VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT FINAL PROJECT

Supervisor: **MSc. MAI VĂN MẠNH**

Authors: **MAI GIA MINH – 521H0102**

BÙI ANH MINH – 521H0362

NGUYỄN PHƯƠNG TÀI – 521H0480

Class **: 21H50201 – 21H50202**

Class of **: 25**

HO CHI MINH CITY, 2024

VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT
FINAL PROJECT

Supervisor: **MSc. MAI VĂN MẠNH**

Authors: **MAI GIA MINH – 521H0102**

BÙI ANH MINH – 521H0362

NGUYỄN PHƯƠNG TÀI – 521H0480

Class **: 21H50201 – 21H50202**

Class of **: 25**

HO CHI MINH CITY, 2024

THANK YOU

First of all, my team would like to sincerely thank Ton Duc Thang University and the Faculty of Information Technology.

The group would like to sincerely thank the subject lecturer Mai Van Manh for enthusiastically conveying his treasure of knowledge to us. Thank you for your dedication to teaching so that we can apply it to this final project.

The last project below is the group's tireless efforts to repay the teachers' dedication. For the final project to be more complete, the group would like to receive comments, contributions, and criticisms from the teachers.

Finally, we would like to thank you for accompanying me throughout this topic and wish you good health and success on the path you are on.

THE PROJECT IS COMPLETED AT TON DUC THANG UNIVERSITY

I hereby declare that this is the product of our project and is guided by Mai Van Manh. The research content and results on this topic are honest and have not been published in any form before. The data in the tables for analysis, comments, and evaluation were collected by the author from different sources and clearly stated in the reference section. In addition, the project also uses several comments, assessments as well as data from other authors and other organizations, all with citations and source notes.
If any fraud is discovered, I will take full responsibility for the content of my project. Ton Duc Thang University is not involved in copyright violations caused by me during the implementation process (if any).

Ho Chi Minh City, date month year

Author

(signaturer and fullname)

Mai Gia Minh

Nguyen Phuong Tai

Bui Anh Minh

INSTRUCTOR VERIFICATION AND EVALUATION SECTION

Confirmation from the instructor

Ho Chi Minh City, date   month   year

Author

(signaturer and fullname)

The teacher's evaluation part marks the test

Ho Chi Minh City, date   month   year

Author

(signaturer and fullname)

SUMMARY

Creating an English vocabulary learning application akin to Quizlet involves designing a user-friendly interface where learners can curate personal flashcard sets. These sets could be organized by themes, such as 'Business English' or 'Everyday Phrases'. The app could offer different modes of study, like matching games, multiple-choice quizzes, and spelling tests, to reinforce learning. Additionally, incorporating spaced repetition algorithms can help users retain vocabulary more effectively over time. The goal is to create an engaging and interactive learning experience that motivates users to consistently expand their language skills.

TABLE OF CONTENT

CHAPTER 1 – INTRODUCTION

1.1 FLUTTER

1.1.1 Research and Comprehension

1.1.1.1 History

Origins and Early Development:

2014: Development of Flutter began internally at Google under the codename "Sky." The initial goal was to create a high-performance rendering engine capable of maintaining 120 frames per second on mobile devices.

Key Milestones:

May 2017: Google I/O Announcement

Flutter was officially announced at Google I/O 2017.

The first alpha version of Flutter was released, giving developers a preview of its capabilities.

September 2017: Flutter Release Preview 1

The first preview release included more features and performance improvements.

February 2018: Flutter Release Preview 2

Continued to refine the framework, adding more robust support for Material Design and enhancing developer tools.

December 2018: Flutter 1.0

Flutter 1.0 was released at the Flutter Live event in London.

This marked the framework’s first stable release, signifying readiness for production use.

Growth and Expansion:

May 2019: Flutter 1.5 at Google I/O 2019

Expanded support for the web.

Enhanced tools and improved overall framework performance.

December 2019: Flutter 1.12

Significant performance improvements.

Enhanced web and desktop platform support.

New features like Add-to-App, which allows integrating Flutter into existing native applications.

March 2021: Flutter 2.0

Introduced stable support for web applications.

Announced sound null safety, helping developers avoid null reference exceptions.

Expanded desktop support for Windows, macOS, and Linux, though still in an experimental stage for some platforms.

March 2022: Flutter 2.10

Stabilized Windows support, making it production-ready.

Further enhancements for macOS and Linux desktop applications.

May 2022: Flutter 3.0

Officially added stable support for macOS and Linux.

Continued to refine the web platform and introduced new performance enhancements.

Expanded support for Material You, Google’s new design language.

Features and Advantages:

Cross-Platform Development: Flutter allows developers to write a single codebase that can be deployed across iOS, Android, web, and desktop platforms, reducing development time and effort.

High Performance:

Uses the Skia graphics engine to provide high performance and smooth animations.

Compiles to native ARM code, ensuring fast execution on mobile devices.

Hot Reload:

Enables developers to see changes in their code instantly, greatly speeding up the development process and improving productivity.

Rich Set of Widgets:

Offers a comprehensive library of customizable widgets that adhere to both Material Design and Cupertino design guidelines, ensuring a native look and feel on both Android and iOS.

Declarative UI:

Flutter's UI is built using a modern, reactive framework inspired by React, allowing for easy and intuitive user interface development.

Community and Ecosystem:

Community Growth:

Since its launch, Flutter has attracted a large and active community of developers.

A vast number of third-party packages and plugins are available, which extend Flutter's functionality and help in faster development.

Adoption:

Used by major companies like Google, Alibaba, eBay Motors, and more for developing high-performance, cross-platform apps.

Google itself uses Flutter for several of its products, demonstrating its confidence in the framework.

Future Directions

Flutter continues to evolve with a focus on improving performance, expanding platform support, and adding new features. Google remains committed to supporting and growing the Flutter ecosystem, ensuring it stays relevant in the fast-changing landscape of software development.

1.1.1.2 Core Principles

Fast Development:

- Hot Reload

Definition: Hot Reload allows developers to see changes to their code in real-time without restarting the application.

Impact: This drastically reduces development time by providing immediate feedback, allowing for quicker iterations and more efficient debugging.

- Rich Set of Widgets

Definition: Flutter comes with a comprehensive library of pre-designed widgets that adhere to Material Design (Android) and Cupertino (iOS) standards.

Impact: These widgets make it easy to build complex, beautiful UIs quickly, ensuring consistency with platform-specific design guidelines.

Expressive and Flexible UI:

- Customizable Widgets

Definition: Every widget in Flutter is customizable and can be tailored to fit the specific needs of an application.

Impact: This flexibility allows developers to create unique, branded user experiences that stand out while maintaining performance and responsiveness.

- Declarative UI

Definition: Flutter uses a declarative programming model, where developers describe the UI’s state and layout, and Flutter handles the rendering.

Impact: This approach simplifies UI development, making it more intuitive to manage and update the UI state, leading to fewer bugs and more predictable outcomes.

Native Performance:

- Compiled to Native Code

Definition: Flutter applications are compiled directly to native ARM code for both iOS and Android.

Impact: This ensures that Flutter apps perform similarly to apps written in native languages, providing smooth animations and quick response times.

- High-Performance Rendering Engine

Definition: Flutter uses the Skia graphics engine to render the UI.

Impact: Skia is a high-performance 2D graphics library that ensures smooth and fast rendering, capable of handling complex graphics and animations efficiently.

Cross-Platform Development:

- Single Codebase

Definition: Developers write one codebase that runs on multiple platforms, including iOS, Android, web, and desktop (Windows, macOS, Linux).

Impact: This significantly reduces development and maintenance efforts, as the same code can be deployed across different platforms, ensuring feature parity and consistent behavior.

- Consistent UI and Business Logic

Definition: Flutter ensures that the same UI and business logic can be reused across different platforms.

Impact: This provides a seamless user experience and reduces the need for platform-specific code, simplifying the development process.

Open Source:

- Community Driven

Definition: Flutter is an open-source project with contributions from a global community of developers.

Impact: The community-driven nature leads to rapid development and a wealth of resources, plugins, and packages that extend Flutter’s functionality and provide solutions to common problems.

- Extensive Ecosystem

Definition: The open-source ecosystem includes numerous third-party packages and plugins.

Impact: This ecosystem enables developers to easily add functionality to their apps, such as integrating with various services and APIs, without having to build everything from scratch.

Productive Development Environment:

- Robust Tooling

Definition: Flutter offers a comprehensive set of tools, including the Flutter DevTools for performance profiling and debugging, and integrates with popular IDEs like Visual Studio Code and Android Studio.

Impact: These tools enhance developer productivity, making it easier to write, test, and debug Flutter applications.

- Well-Documented

Definition: Flutter’s documentation is thorough and includes clear guidance, examples, and best practices.

Impact: Good documentation helps developers get up to speed quickly and find solutions to their problems, making the development process smoother and more efficient.

Internationalization and Accessibility:

- Built-in Support

Definition: Flutter includes built-in support for internationalization (i18n), allowing apps to support multiple languages and locales.

Impact: This makes it easier to develop applications that cater to a global audience, increasing the potential user base.

- Accessibility Features

Definition: Flutter provides extensive support for accessibility (a11y) features, ensuring that apps are usable by people with disabilities.

Impact: Accessibility support ensures compliance with various accessibility standards and makes applications inclusive, reaching a broader audience.

Backward Compatibility:

- Stable APIs

Definition: Flutter aims to maintain backward compatibility with its APIs, ensuring that new updates do not break existing code.

Impact: This stability is crucial for long-term projects, as it minimizes the risk of having to make significant changes to the codebase with each new release.

Security:

- Secure Framework

Definition: Flutter emphasizes security by providing tools and best practices for secure coding, including secure data storage and encrypted communication.

Impact: Ensuring security is fundamental for developing trustworthy applications, protecting user data and preventing vulnerabil.

1.1.1.3 Specifications

- *Architecture and Core Components*

Dart Programming Language:

Language: Flutter applications are written in Dart, a modern, object-oriented language developed by Google.

Features: Dart offers features like strong typing, asynchronous programming with async/await, and a robust standard library.

Flutter Engine:

Core: The engine is primarily written in C++ and provides low-level rendering support using the Skia graphics library.

Components: It includes a Dart runtime, Skia for rendering, text layout, and other core libraries for handling graphics, file and network I/O, accessibility support, and plugin architecture.

Framework:

Layers: The framework is built in layers. The lower layers provide core services like animation, gestures, and painting, while the upper layers provide a rich set of widgets.

Widgets: Widgets are the building blocks of a Flutter app's UI. Everything in Flutter is a widget, from structural elements (buttons, text, images) to layout models and stylistic elements.

- *Rendering and Performance*

Skia Graphics Engine:

High Performance: Skia is a fast, open-source 2D graphics library used by Flutter to render UI components.

Capabilities: It supports advanced graphics features like bezier curves, gradients, anti-aliasing, and transformations.

Native Compilation:

ARM Code: Flutter compiles to native ARM code for both iOS and Android, enabling high performance close to native applications.

AOT and JIT Compilation: Flutter uses Ahead-of-Time (AOT) compilation for release builds and Just-in-Time (JIT) compilation during development for fast iterations.

- *Development Tools*

IDE Support:

Visual Studio Code: Offers robust Flutter support with extensions for Dart and Flutter, providing features like IntelliSense, code completion, and debugging.

Android Studio: Integrates Flutter and Dart plugins, supporting full Flutter development workflows, including project creation, running, and debugging.

Flutter DevTools:

Profiling: Tools for inspecting memory usage, performance, network activity, and debugging layout issues.

Widgets Inspector: Visual tool for inspecting the widget tree, understanding layout issues, and diagnosing problems.

- *UI Design and Widgets*

Material Design and Cupertino:

Material Design: Provides a comprehensive set of widgets that adhere to Google's Material Design guidelines, ensuring a consistent look and feel on Android devices.

Cupertino: Offers a set of widgets designed to mimic Apple’s iOS design language, ensuring a native look and feel on iOS devices.

Custom Widgets:

Composition: Widgets are composable and can be combined to create complex UIs.

Customization: Each widget can be customized and extended, allowing for highly flexible and expressive UI designs.

- *State Management*

Stateful and Stateless Widgets:

Stateless Widgets: Represent static parts of the UI that do not change over time.

Stateful Widgets: Represent parts of the UI that can change dynamically, holding mutable state.

State Management Solutions:

InheritedWidget: Basic method of propagating state down the widget tree.

Provider: A popular package for state management that offers a simple way to manage state and dependencies.

Riverpod, Bloc, Redux: Other state management solutions that provide different paradigms and patterns for managing app state.

- *Platform Integration*

Plugins:

Plugin System: Flutter has a rich plugin ecosystem that allows access to native device features like camera, geolocation, and sensors.

Platform Channels: Enables communication between Dart and native code (Java/Kotlin for Android, Swift/Objective-C for iOS).

Add-to-App:

Integration: Allows Flutter to be integrated into existing native applications, enabling gradual migration or the addition of Flutter UI components to existing apps.

- *Deployment and Distribution*

Compilation:

Release Builds: Flutter compiles to optimized, native code for deployment on iOS and Android, as well as web and desktop platforms.

Tooling: Flutter's CLI provides tools for building, testing, and deploying applications.

Distribution:

App Stores: Flutter apps can be published to the Apple App Store, Google Play Store, and other app distribution platforms.

Web Deployment: Flutter supports deploying web applications that can be hosted on any web server.

- *Internationalization and Localization*

Built-in Support:

i18n: Flutter supports internationalization and localization through the intl package.

Localizations: Allows developers to define and use localized messages, dates, numbers, and other locale-specific data.

- *Accessibility*

Accessibility Features:

Semantics: Flutter provides a semantics tree that helps screen readers interpret and describe the UI.

Focus Management: Supports keyboard navigation and focus management to ensure accessibility for all users.

- *Security*

Secure Data Handling:

Encryption: Provides tools and best practices for implementing secure data storage and transmission.

Authentication: Supports integration with secure authentication me

1.1.1.4 Analyzing the Advantages and Disadvantages of Flutter

Advantages:

- *Cross-Platform Development:*

Advantage: With a single codebase, developers can create apps for iOS, Android, web, and desktop platforms.

Impact: This reduces development time and costs, ensuring a consistent user experience across different platforms.

- *Fast Development:*

Advantage: Features like Hot Reload allow developers to see changes in real time without restarting the app.

Impact: This accelerates the development process, enhances productivity, and makes debugging easier.

- *Expressive and Flexible UI:*

Advantage: Flutter offers a rich set of customizable widgets that adhere to Material Design and Cupertino (iOS) standards.

Impact: Developers can build beautiful, natively compiled applications with smooth animations and a consistent look and feel.

- *High Performance:*

Advantage: Flutter apps are compiled to native ARM code, and the Skia graphics engine ensures high performance.

Impact: This results in fast, responsive apps with smooth graphics and animations, similar to native applications.

- *Open Source:*

Advantage: Flutter is open-source, with contributions from a global community.

Impact: The framework benefits from community-driven development, extensive documentation, and a wide range of plugins and packages.

- *Strong Community and Ecosystem:*

Advantage: A large, active community provides support, resources, and third-party plugins.

Impact: This facilitates problem-solving, learning, and extending the functionality of Flutter apps.

- *Rich Development Tools:*

Advantage: Flutter offers robust tooling, including integration with popular IDEs like Visual Studio Code and Android Studio, and DevTools for performance profiling and debugging.

Impact: Enhanced developer experience and productivity through powerful and user-friendly tools.

- *Internationalization and Accessibility:*

Advantage: Built-in support for internationalization and accessibility.

Impact: Developers can easily create apps that cater to a global audience and are accessible to users with disabilities.

Disadvantages:

- *Large App Size:*

Disadvantage: Flutter apps tend to have larger initial download sizes compared to native apps.

Impact: This can be a drawback for users with limited storage or slower internet connections.

- *Limited Third-Party Libraries:*

Disadvantage: While the ecosystem is growing, Flutter still has fewer third-party libraries compared to native development environments.

Impact: Developers may need to write custom code for specific functionalities not covered by existing libraries.

- *Platform-Specific Features:*

Disadvantage: Implementing platform-specific features can sometimes be challenging and may require writing platform-specific code.

Impact: This can complicate the development process and reduce the advantage of having a single codebase.

- *Limited Desktop and Web Support:*

Disadvantage: While Flutter supports web and desktop, these platforms are not as mature as mobile support.

Impact: Developers may encounter bugs and limitations, and the performance might not be as optimized as on mobile platforms.

- *Learning Curve:*

Disadvantage: Developers need to learn Dart, Flutter’s primary language, which might be unfamiliar to those used to languages like JavaScript or Swift.

Impact: This can slow down the initial adoption and development process as teams get up to speed with the new language and framework.

- *Immature Ecosystem for Some Use Cases:*

Disadvantage: Some specific use cases or advanced functionalities might not have mature solutions or libraries in Flutter's ecosystem.

