Noted Mobile Technical Documentation

1. Introduction

General Project Overview

The mobile application Noted is a companion app to the web application of the same name. Its purpose is to provide a mobile, portable, and intuitive experience for the Noted product. It includes the same core set of features:

- Note Viewing
- Group Management
- Quiz Generation (Coming Soon)
- Recommendation Generation (Coming Soon)
- Note Editing (Coming Soon)

Purpose of the Technical Documentation

The purpose of this technical documentation is to provide detailed information about the project. It is intended for developers and anyone involved in the design, maintenance, or technical understanding of the project. Since the project is open-source, this documentation can serve as a solid knowledge base for future contributors.

Technologies

The Noted mobile application is a cross-platform app for both Android and iOS, developed using Flutter.

- Flutter: Flutter is an open-source framework developed by Google that enables the creation of high-quality cross-platform applications for iOS, Android, web, and desktop using a single codebase. It's based on the Dart programming language, also developed by Google. One of Flutter's main advantages is its rapid development speed and its ability to deliver a responsive and performant user interface. It use a concept called "widgets" to create custom and interactive user interfaces. Widgets in Flutter are basic building blocks for creating interfaces and are highly customizable. By using Flutter, developers can write a single codebase for different platforms, reducing the time and effort required for development. The user interface is rendered using the built-in graphics engine in Flutter, ensuring a consistent and responsive look across different platforms.

> Official Flutter link: https://flutter.dev

- **DART**: Dart is an open-source programming language developed by Google. It is designed to be efficient for both client-side and server-side application development. Here are some key points for a quick understanding of Dart:
    1. **Versatile**: Dart can be used to develop web, mobile, desktop, and server applications. It is often associated with the Flutter framework for cross-platform application development.

2. **Modern Syntax**: Dart offers a modern and clean syntax, similar to other popular programming languages, making it relatively easy to learn for developers familiar with other languages.
3. **Strongly Typed**: Dart is a strongly typed language, which means that variable types are explicitly declared. This can contribute to early error detection and code quality improvement.
4. **JIT and AOT Compilation**: Dart can be compiled Just-In-Time (JIT) for rapid development and Ahead-Of-Time (AOT) for optimal runtime performance.
5. **Efficiency**: Dart emphasises performance, making it a good choice for developing responsive and high-performance applications.
6. **Growing Ecosystem**: While not as widespread as some other languages, Dart has gained popularity through Flutter. Its ecosystem, though smaller than that of some other languages, continues to grow with the increasing adoption of Flutter.
7. **Asynchronicity**: Dart incorporates asynchronicity features that facilitate the management of non-blocking operations, such as network calls and file interactions.

> Official Dart link: https://dart.dev

For certain features such as data tracking, Google integration, and bug tracking, distribution, the application incorporates various Firebase technologies.

- **Firebase**: Firebase is a mobile and web application development platform offered by Google. It provides a set of tools and services to facilitate the development, management, and deployment of high-quality applications. Here's a quick overview of what Firebase is:
    1. **Real-Time Database**: Firebase offers a real-time database (Firebase Realtime Database) that allows applications to synchronize and exchange data in real-time between clients and the server.
    2. **Authentication**: Firebase provides secure authentication features for users. Developers can set up login systems using Google, Facebook, Twitter accounts, etc., or use email and password authentication.
    3. **Hosting**: Firebase offers a web hosting service that enables the rapid deployment of static or dynamic web applications.
    4. **Cloud Storage**: Firebase Storage offers secure and scalable storage space for files such as images, videos, and documents, with easy integration into applications.
    5. **Cloud Notifications**: Firebase Cloud Messaging (FCM) allows sending push notifications to users to keep them informed and engaged.
    6. **Analytics**: Firebase Analytics provides insights into how users interact with your application, helping make informed decisions to improve user experience.
    7. **Crash Reporting**: Firebase Crashlytics monitors application errors and crashes, making it easier to detect and resolve issues.
    8. **Performance and Monitoring**: Firebase Performance Monitoring allows monitoring application performance and identifying bottlenecks to ensure a smooth user experience.
    9. **Machine Learning**: Firebase ML Kit offers ready-to-use machine learning features to add AI capabilities to your applications, such as facial recognition, text detection, etc.

