operations to fetch the album metadata and all associated track data from the backend.

  Locally (Store/Read): No explicit local storage is used.

  The application interacts with the backend via the GetAlbumDetailsUseCase and GetAlbumSongsUseCase.

Data Fetching and Mutation

1. **Fetching Album Metadata (Read):**
   - SDK Call (Abstracted): _getAlbumDetailsUseCase(params: albumId).
   - Purpose: Retrieves the album's core information (title, cover, primary artist).
   - Input: albumId (String).
   - Output: AlbumEntity (fields: id, title, coverUrl, artists (list of entities)).
2. **Fetching Album Songs (Read):**
   - SDK Call (Abstracted): _getAlbumSongsUseCase(params: albumId).
   - Purpose: Retrieves all songs that link to this specific albumId.
   - Input: albumId (String).
   - Output: A list of SongEntity (fields: id, title, duration, artists).

Data Table Structures (Implicit)

- **albums table:** Stores the main album record.
- **songs table:** Stores track data, linked to the albums table via albumId.
- **artists table**: Implicitly joined or queried to retrieve the artist name for display in the header and the song list.