Impact: Developers might need to invest extra effort to implement such functionalities, which can increase development time and complexity.

- *UI and UX Limitations*:

Disadvantage: While Flutter's widgets are highly customizable, there can be limitations in achieving the exact native look and feel for complex UIs.

Impact: This might affect the user experience and could require additional customizations to match the native appearance and behavior.

1.1.1.5 Challenges and Solutions in Flutter Development

- *Large App Size*

Challenge: Flutter apps tend to have larger initial download sizes compared to native apps.

Solution:

Optimize Assets: Use tools like flutter_image_compress to compress images and other assets.

Code Splitting: Split your code into smaller modules and load them dynamically to reduce the initial app size.

Remove Unused Resources: Regularly clean your project to remove unused resources and dependencies.

ProGuard: For Android, use ProGuard to shrink, obfuscate, and optimize your code.

- *Limited Third-Party Libraries*

Challenge: Flutter has fewer third-party libraries compared to native development environments.

Solution:

Custom Implementations: Write custom plugins or native code when necessary. Flutter provides platform channels to communicate with native code.

Contribute to Community: Contribute to the Flutter ecosystem by creating and sharing your own libraries.

Package Discovery: Use tools like pub.dev to discover existing packages that might fit your needs.

- *Platform-Specific Features*

Challenge: Implementing platform-specific features can be challenging and may require writing platform-specific code.

Solution:

Platform Channels: Use Flutter's platform channels to call native APIs and implement platform-specific features.

Plugin Development: Develop or use existing plugins that provide the necessary platform-specific functionality.

Federated Plugins: Consider using federated plugins, which allow separating platform-specific implementations into different packages.

- *Limited Desktop and Web Support*

Challenge: Flutter’s support for desktop and web platforms is not as mature as mobile support.

Solution:

Stay Updated: Keep track of Flutter’s updates and improvements for web and desktop support.

Early Adoption: Be an early adopter and contribute feedback to help improve the stability and features of Flutter for these platforms.

Fallback Solutions: For critical features that are not yet supported, consider using web-specific or desktop-specific solutions.

- *Learning Curve*

Challenge: Developers need to learn Dart, which might be unfamiliar to those used to other programming languages.

Solution:

Educational Resources: Utilize Flutter’s extensive documentation, tutorials, and community resources.

Practice Projects: Start with small projects to get hands-on experience with Dart and Flutter.

Online Courses: Enroll in online courses or bootcamps focused on Dart and Flutter development.

- *UI and UX Limitations*

Challenge: Achieving the exact native look and feel for complex UIs can be challenging.

Solution:

Custom Widgets: Create custom widgets to match the desired native appearance and behavior.

Community Packages: Use packages from the community that provide custom implementations or enhancements.

Design System: Implement a design system that balances the need for a native look and the benefits of Flutter’s cross-platform capabilities.

- *Immature Ecosystem for Some Use Cases*

Challenge: Some specific use cases or advanced functionalities might not have mature solutions or libraries in Flutter’s ecosystem.

Solution:

Community Engagement: Engage with the Flutter community to share your needs and possibly collaborate on solutions.

DIY Approach: Implement the required functionalities yourself and consider open-sourcing your solution to benefit others.

Hybrid Approach: Use a hybrid approach where you integrate Flutter with other mature frameworks or technologies for specific parts of your application.

- *Performance Optimization*

Challenge: Ensuring high performance, especially for complex animations and large data sets.

Solution:

Profiling Tools: Use Flutter’s DevTools to profile and identify performance bottlenecks.

Efficient Coding: Avoid unnecessary rebuilds, use keys appropriately, and manage state efficiently.

Asynchronous Operations: Perform heavy computations asynchronously to keep the UI responsive.

- *Debugging and Testing*

Challenge: Debugging and testing complex Flutter applications can be challenging.

Solution:

Automated Testing: Use Flutter’s testing framework to write unit, widget, and integration tests.

Debugging Tools: Utilize Flutter DevTools and IDE debugging features to step through code, set breakpoints, and inspect state.

Error Reporting: Implement error reporting tools like Sentry to capture and report runtime errors.

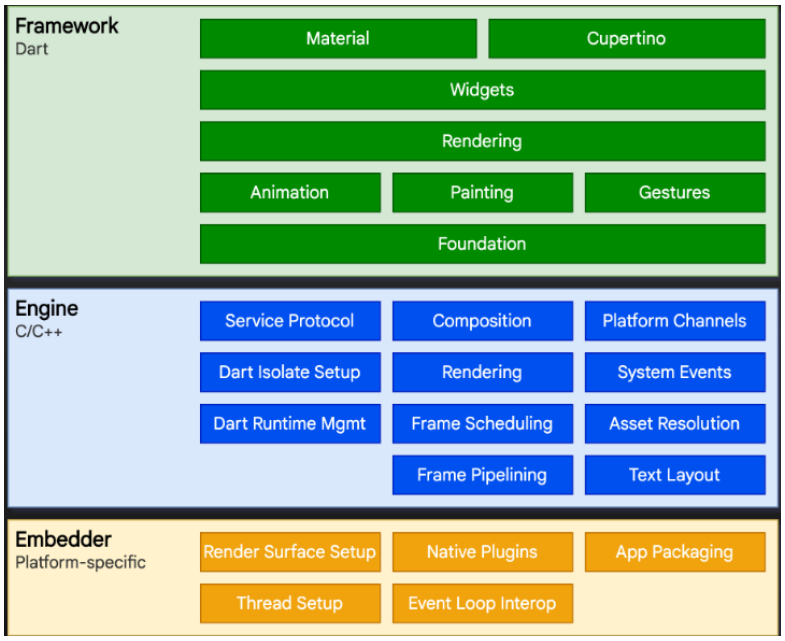


Figure 1.1 Architectural layers

1.2 Flashcard

Flashcard apps have truly revolutionized the way we learn and remember information. With high customization capabilities, users can create card sets that accurately reflect their needs and learning goals, from learning foreign languages to preparing for specialized exams. The spaced repetition system not only helps improve long-term memory but also helps users save time by focusing on information that needs to be reviewed more. This feature is based on scientific principles of how the brain stores and recalls information, optimizing the learning process by adjusting review schedules based on individual performance.

The portability of flashcard apps is an undeniable benefit, allowing users to take advantage of every free moment to study, whether it's while waiting, commuting, or even during their lunch break. This makes learning flexible and accessible to everyone, regardless of work schedules or personal lives. The user community also plays an important role, creating a collaborative learning environment where people can share their card sets and learn from each other. This not only helps expand knowledge but also creates opportunities to connect and build a learning support network.

Learning progress reports and statistics provide insight into each person's learning process, helping them recognize their learning patterns and adjust their learning methods accordingly. This not only helps users achieve better learning results, but also helps them feel more motivated and in control of their learning. With its increasing popularity, flashcard applications are not only a learning tool but also an indispensable companion in each person's knowledge journey. It is a testament to the advancement of educational technology, where convenience and efficiency go hand in hand, bringing unlimited learning opportunities to everyone, everywhere.

1.3 Function Flashcard

1.3.1 Account-related features

Register an account

User login

Change password

Reset password

1.3.2 Topic and folder management

View the list of topics

Create a new topic

Add/delete vocabulary to/from topics (both new and old topics)

Import/export vocabulary lists using csv

Provide a speaker icon to click to hear the pronunciation of each word in the list

Set privacy mode for topic (private/public)

Create a folder and add topics to the folder

Adjust folders (add topics or delete topics

View a list of folders

Delete a topic or folder

Specific statistical function for each vocabulary in a topic (not yet learned, learned, memorized)

Star each vocabulary word to put it in a separate list

1.3.3 Vocabulary learning features

Learn vocabulary with flashcards

Customize settings when studying flashcards

Learn vocabulary with multiple choice (quiz)

Customize settings when studying for multiple choice tests (quiz)

Learn vocabulary by typing words

Customize settings when learning by typing words

Apply text to speech to pronounce English words automatically for each word in learning modes

Save and display learning progress statistics for each specific vocabulary in a topic

1.3.4 Community features

Store vocabulary data, topics, folders online

Store user accounts online

View a list of public topics across the system in a completely separate interface

View information related to the rankings of a public topic

Participate in learning on a public topic created by others

Setting screen and changing settings

CHAPTER 2 APPLICATION ARCHITECTURE

2.1 Clean Architecture

In software development, to create an efficient, scalable and easy-to-read codebase, developers have come up with many solutions such as the 3-tier architecture, in which we split the codebase into 3 separate parts including the data layer, the domain layer and the presentation layer. In details, these layers will have the following functionalities:

Data: Responsible for communicating with the backend, sending requests and receiving responses, extracting local or external data, this layer is solely concentrated on data-related tasks. It does not interfere with the business side of our application.

Domain: Responsible for anything related to business logic of our application. This layer defines all of the processes or use cases of a particular screen, such as Login or Registration for examples. This acts as an intermediate between the presentation layer and the data layer.

Presentation: Responsible for what the users see on screen, this layer holds the codebase of the user interface and our application chosen state management.

In clean architecture, each layer is divided further into 3 subdivisions.

2.1.1 Data Layer

In the data layer, there will be the following three layers:

Data Source: This defines where our data comes from and how we can interact with it. Typically, there will be a local data source such as in app temporary memory and a remote data source such as API.

Model: This subdivision comprises the code of our models class, which will define how we handle data.

Repository: The implementation of the repository interfaces of the Domain layer, which we will get into further in the later section.

2.1.2 Domain Layer

In the domain layer, there will be the following three layers:

Entities: Defines the model class that is exclusively used for business logic. It is different from the data models in the data class. When we are making calculations or extracting information of an entity, we are going to utilise this object.

Repository: Defines an interface which lists all of the data related functions for one particular screen, this interface will then be implemented by the repository of the data layer.

Use Cases: Defines all of the use cases in the system design, such as changing the dark mode of a screen.

2.1.3 Presentation Layer

This will be the final layer and it contains the codebase of the application user interfaces and the state management tools.

State Management: Defines the state management tools used in our application, this can be Bloc, GetX or Riverpod for some examples.

Pages: Defines code of the user interface.

Widgets: Define the components of the user interface.

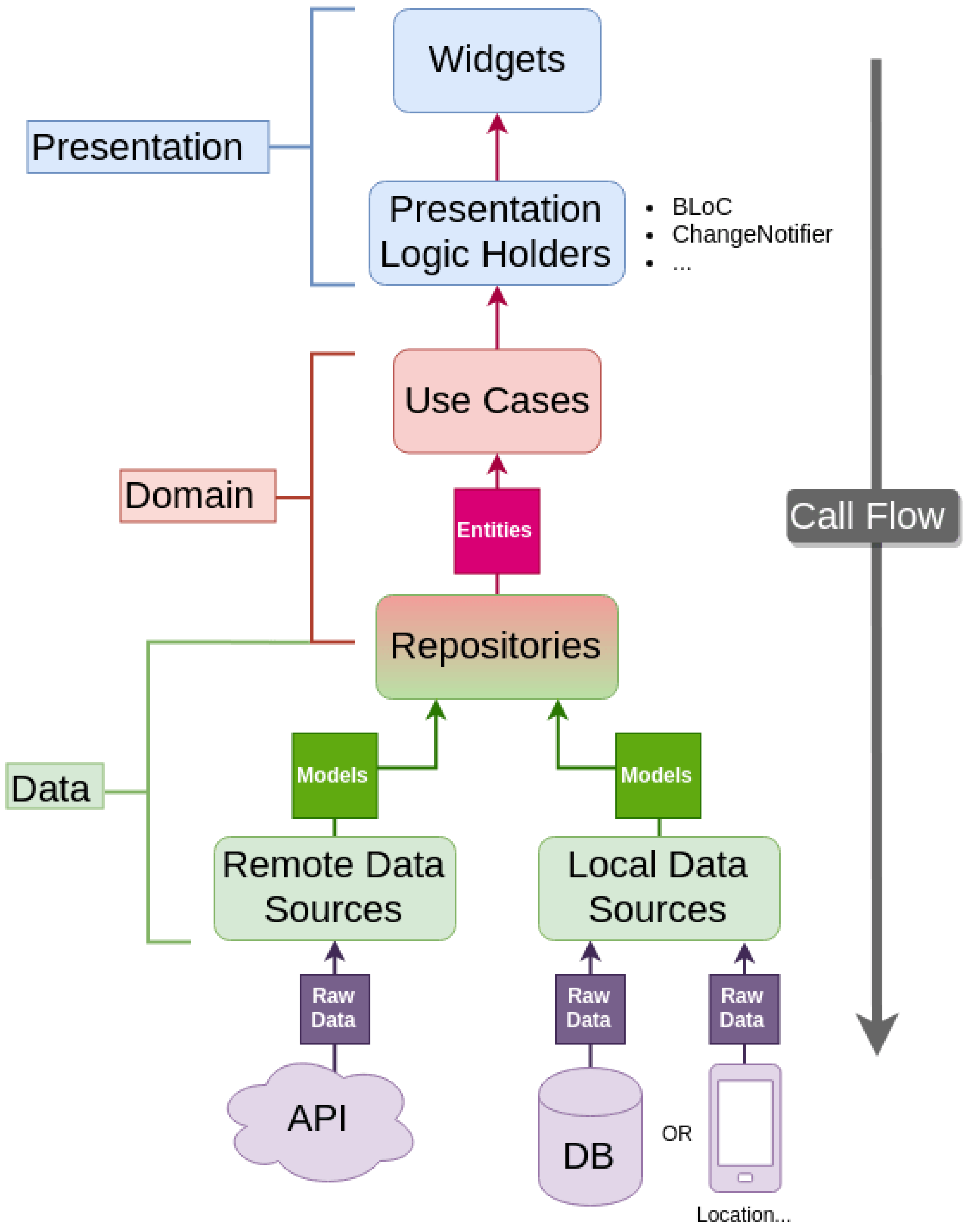


Figure 2.1: Clean architecture diagram

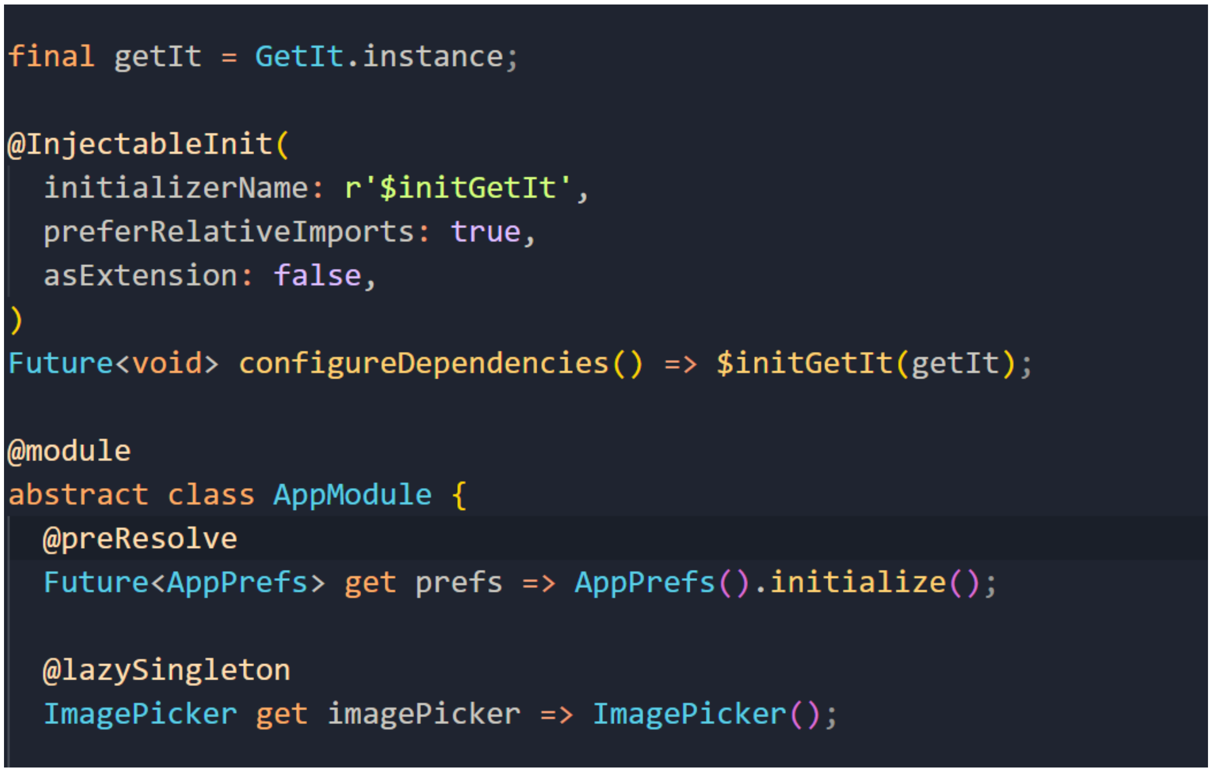
2.2 Network Libraries

2.2.1 get_it

Get_it is one of the most popular packages that provides a direct and simple-to-use implementation of service locator design pattern. This means that the package will get the corresponding services and inject them into a new object, it is very similar to the dependency injection pattern that we are familiarised when coding with Java or C#.