10. **Authentication and Authorization**: Firebase Security Rules allow defining security and access rules for data stored in the database.

> Official Firebase link: https://firebase.google.com

In the Noted Mobile application, we use the following Firebase services:

- **Firebase App Distribution**: This service allows us to distribute the app during the Beta testing phase. It enables us to make the application available to users without the need to publish it on various app stores.

> Official link: https://firebase.google.com/docs/app-distribution

- **Firebase Analytics**: This service facilitates data collection about the app's usage, providing an overall view of how the app is utilized. It enables us to make informed development decisions that align with the needs of our users.

> Official link: https://firebase.google.com/docs/analytics

- **Firebase Crashlytics**: This service allows us to centralize the detection of bugs within the production application. It aids in identifying and subsequently resolving various issues.

> Official link: https://firebase.google.com/docs/crashlytics

- **Authentication**: This service enables us to seamlessly integrate third-party login support, such as Google, into the application. Users can create accounts on our web and mobile platforms using their Google accounts.

> Official link: https://firebase.google.com/docs/auth

2. Installation and Configuration

Flutter Installation Steps

To install Flutter, please follow the instructions available in the official Flutter documentation.

> Flutter Installation: https://docs.flutter.dev/get-started/install

Development Environment Setup

First, you need to clone the "mobile" repository from the Noted GitHub.

Repository Link: **https://github.com/noted-eip/mobile**

Run the following command to clone the repository from a terminal:

```
git clone git@github.com:noted-eip/mobile.git
```

Once the repository is cloned, run the following command at the root of the repository:

```
make update-submodules
```

The `update-submodules` command in the `Makefile` is used to update submodules in a Git repository. Submodules are separate Git repositories embedded within another Git repository, acting as separate directories within the main repository. This is the case with the submodule `protorepo` found within the `mobile` repository.

This specific command performs the following actions:

1. `git submodule update --init`: This part of the command initializes the submodules, meaning it fetches the submodules specified in the repository's configuration file and initializes them in their current state.
2. `remote`: This option tells Git to update the submodules from their remote reference (origin) rather than from the version recorded in the main repository. This ensures that submodules are synchronized with their remote repositories.

In summary, the `update-submodules` command ensures that submodules embedded in your main repository are up to date with the latest versions available in their remote repositories. This can be particularly useful when working with projects composed of multiple submodules and you want to ensure all parts of the project are synchronized and up to date.

This command should be executed to update the project when changes are made to the backend.

Then run the following command at the root of the repository:

```
make build-runner
```

This command is used to manage dependencies and run code generators using `build_runner`.

This command in the `Makefile` is used to manage dependencies and run code generators using `build_runner` in the project.

Here's a breakdown of each command:

1. `flutter pub get`: This command is used to fetch and install the dependencies specified in the `pubspec.yaml` file. The `pubspec.yaml` file describes the packages used in your project. By executing this command, you download the necessary packages from the pub.dev registry.
2. `flutter pub upgrade`: This command updates the dependencies specified in the `pubspec.yaml` file to the latest compatible versions. This can include minor or patch updates.
3. `flutter pub upgrade --major-versions`: This command updates dependencies to the latest major versions, which can result in more significant changes in features and compatibility.
4. `flutter pub run build_runner build --delete-conflicting-outputs`: This command runs the `build_runner` code generator. Code generators are used to automatically create code based on annotations and specific configurations. In the case of Flutter, this is often used for JSON serialization, generating

reflection code, etc. The `delete-conflicting-outputs` option is used to delete conflicting outputs if they exist.

This command ensures that dependencies are up to date and code generators are run to create necessary files before compiling and running the Flutter application. This can help avoid compilation errors related to missing or outdated code generators.

> 💡 This command should be executed to update the project when changes are made to the data part of the project.

3. Application Architecture

Overall Application Architecture

The architecture follows the typical structure of a Flutter application, adhering to the MVVM (Model-View-ViewModel) pattern by organizing the code into different parts:

- **Libraries (lib):** This section contains the core of the application, with folders for components, data, models, providers, services, and pages.
  - **components:** Reusable components, such as custom widgets.
  - **data:** Data management, including clients for interacting with services, data models, and notifiers for state management.
  - **pages:** Screens and pages of the application, grouped by features.
  - **utils:** Utilities and helper functions for various tasks.
  - **firebase_options.dart:** Firebase configuration for the application.
  - **generated_plugin_registrant.dart:** Generated file for plugins used in the application.
  - **main.dart:** Entry point of the application.
- **Resource Folders (images, web):** These folders contain static resources, such as images for different platforms.
- **Configuration Folders (docs, test):** These folders contain documentation-related documents and unit tests.
- **Configuration Files (LICENSE, Makefile, README.md, analysis_options.yaml, pubspec.yaml, pubspec.lock, mobile.iml):** These files contain licensing information, project configuration, and dependencies.

> Understanding MVVM Architecture: https://medium.com/flutterworld/flutter-mvvm-architecture-f8bed2521958

Interaction Between App Components

- **Components (lib/components):** Components are reusable widgets that encapsulate the logic and appearance of specific parts of the user interface. They are designed to be used in multiple places within the application.

    - Common components (**lib/components/common**), such as **base_container.dart**, **custom_alert.dart**, etc., provide reusable UI elements like custom containers, alerts, modals, etc.
    - Group components (**lib/components/groups**), such as **group_card.dart**, **group_member_card.dart**, encapsulate display and interaction logic for groups and their members.
    - Home page components (**lib/components/home**), like **home_info_widget.dart**, contain widgets specific to the home page.
- **Pages (lib/pages):** Pages represent different views and screens of the application. Each page is composed of components and uses data to display a specific user interface.
    - Account pages (**lib/pages/account**), such as **login_screen.dart**, **registration_screen.dart**, handle aspects related to user account management.
    - Group pages (**lib/pages/groups**), like **group_detail_page.dart**, **groups_list_screen.dart**, display information about groups and their members.
    - Home pages (**lib/pages/home**), like **home_screen.dart**, are responsible for the application's overview.
    - Note pages (**lib/pages/notes**), such as **note_detail_screen.dart**, **notes_list_screen.dart**, display notes and their details.
- **Data (lib/data):** Data is managed by clients, models, and providers.
    - Clients (**lib/data/clients**) interact with external services, such as APIs, to fetch and send data.
    - Models (**lib/data/models**) define the structure of data within the application. They often handle serialization/deserialization of data from APIs or databases.
    - Notifiers (**lib/data/notifiers**) manage the application's state. For example, **main_screen_notifier.dart** manages the global application state, and **user_notifier.dart** manages the user's state.
- **Services (lib/data/services):** Services provide application-specific functionalities.
    - **api_endpoints.dart**, **api_helper.dart**, **dio_singleton.dart:** These files handle API calls, endpoints, and Dio (HTTP library) configuration.
    - **failure.dart:** Manages structured handling of API call failures.

In general, pages use components to build the user interface, and components, in turn, can use notifiers to access global application data. Pages and components can also interact with clients to get or send data to/from external services.

The overall architecture of the application facilitates the separation of responsibilities and enables easier maintenance and evolution by isolating different parts of the logic.

4. Project Structure

Project Structure

Once the repository is cloned, you will have access to the project's file structure.

The first level of the structure is common to all Flutter projects:

At the root of the repository, you will find different folders:

- **/Android:** This folder will contain all the necessary files to build the application for Android.
- **/IOS:** This folder will contain all the necessary files to build the app for iOS.
- **/web:** This folder contains all the necessary files for building the web version of the app (consider removing it if not needed).
- **/build:** This folder will contain all the builds generated from the "flutter build" command.
- **/docs:** This folder contains documentation elements for the repository (e.g., a README with instructions for contributing to the project).
- **/images:** This folder contains all the images that will be used in the application.
- **/protorepo:** This folder is a submodule. It contains the content of another repository in the Noted organization (it's present in the app because it's where the HTTP client is generated from all the backend endpoints).
- **/test:** This folder contains all the test files for the mobile application.
- **/lib:** This folder is the main folder where you'll find all the source files of the application.