To implement get_it inside of Flutter, we will need to add some libraries such as get_it, injectable and injectable_generator. In the later course of this essay, we will implement a lot of code generation, so make sure we also add the build_runner package as well.

After we have done the initial process, we first need to create a file which will hold the code of our service locator.



```
final getIt = GetIt.instance;

@InjectableInit(
  initializerName: r'$initGetIt',
  preferRelativeImports: true,
  asExtension: false,
)
Future<void> configureDependencies() => $initGetIt(getIt);

@module
abstract class AppModule {
  @preResolve
  Future<AppPrefs> get prefs => AppPrefs().initialize();

  @lazySingleton
  ImagePicker get imagePicker => ImagePicker();
```

Figure 2.2: Service locator

In this code, we are taking an instance of GetIt and later on, we will use this instance to access the service that we define. @InjectableInit is an annotation that signifies we are going to generate and configure the services (or dependencies), the $initGetIt(getIt) method will not be generated if we have not run the following command line.

flutter pub run build_runner build --delete-conflicting-outputs

This command line is used to trigger the build_runner package's code to start building the code for the initGetIt method.

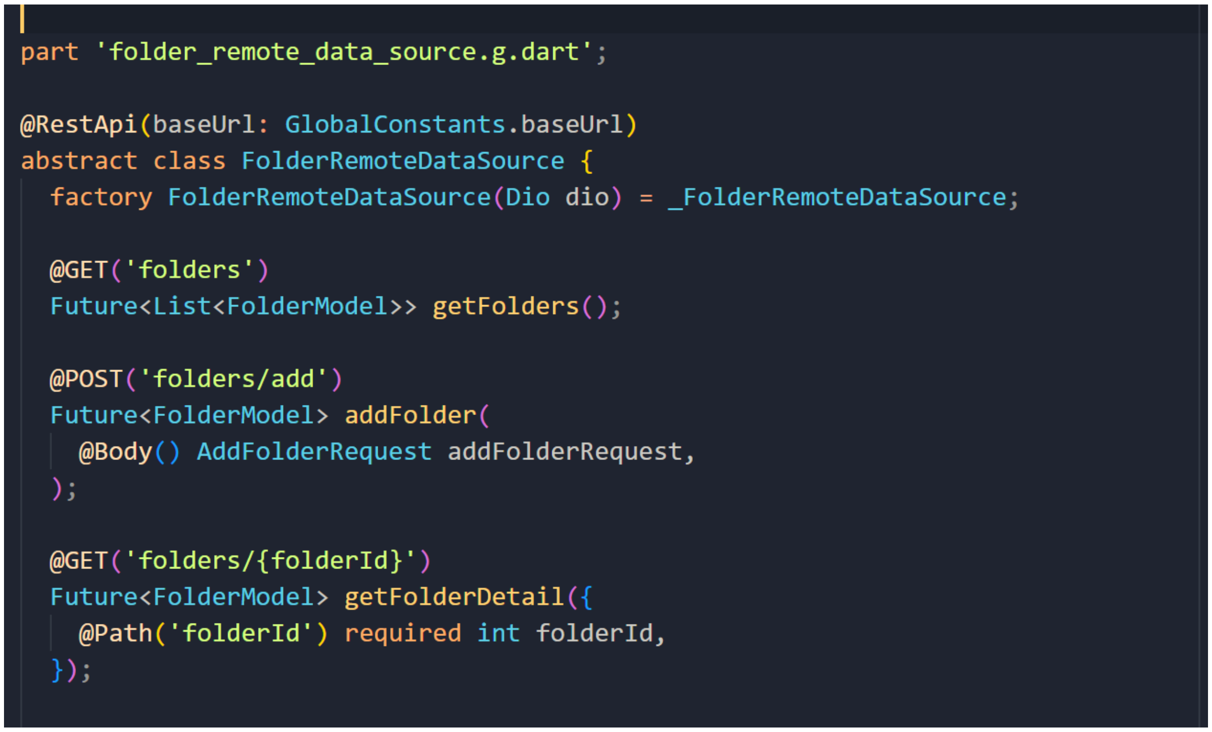
We can also define some third party dependencies such as AppPrefs, which is our implementation and extension from SharedPreferences in Flutter, or the ImagePicker package. The @module annotation defines the class as holding all of the third party service, the @preResolve annotation is to make sure that service got to be initialised before starting the app.

There will be three most used annotations in Flutter to define which design patterns an object of that class will be implementing. First, it is @factory, @singleton and @lazySingleton, as the name suggests, each of the classes will be noted with their desired design pattern. The main difference between singleton and lazy singleton is that a lazy singleton object will only be initialised when it is first used, and the singleton will already do so before the application starts.

2.2.2 retrofit, dio

In Flutter, dio is a package that provides a light-weight, customizable and powerful HTTP client for making requests and handling responses. While retrofit is an interface built on top of Dio, it will provide us with annotations to define the type of requests, the endpoints location and its headers as well.

To set up, we need to add the primary three packages including retrofit, dio and retrofit_generator. We will use build_runner to generate the code as well.

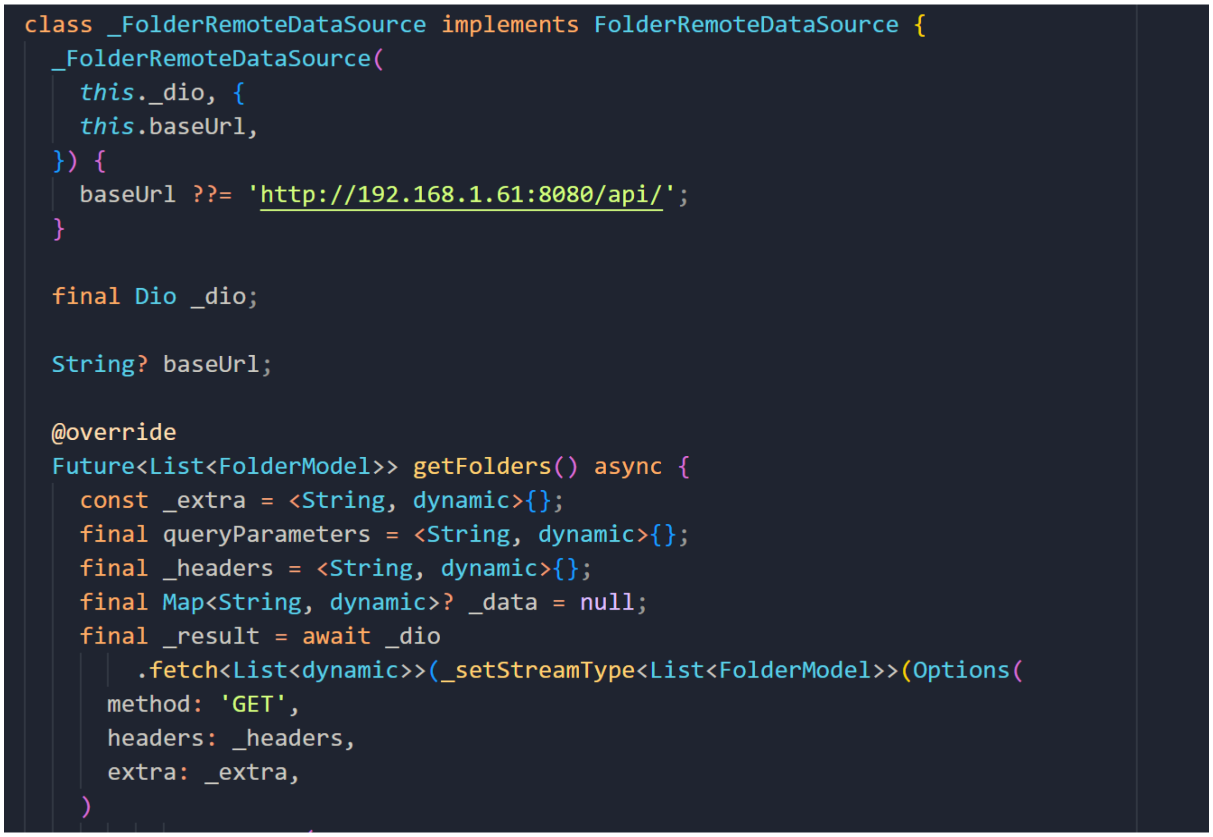


```dart
part 'folder_remote_data_source.g.dart';

@RestApi(baseUrl: GlobalConstants.baseUrl)
abstract class FolderRemoteDataSource {
  factory FolderRemoteDataSource(Dio dio) = _FolderRemoteDataSource;

  @GET('folders')
  Future<List<FolderModel>> getFolders();

  @POST('folders/add')
  Future<FolderModel> addFolder(
    @Body() AddFolderRequest addFolderRequest,
  );

  @GET('folders/{folderId}')
  Future<FolderModel> getFolderDetail({
    @Path('folderId') required int folderId,
  });
```

Figure 2.3: Retrofit interface

Here, we have an abstract class that is taking in a Dio object through service locator and initialising the object by the factory keyword. After that, we define the base endpoint of the interface by using the baseUrl attribute of the @RestApi annotation, and then we can define the methods, what are the HTTP request methods, its return type and what is their request body.

Make sure to add the part “your_file_name.g.dart” for the build runner package to generate new code, and copy them into our .g.dart file. After that, the code inside the generated file must not be modified, it will auto regenerate after every time you use the build runner command line.



```
class _FolderRemoteDataSource implements FolderRemoteDataSource {
  _FolderRemoteDataSource(
    this._dio, {
    this.baseUrl,
  }) {
    baseUrl ??= 'http://192.168.1.61:8080/api/';
  }

  final Dio _dio;

  String? baseUrl;

  @override
  Future<List<FolderModel>> getFolders() async {
    const _extra = <String, dynamic>{};
    final queryParameters = <String, dynamic>{};
    final _headers = <String, dynamic>{};
    final Map<String, dynamic>? _data = null;
    final _result = await _dio
        .fetch<List<dynamic>>(_setStreamType<List<FolderModel>>(Options(
      method: 'GET',
      headers: _headers,
      extra: _extra,
    )
```

Figure 2.4: Generated Retrofit code by Build Runner

This is an example of the code, everything is written out of the box, we do not need to write every single dio request manually.

2.2.3 flutter_bloc

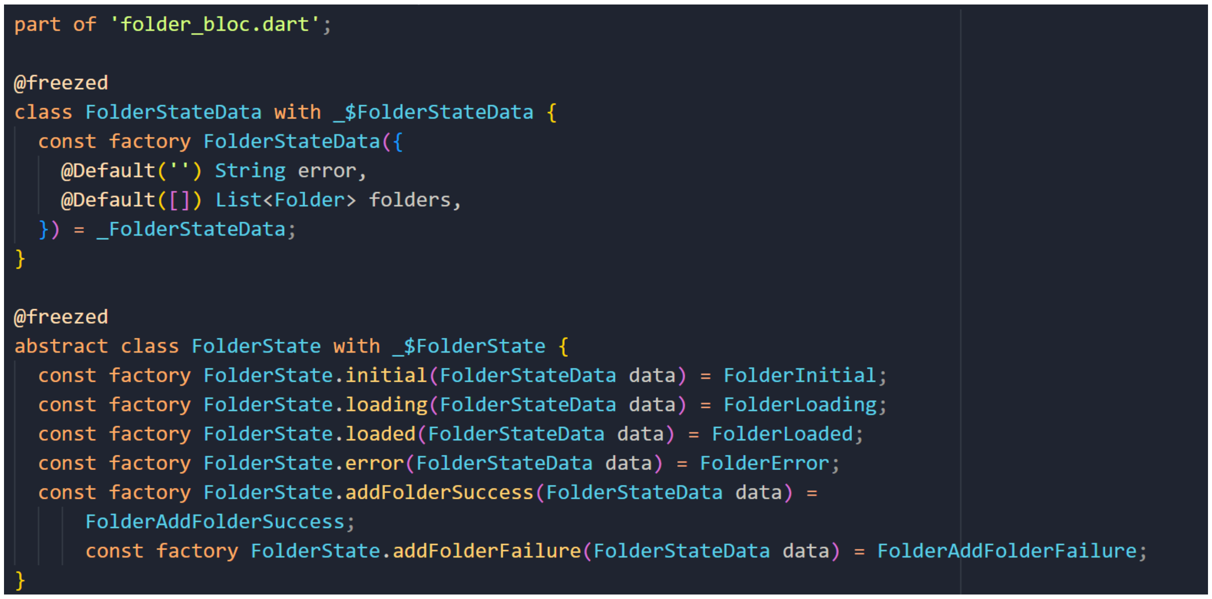
For state management tools in Flutter, we have many options such as Provider, Riverpod, getX and Bloc. Based on simplicity, efficiency and to help all members of the team to be on the same page when it comes to reading the source code, Bloc is the top choice for us, there are many reasons to choose Bloc over other state management tools.

Bloc utilises the business logic component pattern, which means we will need to separate the presentation or views from the business logic side and from the data side as well. It is an ideal clean architecture design.

Bloc is event-based and the corresponding views will be changed based on the stream of states.

Bloc is easy to implement even if you are not familiar with Flutter. We need a Bloc class, a Bloc event class and a Bloc state class. The Bloc class will define what will happen after receiving an event, the view on the front-end side can listen to the changing of state to update its corresponding view.

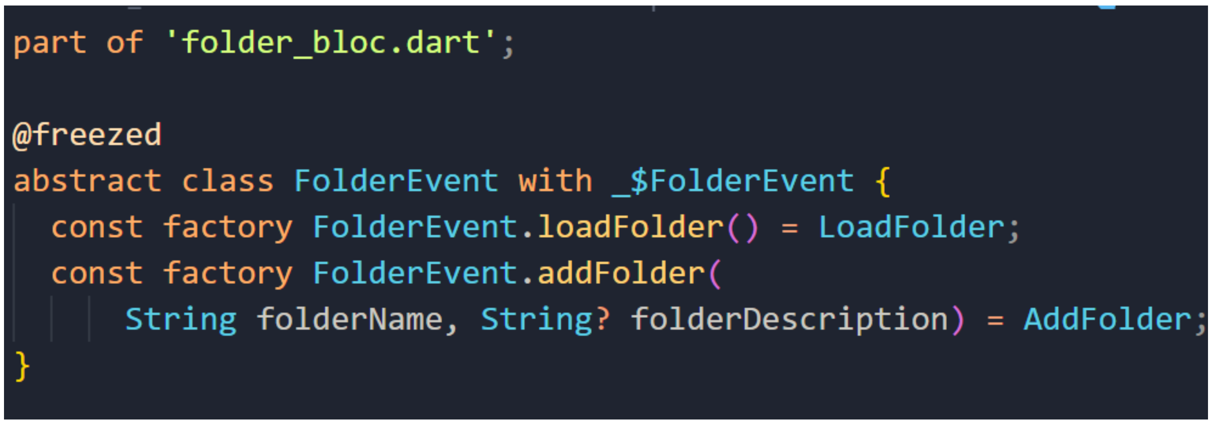
To implement the Bloc Pattern inside Flutter, we will first need to add some packages, these are flutter_bloc, freezed, freezed_annotation. We also generate some code to pre-define the data of each state of our bloc. We are taking an example of the folder feature, the Bloc file is called folder_bloc.dart, its state file is folder_state.dart and its event file is folder_event.dart



```
part of 'folder_bloc.dart';

@freezed
class FolderStateData with _$FolderStateData {
  const factory FolderStateData({
    @Default('') String error,
    @Default([]) List<Folder> folders,
  }) = _FolderStateData;
}

@freezed
abstract class FolderState with _$FolderState {
  const factory FolderState.initial(FolderStateData data) = FolderInitial;
  const factory FolderState.loading(FolderStateData data) = FolderLoading;
  const factory FolderState.loaded(FolderStateData data) = FolderLoaded;
  const factory FolderState.error(FolderStateData data) = FolderError;
  const factory FolderState.addFolderSuccess(FolderStateData data) =
      FolderAddFolderSuccess;
      const factory FolderState.addFolderFailure(FolderStateData data) = FolderAddFolderFailure;
}
```

Figure 2.5: Bloc State

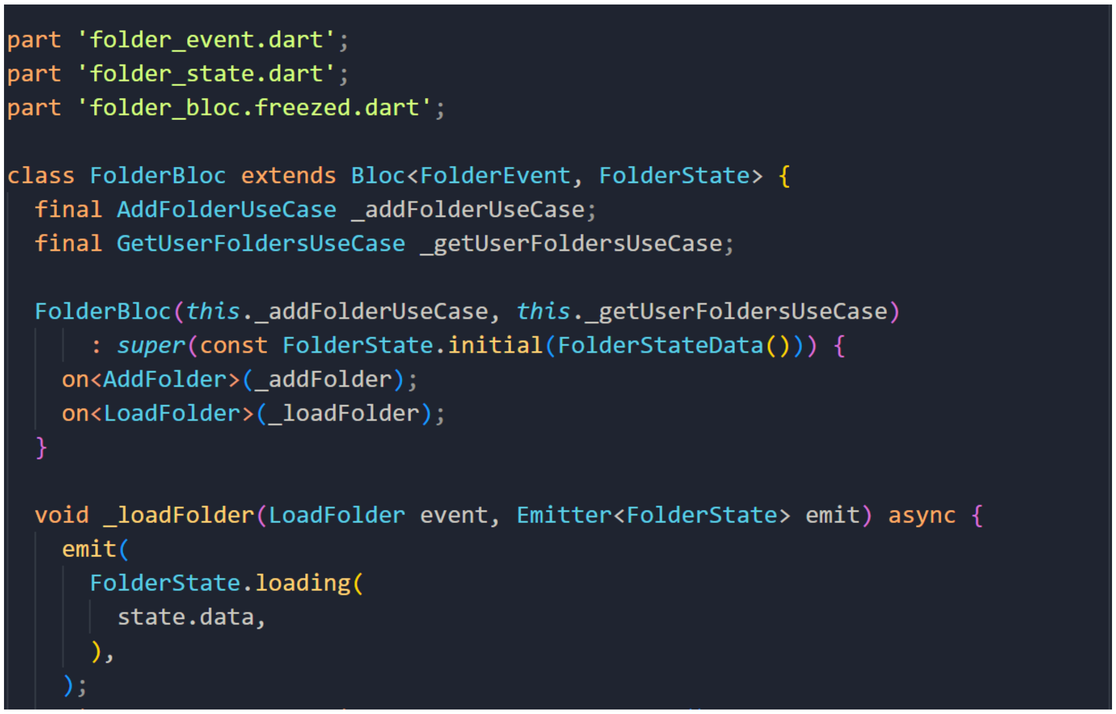
This is how we define our folder state, we have a state data to note the data of every state of bloc.



```dart
part of 'folder_bloc.dart';

@freezed
abstract class FolderEvent with _$FolderEvent {
  const factory FolderEvent.loadFolder() = LoadFolder;
  const factory FolderEvent.addFolder(
      String folderName, String? folderDescription) = AddFolder;
}
```

Figure 2.6: Bloc Event

This is the folder_event.dart file, it has two events when it comes to interacting with the view. Those two events when pushed into the bloc will execute the corresponding code that mapped to the event.



```dart
part 'folder_event.dart';
part 'folder_state.dart';
part 'folder_bloc.freezed.dart';

class FolderBloc extends Bloc<FolderEvent, FolderState> {
  final AddFolderUseCase _addFolderUseCase;
  final GetUserFoldersUseCase _getUserFoldersUseCase;

  FolderBloc(this._addFolderUseCase, this._getUserFoldersUseCase)
      : super(const FolderState.initial(FolderStateData())) {
    on<AddFolder>(_addFolder);
    on<LoadFolder>(_loadFolder);
  }

  void _loadFolder(LoadFolder event, Emitter<FolderState> emit) async {
    emit(
      FolderState.loading(
        state.data,
      ),
    );
```

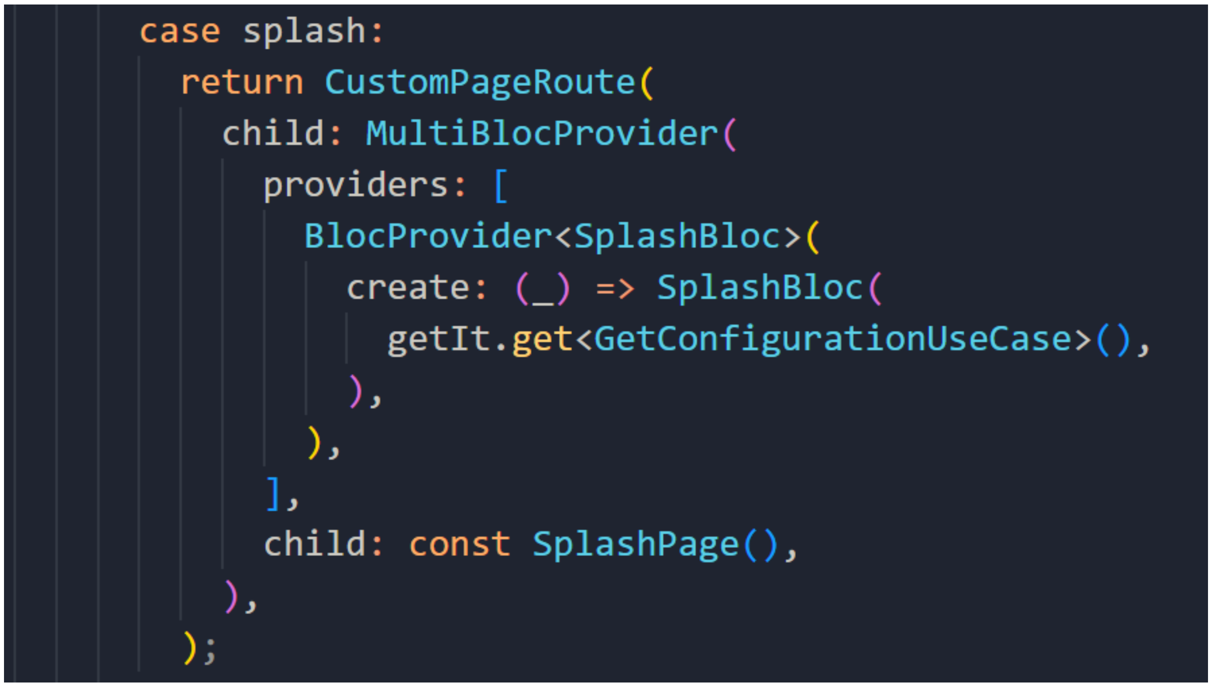
Figure 2.7: Bloc Class

This is the FolderBloc class, which holds the use cases of our clean architecture, and it also has the two lines of code that determine which method should be triggered on a new event. Make sure to add the folder_bloc.freezed.dart file part to generate the code using build runner for the two state and event file.

To add a new event to the state, you can use the following code as follows.

context.read<FolderBloc>().add(const LoadFolder());

This code will read the Bloc from the context of the widget tree, to add it inside the widget tree, you need a bloc provider whenever we navigate to the screen that needs that Bloc.

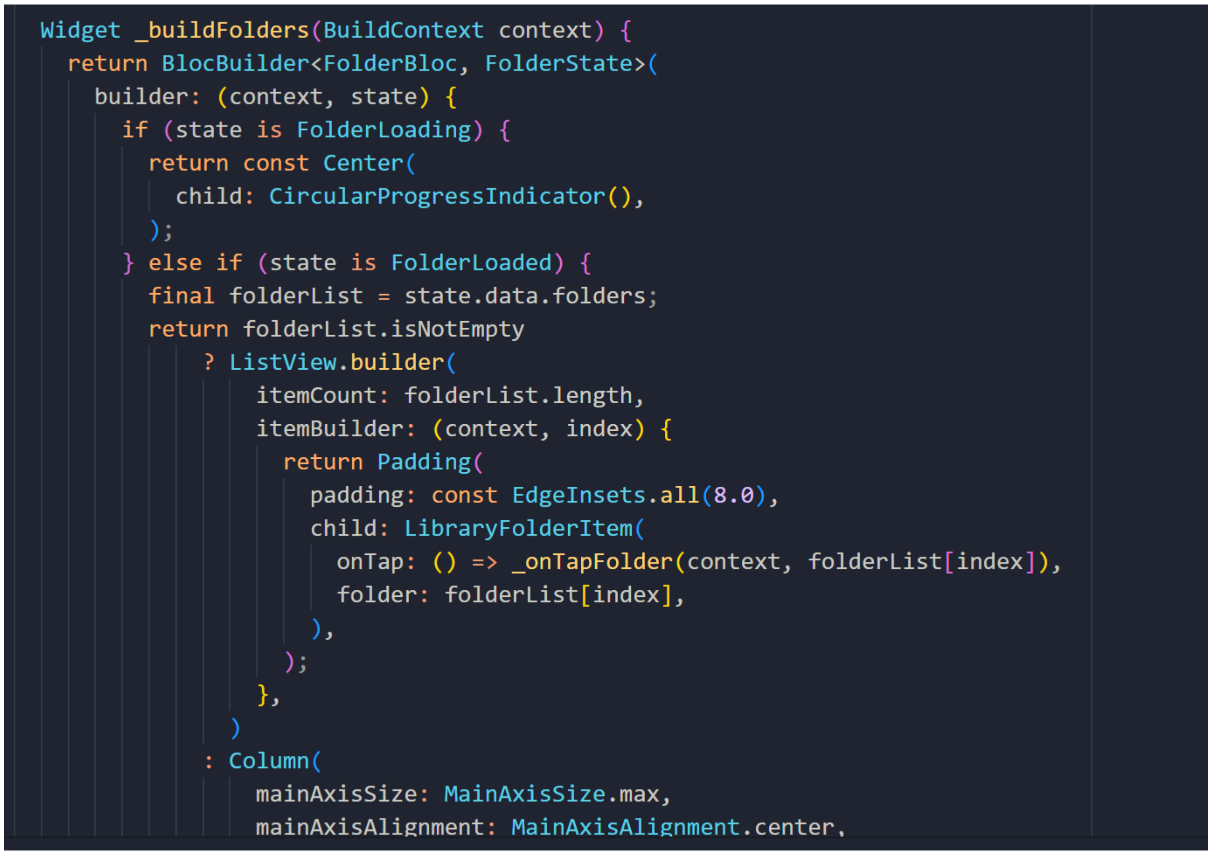


```
case splash:
  return CustomPageRoute(
    child: MultiBlocProvider(
      providers: [
        BlocProvider<SplashBloc>(
          create: (_) => SplashBloc(
            getIt.get<GetConfigurationUseCase>(),
          ),
        ),
      ],
      child: const SplashPage(),
    ),
  );
```

Figure 2.8: Provide a Bloc

This code provides a screen with a bloc provider, that bloc will stay until the end of the widget lifecycle of that screen, such as when the user pops back out of the stack. It is not the Folder Bloc but the process is essentially similar.

To create a corresponding view of the screen, we have three options, they are BlocBuilder, BlocListener and BlocConsumer. BlocBuilder is to build the view of the screen based on state, BlocListener is used to call some methods to show popups or warnings and BlocConsumer is the combination of the two.



```
Widget _buildFolders(BuildContext context) {
  return BlocBuilder<FolderBloc, FolderState>(
    builder: (context, state) {
      if (state is FolderLoading) {
        return const Center(
          child: CircularProgressIndicator(),
        );
      } else if (state is FolderLoaded) {
        final folderList = state.data.folders;
        return folderList.isNotEmpty
            ? ListView.builder(
                itemCount: folderList.length,
                itemBuilder: (context, index) {
                  return Padding(
                    padding: const EdgeInsets.all(8.0),
                    child: LibraryFolderItem(
                      onTap: () => _onTapFolder(context, folderList[index]),
                      folder: folderList[index],
                    ),
                  );
                },
              )
            : Column(
                mainAxisSize: MainAxisSize.max,
                mainAxisAlignment: MainAxisAlignment.center,
```

Figure 2.9: Using BlocBuilder

For this BlocBuilder, we see that if the state is loading then we only return a circular progress indicator in the middle of the screen. Or else the screen will load the folders and return a list view of those folders.

2.2.4 google_sign_in

To implement google sign in inside Flutter, we would generally use Firebase since it is developed and maintained by Google and their apis are already provided and well supported.

First of all, we need to create a firebase project and connect it with your Flutter project. For simplicity, we will use Flutter Fire CLI which is a command line that is specialised to integrate your firebase project into Flutter.

To install the tool, we run the following command inside your command prompt (CMD):

npm install -g firebase-tools

And activate it also:

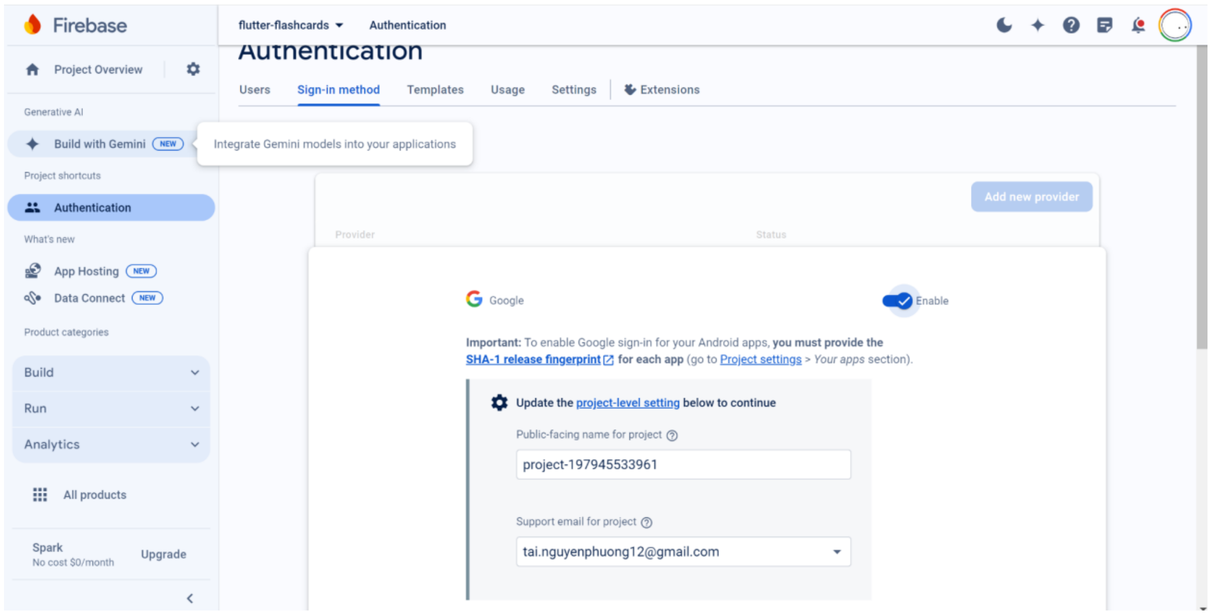
dart pub global activate flutterfire_cli

To configure firebase to your project, navigate the prompt to the project and use:

flutterfire configure

From there, you can select your project, the app's supported platforms and the process will be automated by flutterfire

After that, we navigate to the firebase website and go into our project, you can select the Authentication feature and enable Google Sign In.



Google ✓ Enable

Important: To enable Google sign-in for your Android apps, **you must provide the SHA-1 release fingerprint for each app** (go to Project settings > *Your apps* section).