You will also find different files:

- **.gitignore:** This file lists the files that will be ignored during commits and pushes to the repository.
- **makefile:** This Makefile contains several commands for different actions on the repository.
- **pubspec.yaml:** This is the specific file where you can add specs to a Flutter project. It contains declarations for image paths, font paths, a list of dependencies, and environment specifications.

Let's focus on the /lib folder:

At the second level, you'll find different folders:

- **/components:** This is where you'll find all the components that will be used to fill the pages of the application.
- **/data:** This is where you'll find all the files related to data within the app (type declarations, providers, etc.).
- **/pages:** This is where you'll find all the files for the different pages of the application.
- **/utils:** This is where you'll find files containing functions, widgets, etc., that can be used in pages, components, and data.

And two files:

- **firebase_option.dart:** This file is auto-generated and contains all the information needed for integrating Firebase and its various services.
- **main.dart:** This is the entry file of the application. When the application is executed, this file is read first. It contains the declaration of the application, initialization of various services like providers, Firebase, or even screen orientation. It also includes the declaration of different routes for the application, which will be used by the Navigator (see Flutter navigation).

Let's go one level deeper to reach level 3. At this level, there will be a specific structure for the folders:

- **/component:** Components are organized by category/feature. You'll find components for:

    - **/group:** Display and management of groups.
    - **/invites:** Display and management of invitations.
    - **/notes:** Display and management of notes.
    - **/home:** Display and management of the app's home page.
    - **/common:** Generic components that can be used in multiple places in the app and are not specific to a feature.
- **/data:** In the data folder, you'll find several entities:
    - **/clients:** A client is a Dart class that encapsulates all methods related to an entity (entity list: account, group, invite, note). For example, in the Account client, you'll find all methods for data management related to accounts. Each method of the client makes API calls using the client auto-generated by the backend.
    - **/models:** A model is a declarative class for an object. In each model, you'll find the properties of an object and a method to create the object from the data model returned by the API.
    - **/services:** Services provide application-specific functionalities.
    - **/notifiers:** A notifier is a class that has properties and also has a changeNotifier. This has the effect of notifying every place in the application where the notifier is used whenever a change occurs. This automatically triggers a redraw of the screen.
- **/providers:** Provider is a library that simplifies state management in Flutter by allowing widgets to consume and provide data to other widgets. In our project, we're not using the default Flutter Provider, but we're using the Riverpod package instead.

↔ **The Relationship Between Riverpod and Provider**

Riverpod is designed to be the spiritual successor to Provider. Hence the name "Riverpod," which is an anagram of "Provider."
Riverpod was born out of the search for solutions to the various technical limitations that Provider faced. Originally, Riverpod was supposed to be a major version of Provider to address these issues. However, this was not done, as it would be a relatively major breaking change, and Provider is one of the most widely used Flutter packages.
Yet, conceptually, Riverpod and Provider are quite similar.
Both packages fulfill a similar role. They both try to:

- Cache and release certain stateful objects
- Offer a way to mock these objects during tests
- Offer a way for Widgets to listen to these objects in an easy way.

Over time, think of Riverpod as what Provider could have been if it continued to mature for a few years.

Riverpod addresses various fundamental issues with Provider, including, but not limited to:

- Greatly simplifying the combination of "providers." Instead of a tedious and error-prone `ProxyProvider`, Riverpod exposes simple yet powerful utilities such as **ref.watch** and **ref.listen**.
- Allowing multiple "providers" to expose a value of the same type. This removes the need to define custom classes when exposing a simple value like `int` or `String` would work just as well.
- Removing the need to redefine providers in tests. With Riverpod, providers are ready to be used in tests by default.
- Reducing excessive reliance on "scoping" to release objects by providing alternative ways to release objects (**autoDispose**). While powerful, scoping a provider is quite advanced and difficult to set up.

... And more.

> Riverpod Official Documentation: https://docs-v2.riverpod.dev/docs/why_riverpod

💡 I recommend thoroughly reading the documentation and fully understanding the concepts of Provider from the Riverpod package before using them.

5. Main Features

In this section, we will delve into the core functionalities of the application, explaining each one comprehensively. We'll also provide code examples to illustrate how these features are implemented.

Adding a New Page to the Application

In a Flutter application, a "page" or "screen" represents a distinct view that users interact with. Adding a new page involves creating a dedicated Dart file that defines the logic and UI for that particular view. After creating the page, it can be navigated to from other parts of the app.

For instance, let's create a new page named `NewPage`:

```
// lib/pages/new_page.dart

import 'package:flutter/material.dart';

class NewPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('New Page'),
      ),
      body: Center(
        child: Text('This is a new page!'),
      ),
    );
  }
}
```

Now, let's navigate to this new page from an existing page:

```
Navigator.push(context, MaterialPageRoute(builder: (context) => NewPage()));
```

By following this approach, new functionalities and content can be modularly added to the application while maintaining a clean and organized codebase.

Riverpod Provider / API Calls and Data Caching

Managing state in a Flutter application is crucial for maintaining consistent and dynamic user interfaces. Riverpod is a state management library that facilitates state management through the concept of "providers." These providers encapsulate data and expose it to widgets that need it.

To illustrate, let's consider fetching data from an API and caching it using Riverpod:

```
// lib/data/providers/account_provider.dart

import 'package:flutter_riverpod/flutter_riverpod.dart';
import 'package:your_app/data/models/account/account.dart'; // Import your model
import 'package:your_app/data/services/api_helper.dart'; // Import your API helper

final accountProvider = FutureProvider<Account>((ref) async {
  final apiResponse = await ApiHelper.fetchAccountData();
  return Account.fromJson(apiResponse.data);
});
```

To display the fetched account data in a widget:

```
// lib/pages/account/profile_screen.dart

import 'package:flutter/material.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';
import 'package:your_app/data/providers/account_provider.dart';

class ProfileScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Profile'),
      ),
      body: Consumer(
        builder: (context, watch, child) {
          final account = watch(accountProvider);
          return account.when(
            data: (account) {
              return Column(
                crossAxisAlignment: CrossAxisAlignment.center,
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  Text('Username: ${account.username}'),
                  Text('Email: ${account.email}'),
                ],
              );
            },
            loading: () => CircularProgressIndicator(),
            error: (error, stackTrace) => Text('Error loading account data'),
          );
        },
      ),
    );
  }
}
```

By adopting this approach, the application can manage complex states, including fetching data from APIs and efficiently updating the UI when the state changes.

Invalidating the Cache

Caching data is a strategy employed to enhance app performance by reducing unnecessary network requests. Yet, there are scenarios where cached data becomes outdated or requires an update. To tackle this, you can utilize Riverpod's capabilities to invalidate the cache.

For instance, by passing a unique cache key to the provider, you can ensure that new data is fetched when needed:

```
// lib/data/providers/account_provider.dart

final accountProvider = FutureProvider<Account>((ref) async {
  // Force refresh by passing a different cache key each time
  final apiResponse = await ApiHelper.fetchAccountData(cacheKey: DateTime.now().toString());
  return Account.fromJson(apiResponse.data);
});
```

With this mechanism, you're equipped to refresh the cache selectively, thus maintaining data accuracy without unnecessary overhead.

By understanding and employing these main features, newcomers to the project can gain insight into the app's architecture and its capabilities. This approach enhances collaboration and encourages the development of new functionalities while maintaining code quality.