Update the project-level setting below to continue

Public-facing name for project: project-197945533961

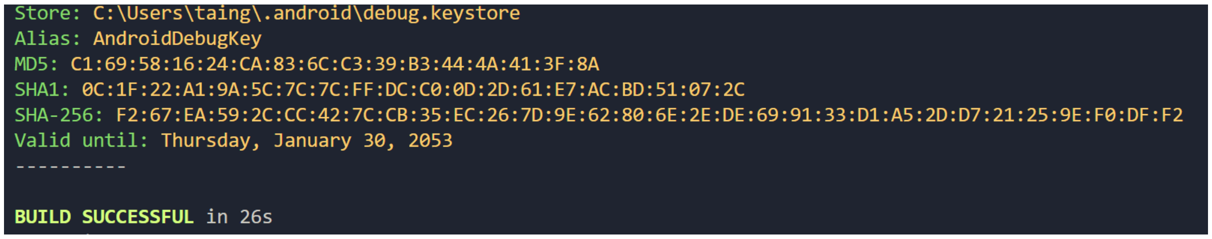
Support email for project: tai.nguyenphuong12@gmail.com

Figure 2.9: Enabling Google Login

Then, navigate to Project overview - Project settings, inside of this section, make sure to provide the android app the SHA-1 and SHA-256 key. For this, we can navigate to the android folder of our Flutter project and run the command of:

.\gradlew signReport

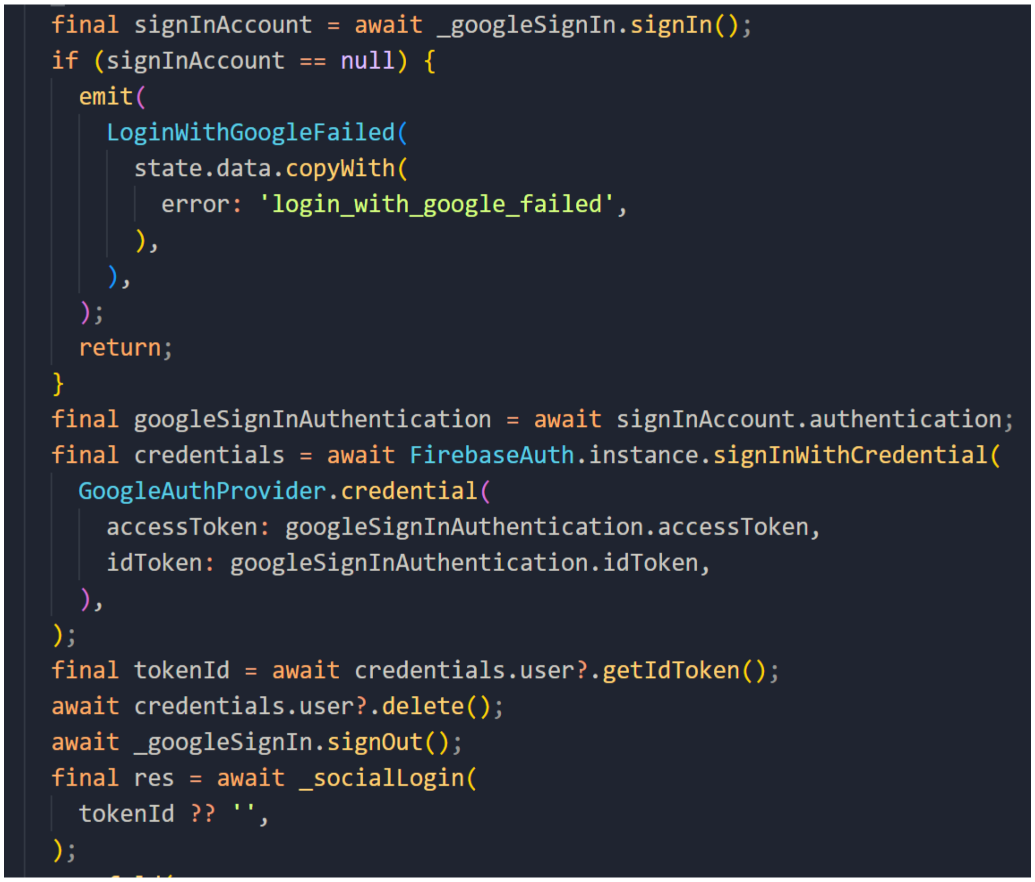
It will return the two necessary SHA keys for us to use inside Firebase.



```
Store: C:\Users\taing\.android\debug.keystore
Alias: AndroidDebugKey
MD5: C1:69:58:16:24:CA:83:6C:C3:39:B3:44:4A:41:3F:8A
SHA1: 0C:1F:22:A1:9A:5C:7C:7C:FF:DC:C0:0D:2D:61:E7:AC:BD:51:07:2C
SHA-256: F2:67:EA:59:2C:CC:42:7C:CB:35:EC:26:7D:9E:62:80:6E:2E:DE:69:91:33:D1:A5:2D:D7:21:25:9E:F0:DF:F2
Valid until: Thursday, January 30, 2053
----------

BUILD SUCCESSFUL in 26s
```

Figure 2.10: SHA keys

After this is done, make sure to add google_sign_in and some firebase packages like firebase_core and firebase_auth. We are going to need those two packages to help us implement Google Sign In.



```
final signInAccount = await _googleSignIn.signIn();
if (signInAccount == null) {
  emit(
    LoginWithGoogleFailed(
      state.data.copyWith(
        error: 'login_with_google_failed',
      ),
    ),
  );
  return;
}
final googleSignInAuthentication = await signInAccount.authentication;
final credentials = await FirebaseAuth.instance.signInWithCredential(
  GoogleAuthProvider.credential(
    accessToken: googleSignInAuthentication.accessToken,
    idToken: googleSignInAuthentication.idToken,
  ),
);
final tokenId = await credentials.user?.getIdToken();
await credentials.user?.delete();
await _googleSignIn.signOut();
final res = await _socialLogin(
  tokenId ?? '',
);
```

Figure 2.11: Implementing Google Login

We are essentially obtaining the Id Token from Firebase to send to our written backend, what our backend essentially does is to extract the information from the token to either register or sign in to our apps.

2.3 Database Design

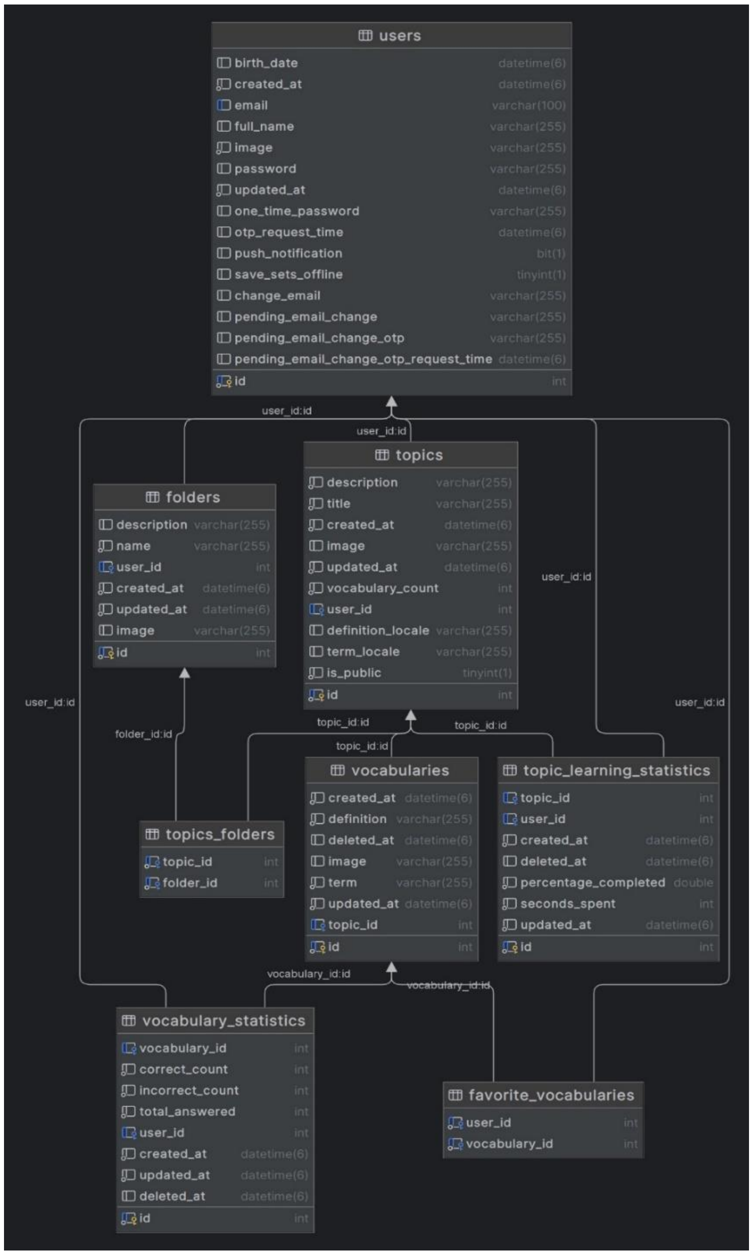


Figure 2.12: Datebase Design

CHAPTER 3: IMPLEMENTATION

In this chapter, we will go through each of the screens that our group has implemented.

Java Spring Boot will serve as our backend framework, and MySQL is our chosen database.

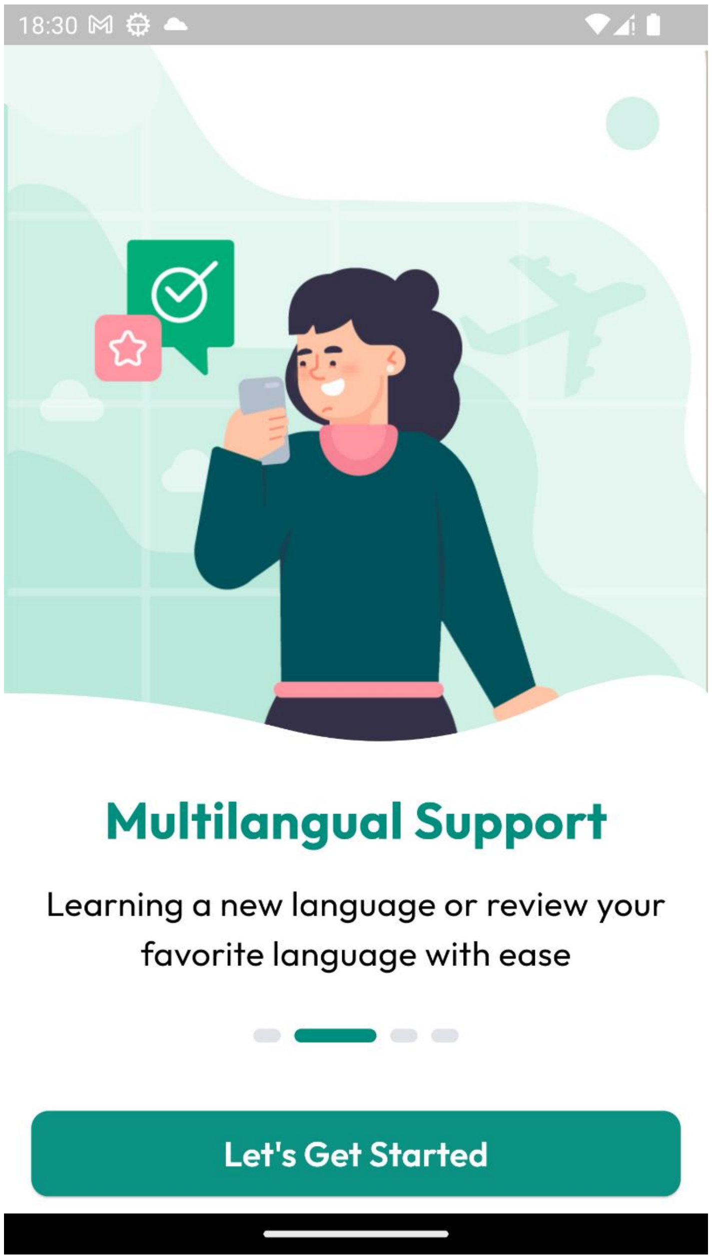

18:30
Multilangual Support
Learning a new language or review your favorite language with ease
Let's Get Started

Figure 3.1: Onboarding Screen

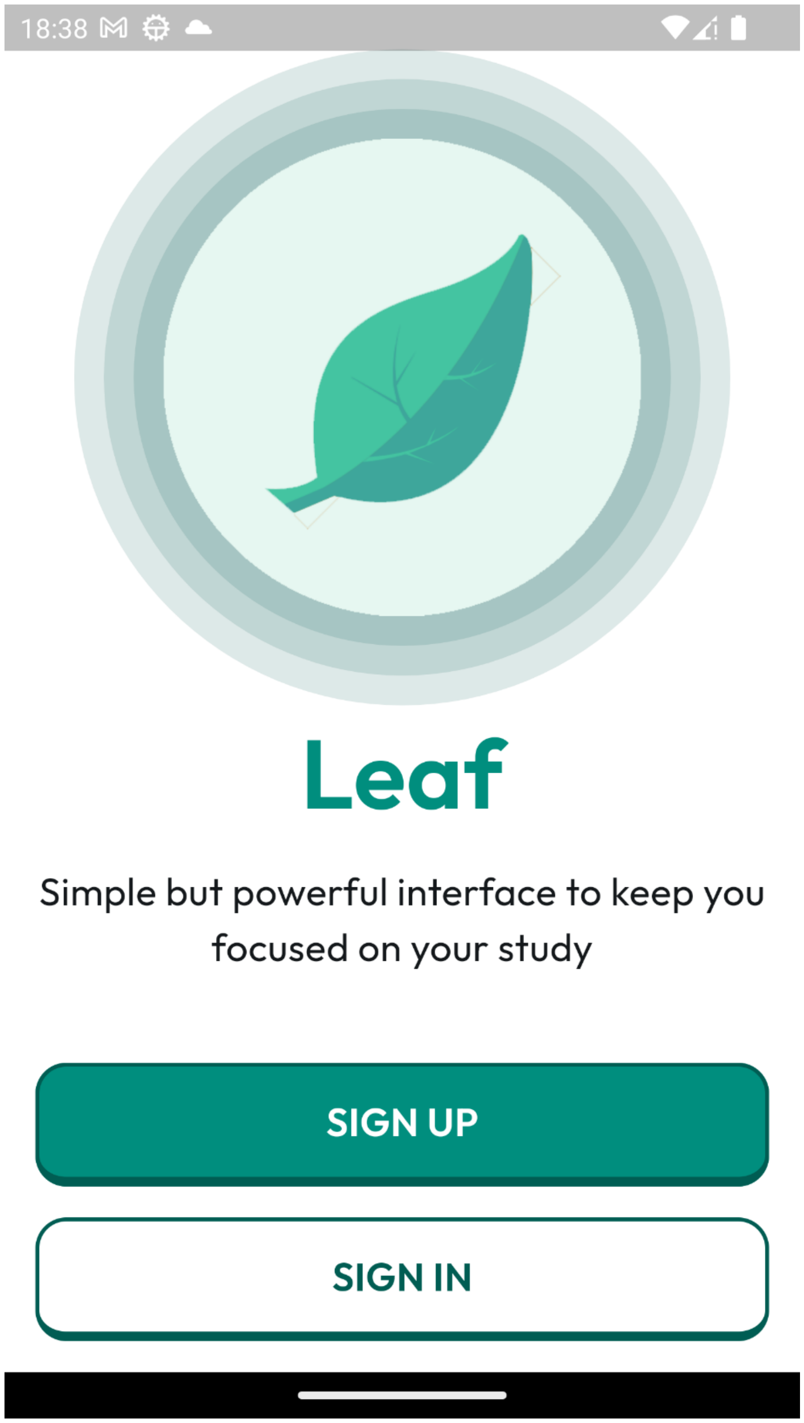

18:38
Leaf
Simple but powerful interface to keep you focused on your study
SIGN UP
SIGN IN