6. Dependencies

Regarding the various dependencies of the project, which will be detailed later, the following are present:

- Repo "protorepo/openapi": HTTP client, allowing the setup of HTTP requests with auto-generated classes from objects in the backend.
- **otp_text_field**/**pin_code_fields**: Integration of an OTP code for account verification.
- **jwt_decode**: Allows decoding of the JWT token returned by the API, enabling request authentication.
- **shared_preferences**: Package enabling data storage on the user's device.
- **rounded_loading_button**: Package allowing the use of buttons with different states (loading, valid, not valid).
- **flutter_zoom_drawer**: Package enabling the import of a custom drawer.
- **flutter_slidable**: Package enabling the use of cards with slide gestures (invitations, etc.).
- **flutter_riverpod**: Package enabling the implementation of caching within the app.
- **shimmer**: Package enabling the use of skeletons (indicates to the user that content is loading by taking up the same space as the loading component).
- **tuple**: Package enabling the use of the tuple data type, which is not one of the basic types in Dart.
- **auto_size_text**: Package enabling text size adaptation based on screen space to avoid overflow.
- **fluttertoast**: Package allowing the implementation of toasts within the app to notify the user of critical actions or completed actions.
- **firebase_core**: Base package allowing the use of various Firebase services.
- **google_sign_in**: Package allowing simple and quick implementation of Google authentication.
- **firebase_analytics**: Package enabling the setup of tracking with Firebase Analytics.

7. Data Flow and State Management

In the realm of data management, I've opted for a powerful solution: the Riverpod package.

Riverpod, which intriguingly is an anagram of **Provider**, stands as a reactive caching framework tailored for Flutter/Dart applications.

By embracing declarative and reactive programming paradigms, Riverpod assumes the role of a dependable companion in handling a significant chunk of your application's logic. It boasts the capability to effortlessly manage network requests, complete with built-in error handling and caching. Moreover, it autonomously orchestrates the process of re-fetching data whenever the need arises.

Riverpod's prowess extends to facilitating caching mechanisms. This functionality serves the purpose of orchestrating API calls only when genuinely necessary, thus lending a higher degree of control to the management of data retrieval and transmission.

Imagine a scenario where certain data remains largely static throughout a user session. Repeatedly fetching such data upon navigating to various screens would not only burden the API but also impact the application's performance. This is where the beauty of caching shines. When data is initially retrieved, Riverpod stashes it away in a cache. Subsequent instances of data consumption within the application are fulfilled directly from this cache, bypassing the need to make repetitive API requests.

> Let's delve into illustrative code examples that showcase how to implement caching, create providers, and leverage their power.

Example Code - Cache Implementation and Provider Creation:

```
import 'package:flutter_riverpod/flutter_riverpod.dart';

final cachedDataProvider = FutureProvider<String>((ref) async {
  // Fetch data from API or another source
  final fetchedData = await fetchDataFromAPI();

  // Store the fetched data in the cache
  ref.maintainState = true;
  ref.onDispose(() {
    // Clear the cache when the provider is disposed
    clearCache();
  });

  return fetchedData;
});
```

Within this snippet, the `cachedDataProvider` is a Riverpod provider designed to fetch data from an API or another source. The fetched data is stored in a cache and retained as long as the provider remains relevant. Notably, as the provider's lifecycle ends, it clears the cache to ensure data integrity.

Managing State Changes:

Riverpod's forte lies in efficient data caching and seamless state management. When data within a provider changes, a ripple effect triggers an automatic rebuild of widgets dependent on that data. This orchestration is crucial for a dynamic and responsive user interface.

Example Code - Managing State Changes:

```
final countProvider = StateProvider<int>((ref) => 0);

Consumer(
  builder: (context, watch, child) {
    final count = watch(countProvider).state;
    return Text('Count: $count');
  },
)
```

In this snippet, the `countProvider` is a Riverpod state provider holding an integer. The `Consumer` widget responds to changes in the count state and automatically rebuilds the widget subtree with the updated value.

Example Code - Combining Providers:

```
final userDataProvider = FutureProvider<User>((ref) async {
  final userId = ref.read(userIdProvider).state;
  final user = await fetchUserFromAPI(userId);
```

```
  return user;
});
```

Here, `userDataProvider` fetches user data by combining two providers. The `userIdProvider` holds the current user's ID, and the data is fetched through `fetchUserFromAPI()`.