Figure 3.2: Authentication Options Screen

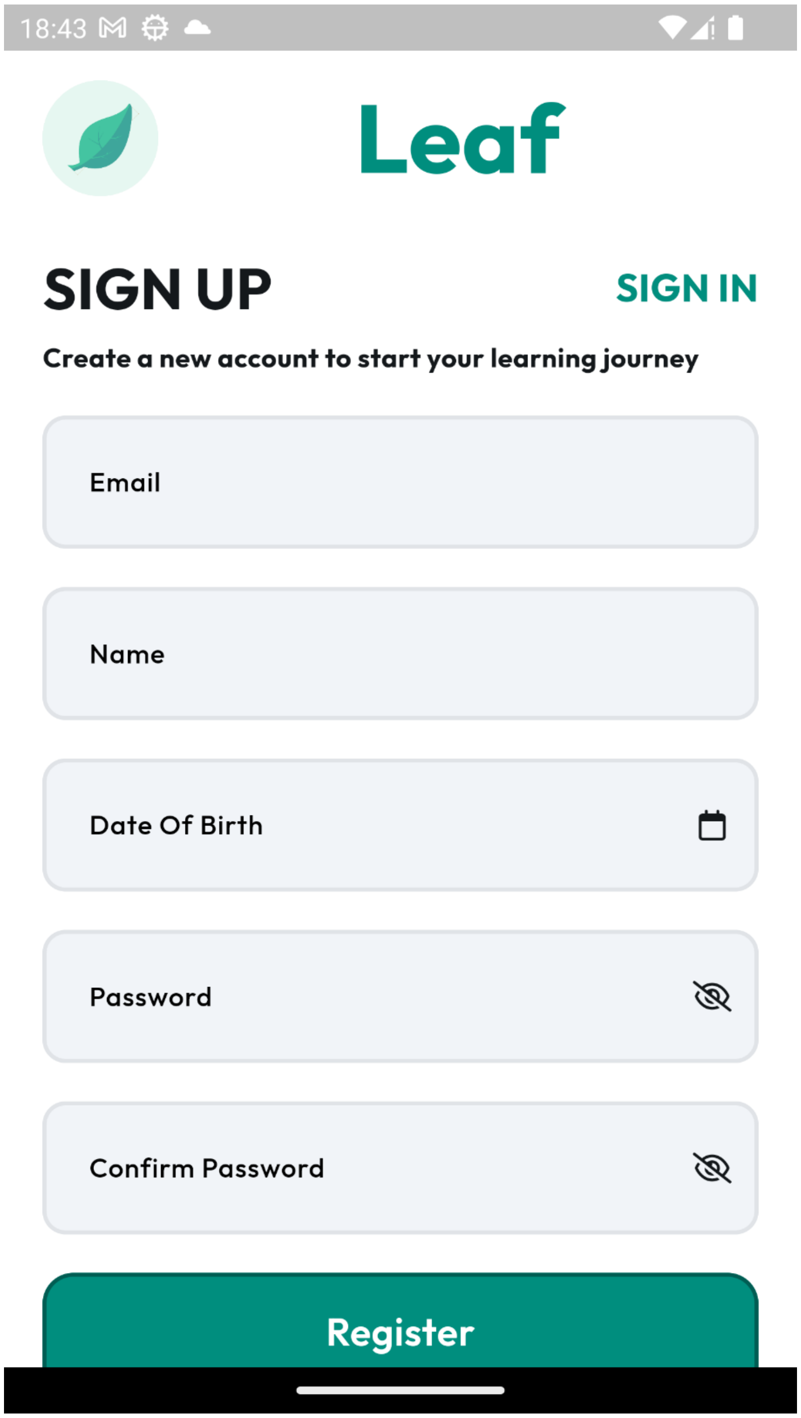


Figure 3.3: Sign up Screen

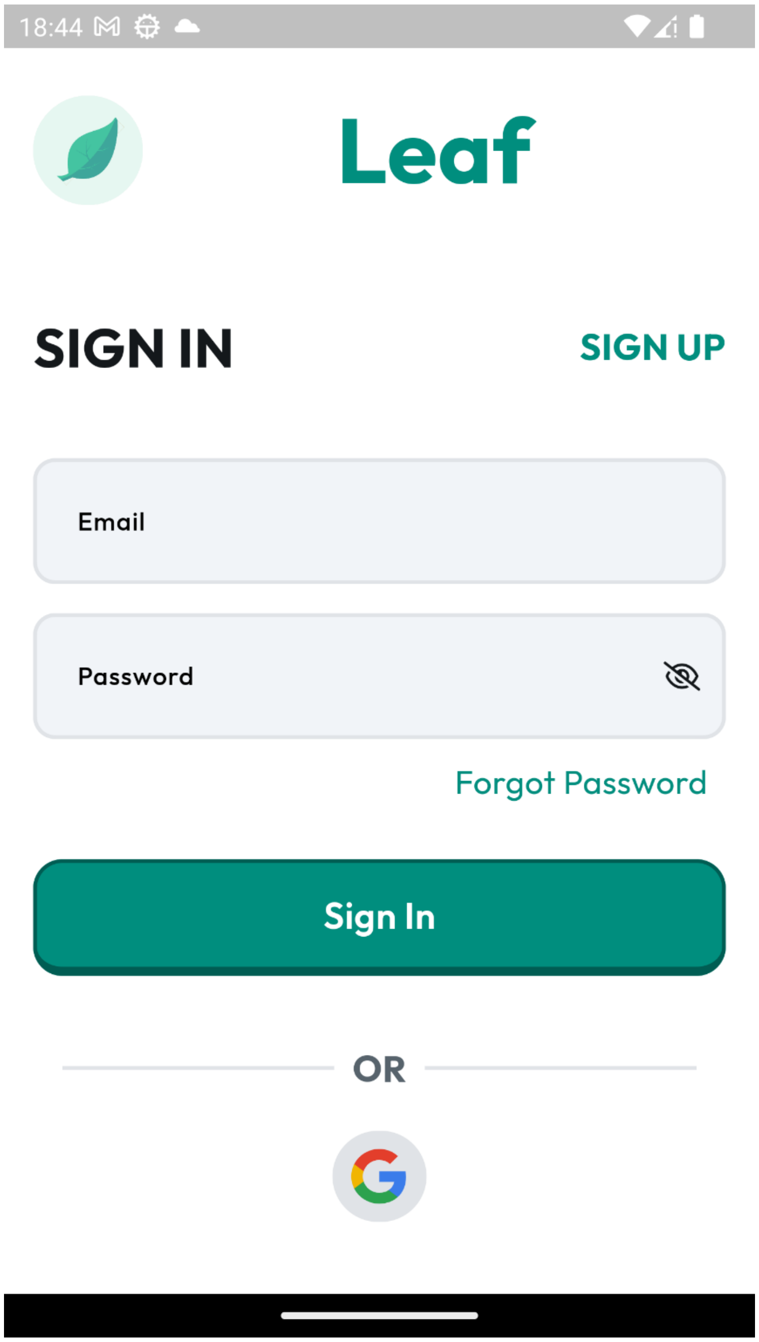


Figure 3.4: Sign In Screen

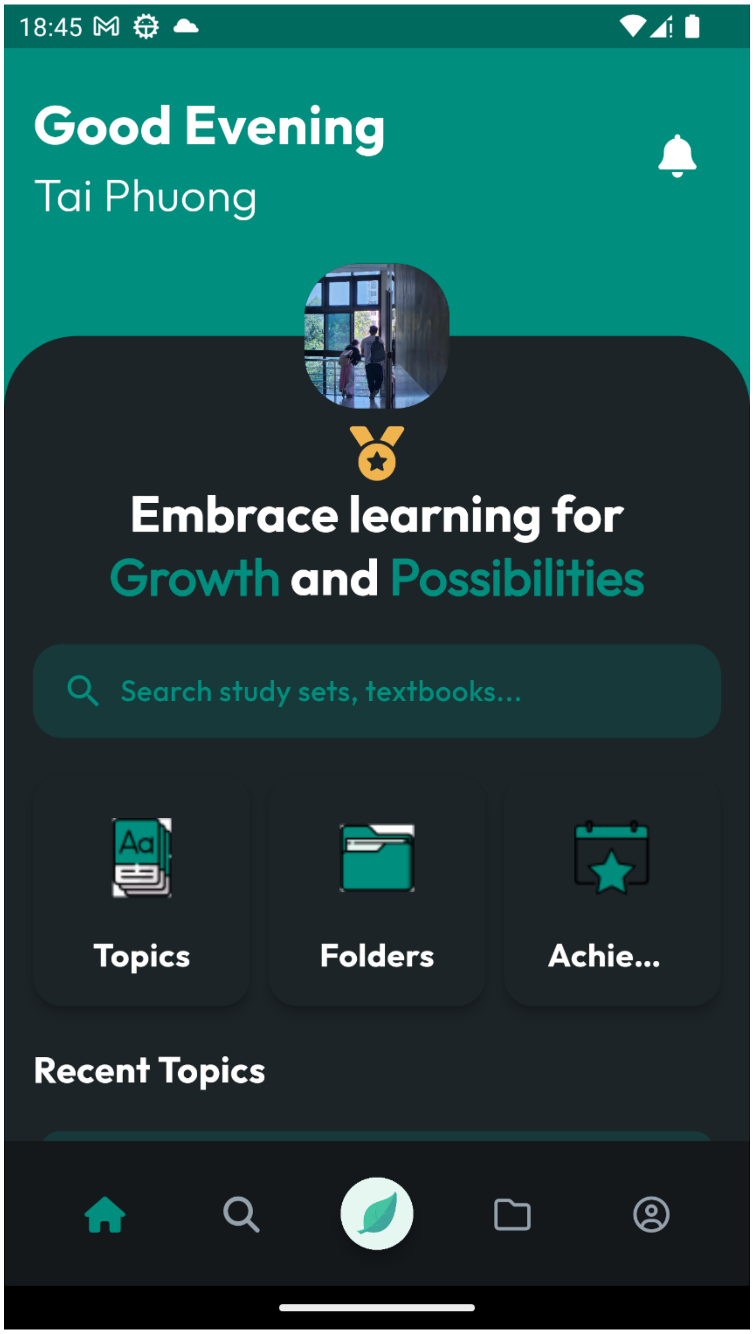


Figure 3.5: Home Screen

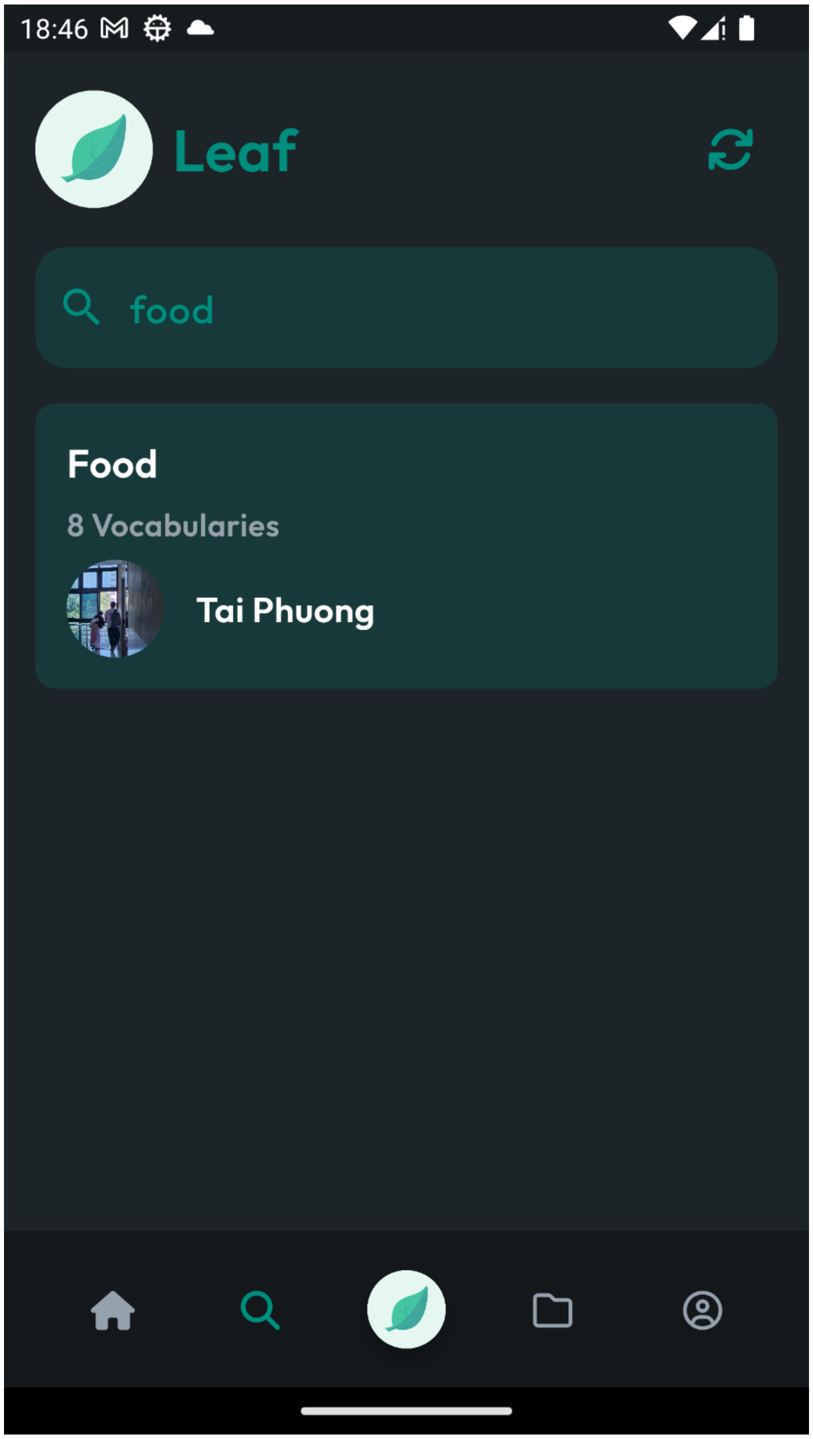


Figure 3.6: Search Screen

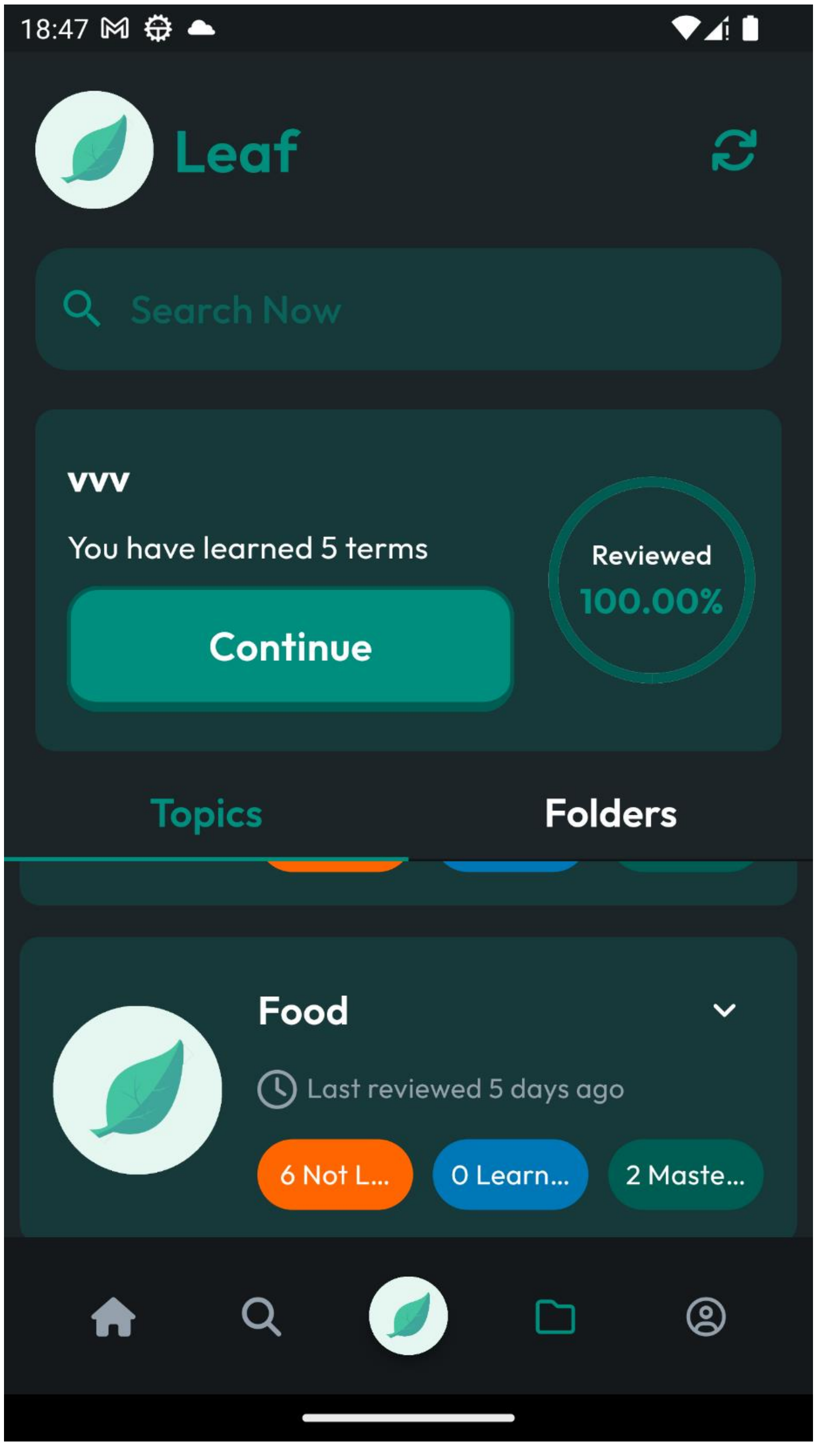
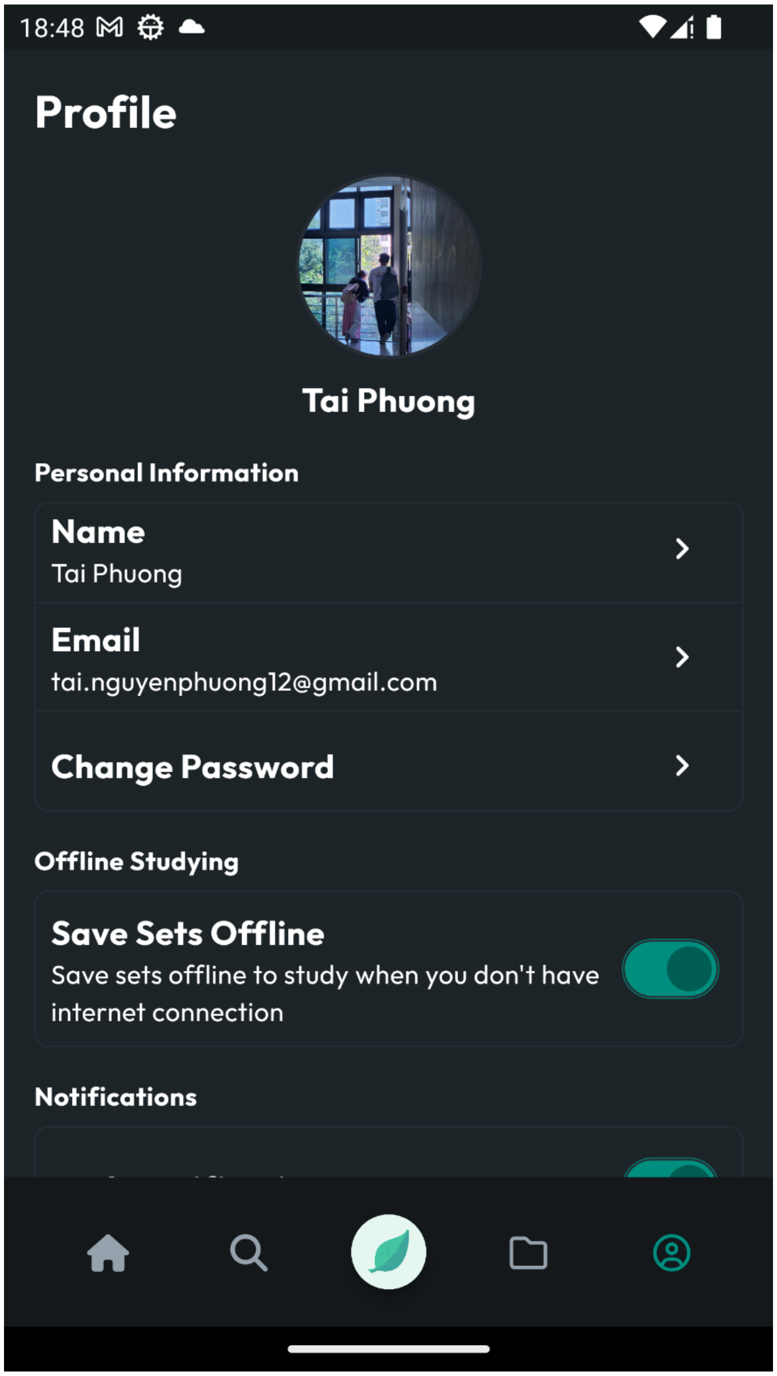


Figure 3.7: Library Screen



18:48

Profile

Tai Phuong

Personal Information

Name
Tai Phuong

Email
tai.nguyenphuong12@gmail.com

Change Password

Offline Studying

Save Sets Offline
Save sets offline to study when you don't have internet connection

Notifications

Figure 3.7: Profile Screen

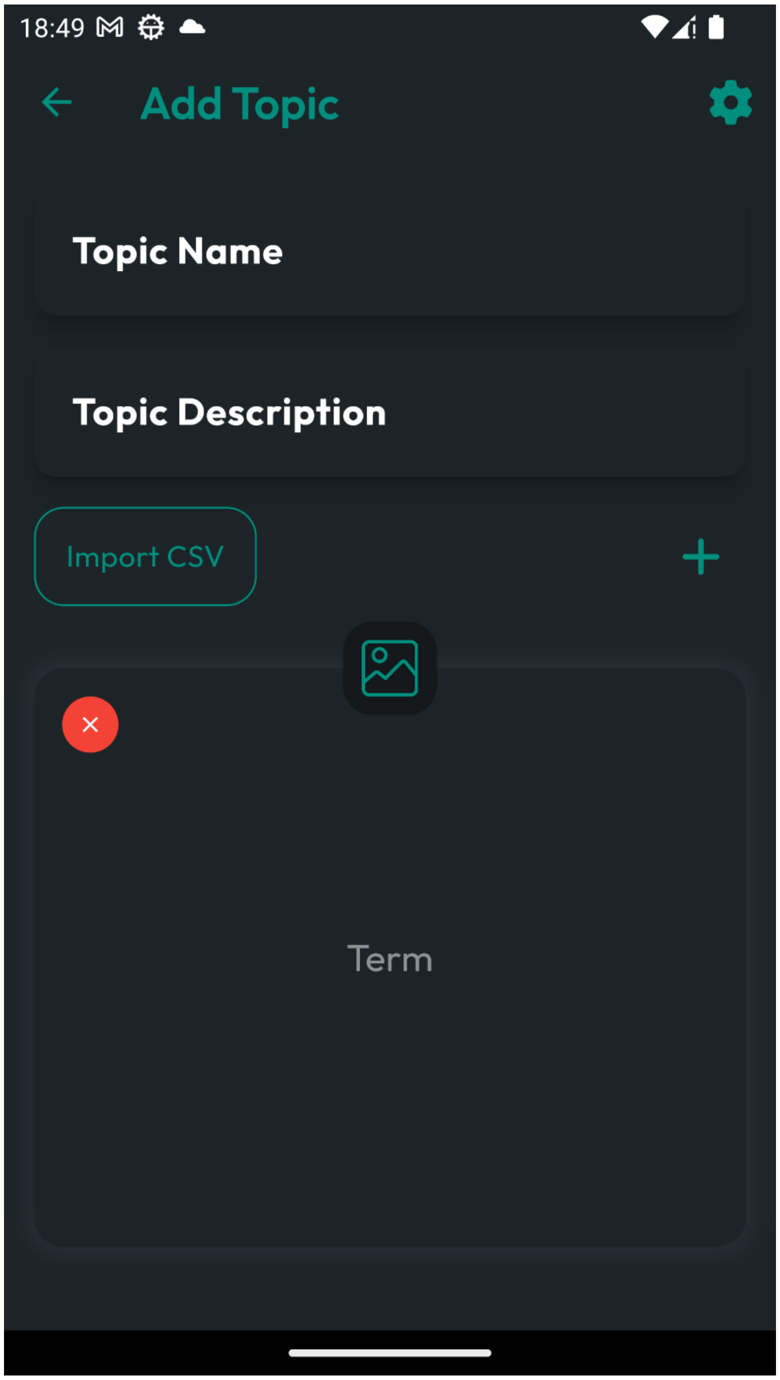


Figure 3.8: Add Topic Screen

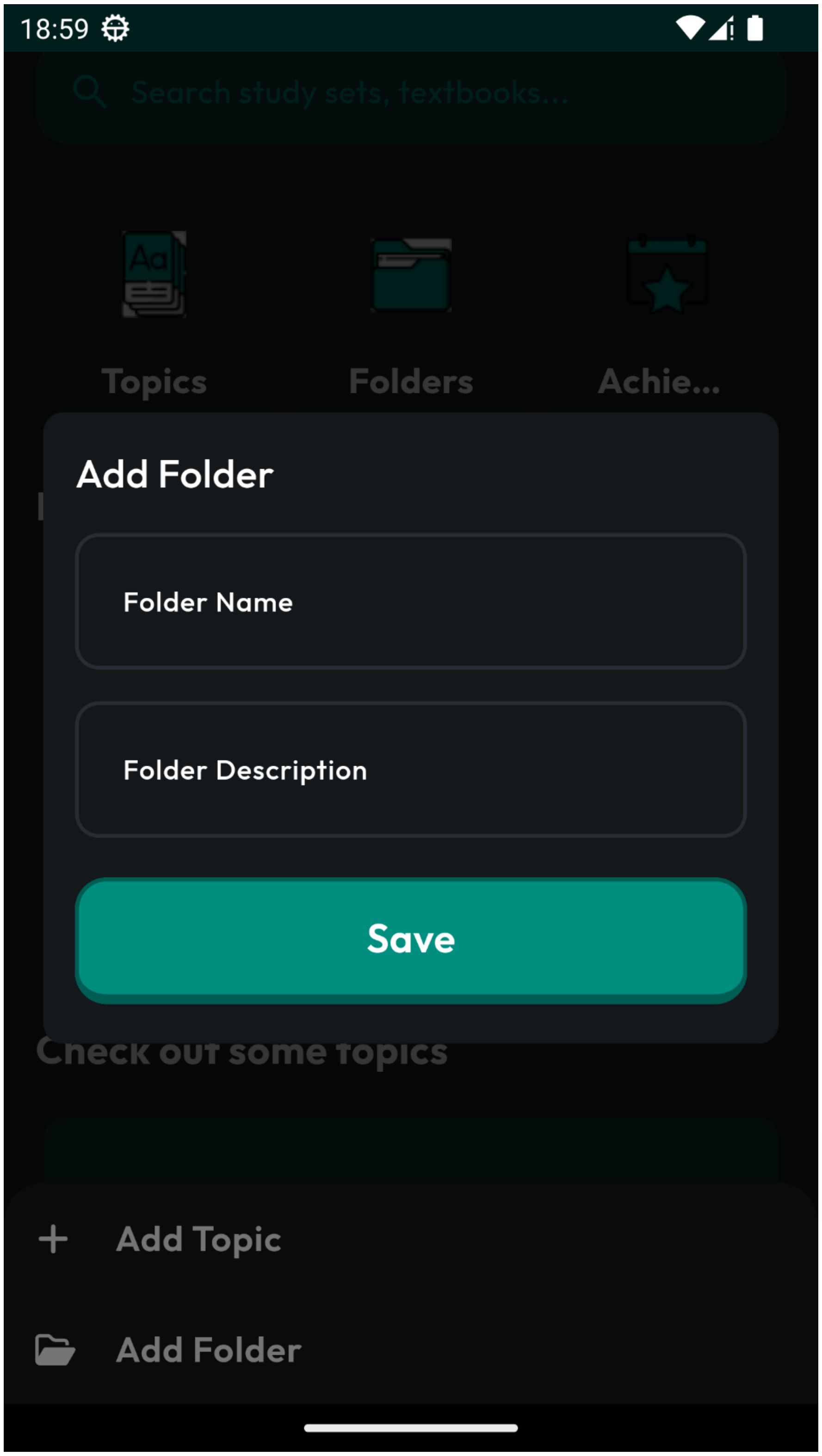


Figure 3.9: Add Folder Dialog

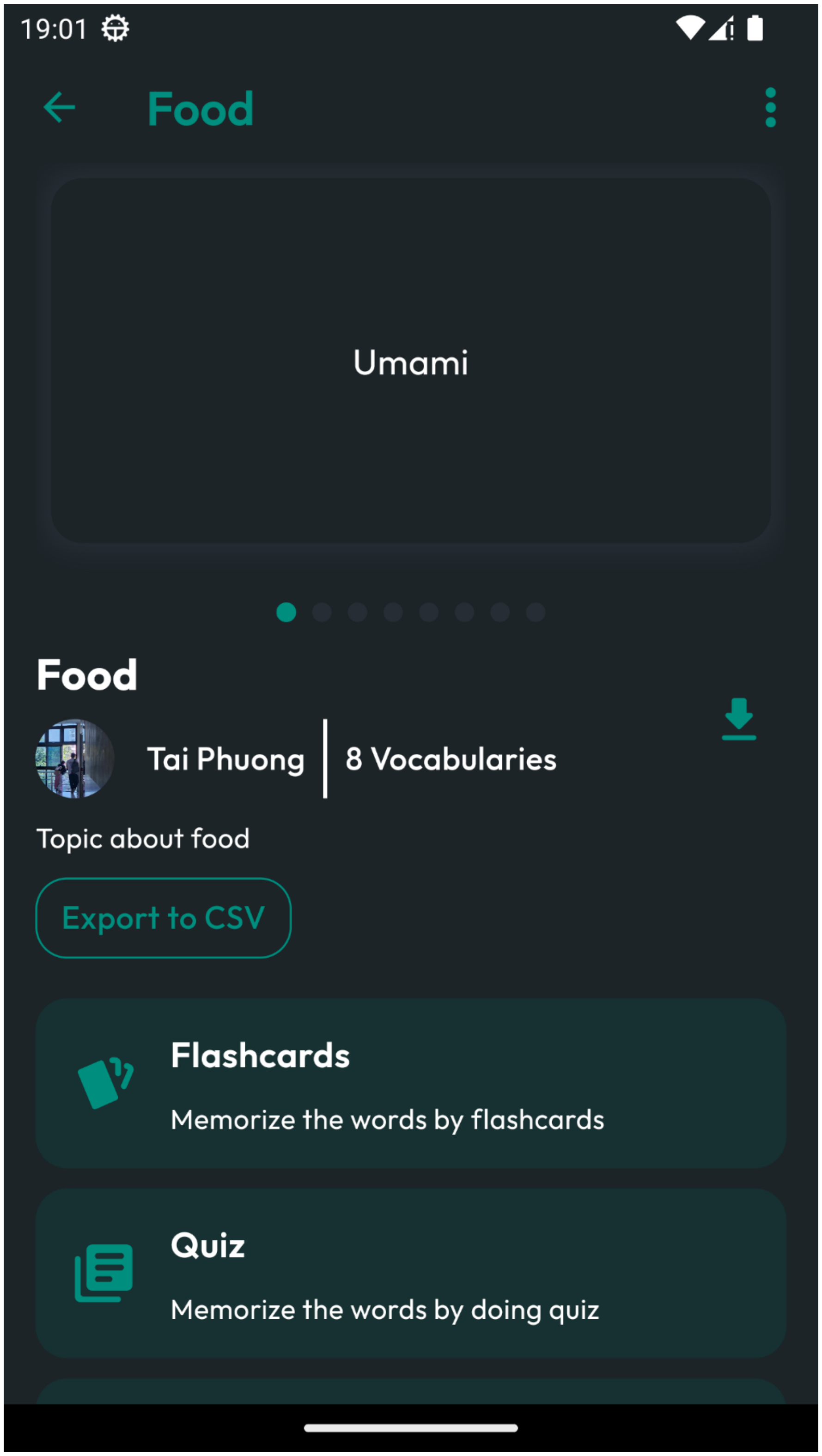
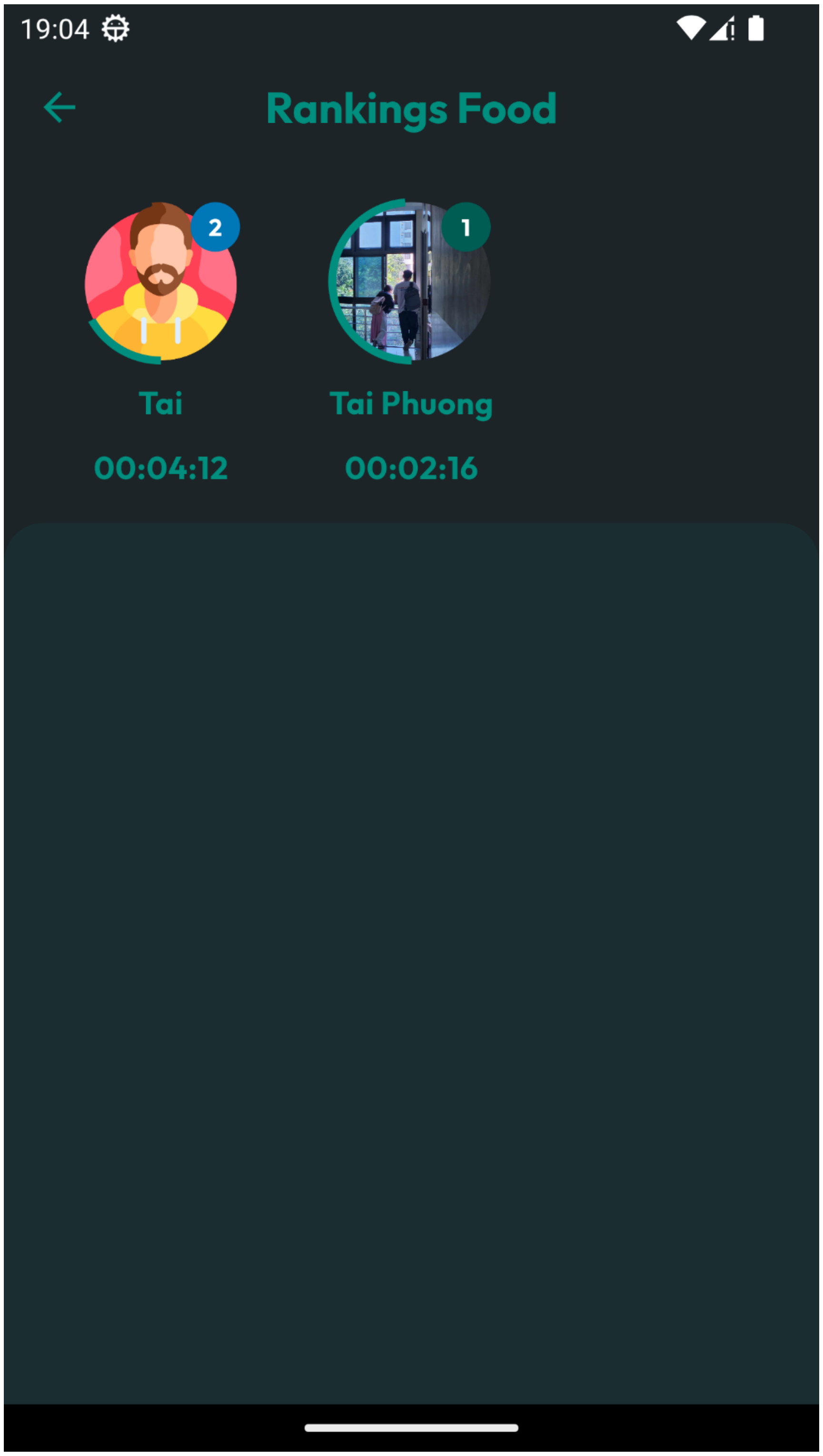