The marriage of Riverpod's intuitive caching capabilities with its seamless state management ensures data integrity, efficient API utilization, and an interface that evolves harmoniously with changes in underlying data.

In essence, Riverpod's integration exemplifies data mastery, efficient API consumption, and the automated propagation of state alterations. These attributes coalesce to elevate your application's performance while bestowing users with an experience marked by fluidity and responsiveness.

8. Best Practices

Flutter Development

- It is recommended to adopt a global approach whenever possible. Avoid creating overly specific or excessively complex components. Instead, prioritize the creation of generic components using arguments. For instance, instead of creating a specific button for each screen, consider creating a reusable button component with the ability to customize the text and associated action.
- Ensure a clear separation between display logic and data management logic. For example, for a list of articles, separate data retrieval from their display by using separate classes or files for each responsibility.
- Familiarize yourself with the project's architecture and add files to appropriate locations when necessary. For example, follow a folder structure such as "screens" for interface components, "models" for data objects, and "services" for API calls or global state management.
- Adhere to naming conventions: use UpperCamelCase for class names and follow the same convention when naming your files. For instance, name a data management class "DataManagement" and the corresponding file "data_management.dart".

State Management

- Choose a state management approach that suits the project, such as Provider, Riverpod, or BLoC, to maintain a consistent and scalable code structure. Avoid overloading local state with excessive information and prefer targeted updates. For example, for a task list application, use a state management solution to handle the global state of tasks rather than storing all information directly in widgets.

User Interface Design

- When you need a component with a behavior already present in the application, use existing components employed within it. For example, if your application already uses a customized button with a press animation, reuse this button to maintain a consistent user experience.
- Build your interfaces based on existing patterns to ensure visual consistency throughout the application. For example, if your application employs a specific visual style for section headers, apply this style across all sections of the application.

- Maximize the use of globally defined styles in the application's theme to ensure aesthetic consistency. For example, define colors, fonts, and margins consistently within the application's theme, so that all elements adhere to the same visual style.

9. Contribution

Dear prospective contributor,

We are delighted that you are considering contributing to our project. To ensure a smooth and productive contribution experience, please follow these guidelines:

1. **Comprehensive Documentation:** We encourage you to provide comprehensive and clear documentation about the project's overall architecture, key components, and workflows. This will help new contributors, like yourself, quickly grasp the project's big picture.
2. **Issue and Task Tracking:** Utilize our issue tracking system (such as GitHub Issues) to identify ongoing tasks and issues in the project. Please make sure to label these tasks with appropriate tags like "enhancement," "bug," "feature," etc. This will assist you in selecting tasks to work on.
3. **Use of Separate Branches:** We strongly encourage you to work on separate branches for each new feature, enhancement, or fix. This approach simplifies managing your contributions and allows other team members to review your changes before merging them into the main branch.
4. **Submitting Pull Requests (PRs):** When you are ready to submit your contributions, please do so through Pull Requests. This method provides an opportunity for other developers to discuss and review your changes before merging them. Ensure that you provide a clear description of the purpose of your PR, the changes made, and the tests you have conducted.
5. **Code Review Importance:** Your contribution will undergo a code review by other team members. This step is critical to maintaining high quality and consistency in our codebase.
6. **Comprehensive Testing:** We strongly encourage you to include unit tests and/or integration tests to validate your changes. This will ensure that your new features or fixes do not adversely affect other parts of the code.
7. **Adherence to Coding Conventions:** Please ensure that the code you submit adheres to our defined coding conventions. This includes indentation style, code formatting, variable and function naming conventions, and more.
8. **Explanatory Comments:** Feel free to add explanatory comments, especially for complex portions of the code you modify. This will aid other team members in understanding your contribution and in future maintenance.
9. **Respectful Communication:** Finally, we encourage respectful and constructive communication among contributors. Please focus your comments on ideas and solutions rather than personal criticisms.

We look forward to seeing your contributions and collaborating with you to enhance our project. Your commitment will contribute to advancing our work while maintaining the quality of our code and community.

Thank you for your interest and your future contribution!

Best regards,
The Development Team