Figure 3.10: Topic Details Screen



19:04

Rankings Food

2	1
Tai	Tai Phuong
00:04:12	00:02:16

Figure 3.11: Topic Learning Rankings

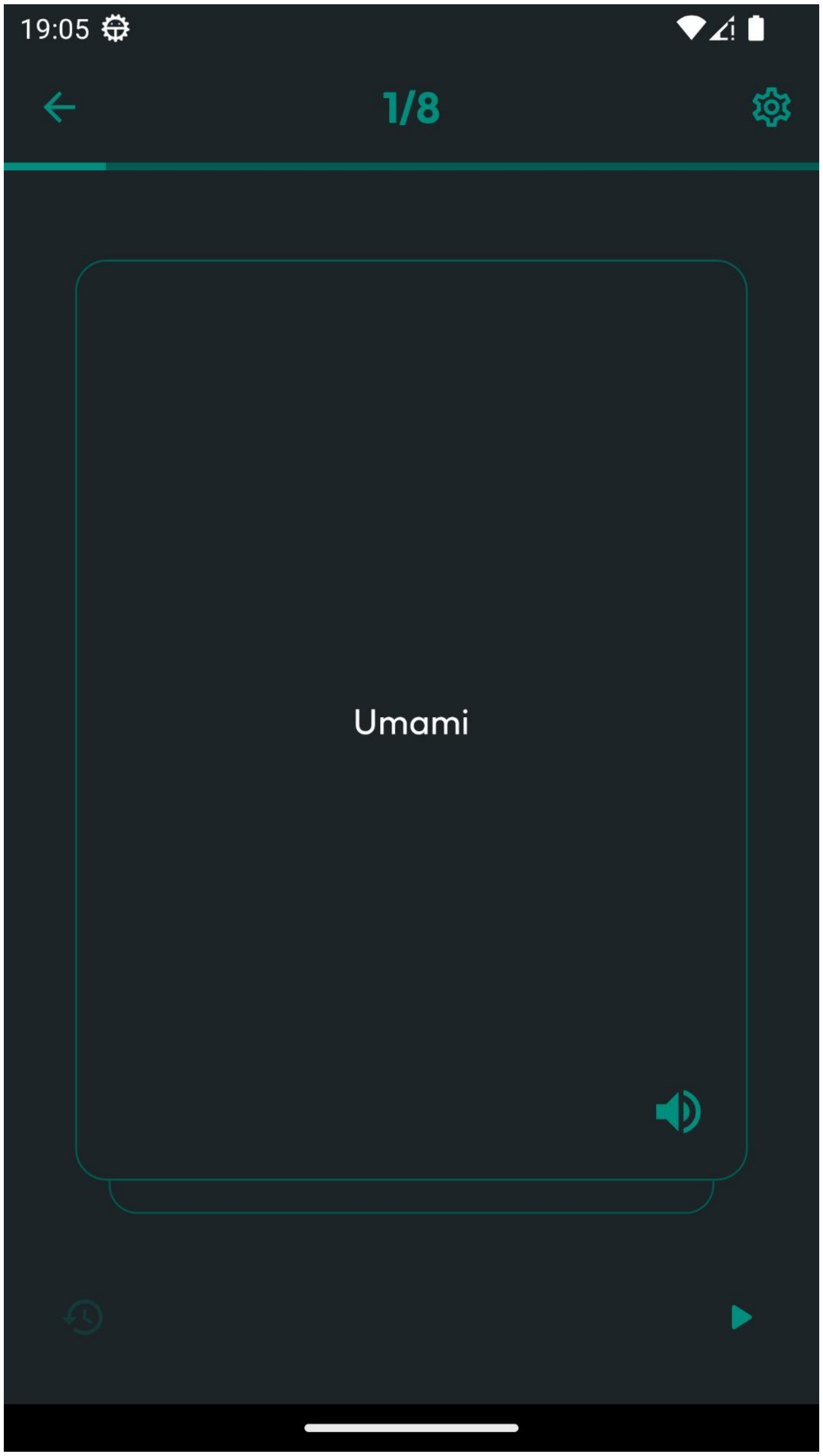
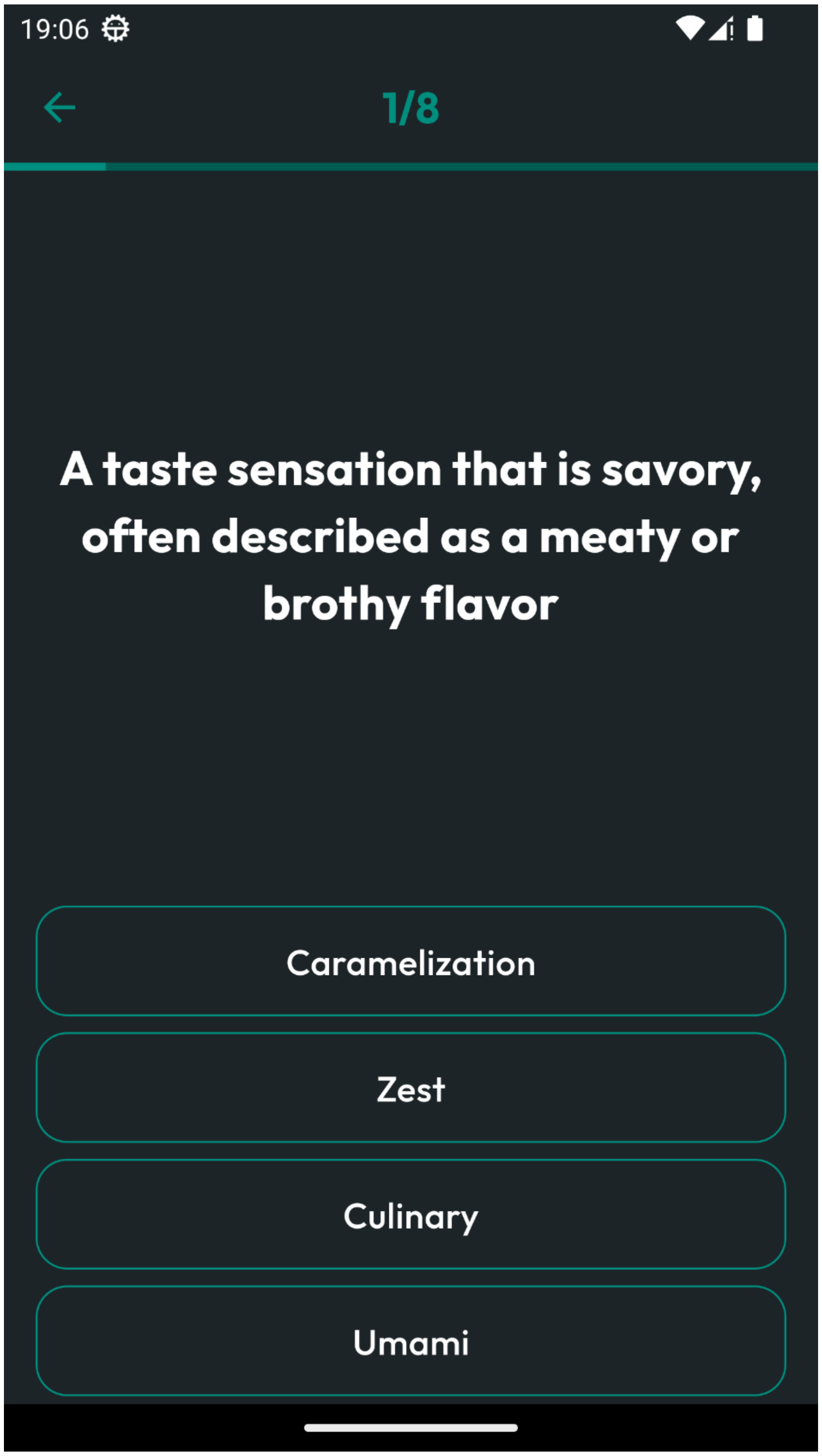
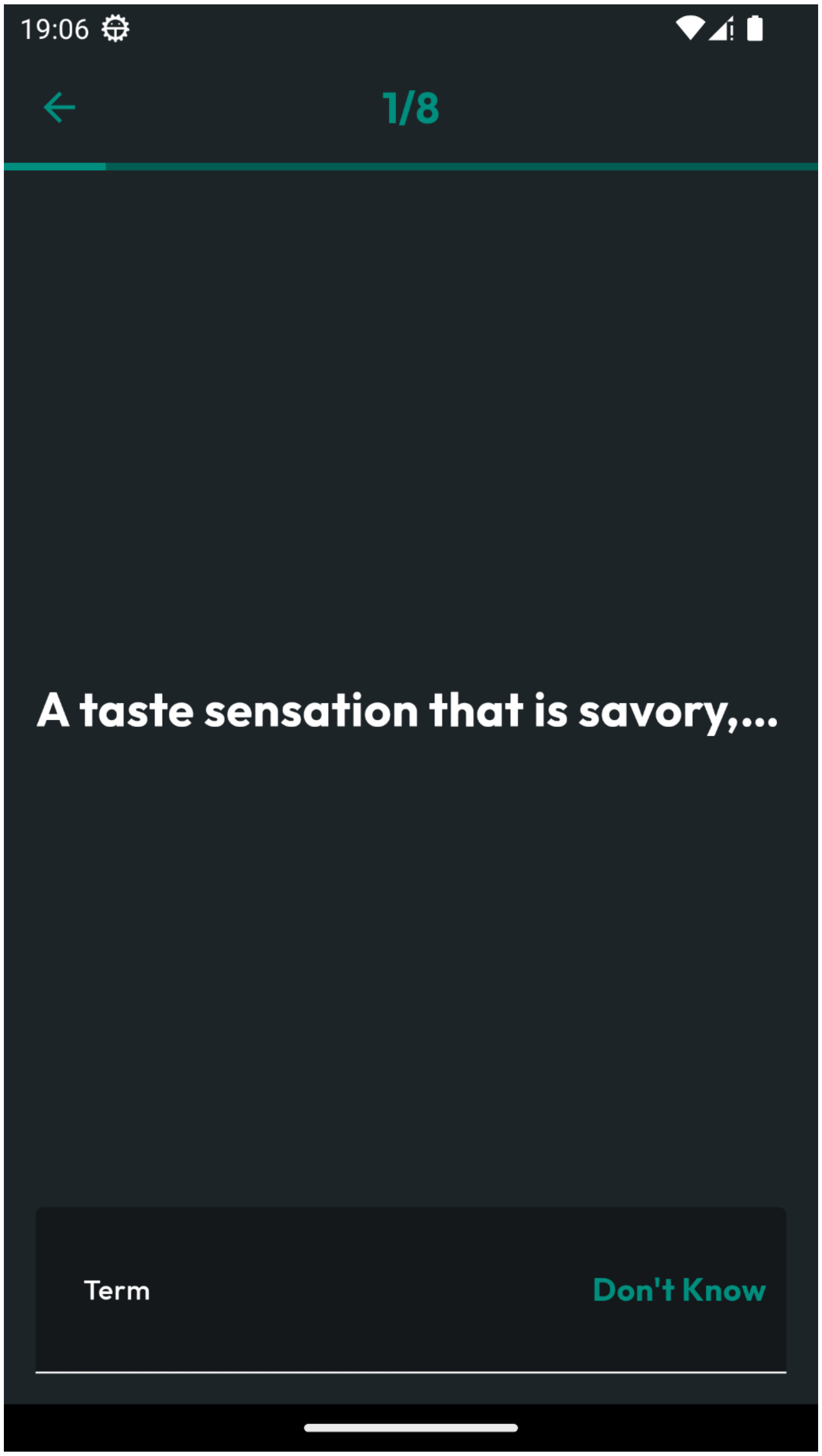


Figure 3.12: Flashcard Learning Screen



A taste sensation that is savory, often described as a meaty or brothy flavor

Figure 3.13: Quiz Learning Screen



A taste sensation that is savory,...

Figure 3.14: Typing Learning Screen

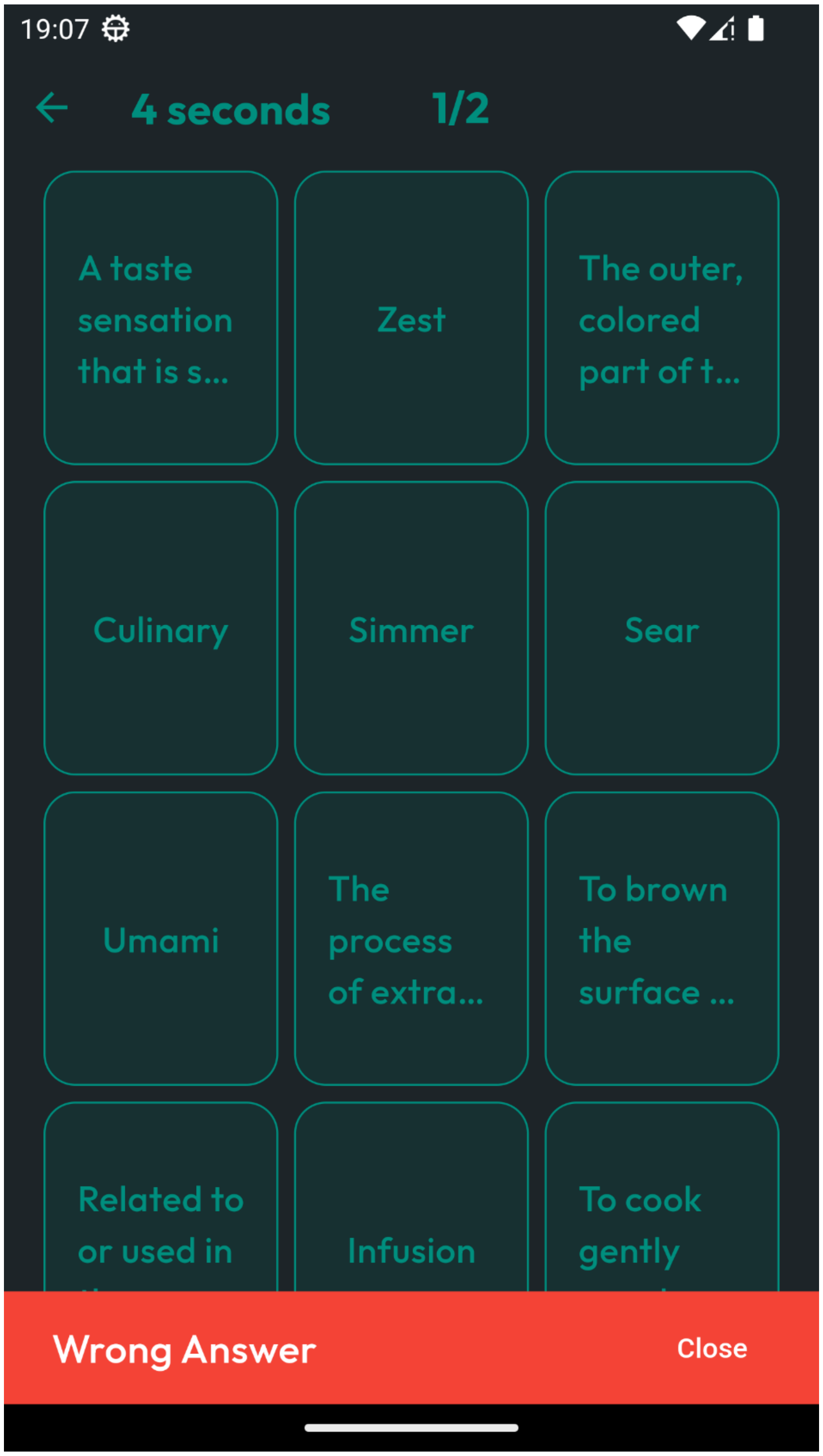


Figure 3.15: Matching Learning Screen

REFERENCES

English

[1]. Marco L. Napoli [2019], Beginning Flutter: A Hands On Guide To App Development, 1st Edition, Wrox Press, New Jersey.

[2]. Eric Windmill [2019], Flutter in Action, 1st Edition, Manning Publications, New York.

[3]. Prajyot Mainkar, Salvatore Giordano [2019], Google Flutter Mobile Development Quick Start Guide, 1st Edition, Packt, United Kingdom.

[4]. Ivo Balbaert, Dzenan Ridjanovic [2015], Learning Dart, 2nd Edition, Packt, United Kingdom.

[5]. Gilad Bracha [2015], The Dart Programming Language, Addison-Wesley, Boston.

Tutorials | Dart

Docs | Flutter