Politecnico di Milano

DESIGN AND IMPLEMENTATION OF MOBILE APPLICATIONS

AY 2023-2024

FlashcardAI

Design Document (DD)

Giacomo Verdicchio, Lorenc Leci

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose of the app

A while ago, we where searching Play Store trying to find an app based on the flashcard learning method (that we particularly like because it's a funny way of learning and studying) but we weren't able to find an app that gave us, only after a very short amount of time, the possibility to start playing. All the app currently in the store require an overhead of time to create the flashcard and that is the main reason why anyone want to use them. Because are very slow at the beginning. That was the reason why we decided it was time to address this need and create our own app. The result is our innovative FlashcardsAI app, designed to improve the way you study and learn. For those unfamiliar with flashcard apps, they serve as digital tools where users can create, interact with, and review sets of virtual flashcards. Our app goes beyond the basics by offering a comprehensive suite of features.

You can create personalized flashcards effortlessly, play with them in various interactive modes, and track your progress through detailed performance metrics. The app includes a sophisticated system that not only memorizes your errors but also allows you to review, revise, and filter them according to your needs. Each session provides instant feedback, showcasing the points earned, and even presents a concise histogram summarizing your monthly learning trends.

But what truly sets our app apart is its integration of artificial intelligence (AI) to enhance your learning experience. Imagine skipping the tedious task of creating individual flashcards, which typically consumes around 5 minutes each. With our app, AI steps in to generate complete sets based on specific parameters you provide, saving you time and effort.

Moreover, ChatGPT plays a pivotal role as the app's main feature. Users can engage with ChatGPT to receive explanations and insights into various topics. By using ChatGPT's capabilities, users gain a clearer understanding of their study material and uncover additional nuances that contribute to a richer learning experience.

In essence, our app combines the convenience of modern technology with advanced AI to deliver a truly personalized and efficient learning tool. Whether you're studying for exams, learning a new language, or mastering complex concepts, our flashcard app is here to optimize your learning journey.

1.2 Purpose of the document

This document outlines the design decisions that were made and describes how the core functionalities of the application have been put into practice. Following an introduction to the features available to users, the document provides an overview of the application's architecture, the underlying database, and the imported libraries. It then proceeds with a detailed look at the User Interface. Lastly, the document discusses the testing phase of the application and the future improvements.

1.3 Features

1.3.1 Standard managing of the decks

As all the other app, our FlashcardsAI allow us to create and manage sets of decks, that we decided to call subjects. In particular we can create,modify and delete each subject,deck and flashcard that we want

1.3.2 AI usage to enahnce our sills

This app uses the AI to reduce the times of creation of the subjects/decks/flashcards. This is very important, because, as I said in the introduction, the initial barrier of the usage of this kind of learning method is the setup of the games, so of the flashcard to play with. Now instead with this app you can directly focus on your game, because the deck needed to play will be generating in some seconds. Another important role in which the AI come and help us is that it gives us the insights for each questions that we screw up, making easier to us to memorize our mistakes and learn from it.

1.3.3 Personal records

Each user will have it's own set of data saved both in local storage and in firebase. The data saved are: the set of subjects, decks and flashcards (see section on the architecture to understand better how the app is structure from a technical point of view), all the played set of flashcards and the API_KEY used to acces to the ChatGPT services

1.3.4 Possibility to improve our personal knowledge

Because of the possibility to store our past games we can also use this knowledge to learn from our error. This is exactly one of the important feature of this app that allows the user, in many redundants way, to use some functionalities, such as the Calendar or the HistoryOfError, to see our previous games. Moreover there are also 2 dedicated functionalities on this need: the PlayWithPastErrors and the AskDescriptionToAI. Both help us come back again on our path of errors to learn from them and grow. In particular the first allow the user to replay 5 flashcard choosen among the wrong ones, sorted by subjects or decks. The second allow to ask to the AI more details on the topic of the wrong answer so that you can have a more specific and detailed answer, enlarging our personal knowledge.

1.3.5 Interoperability

With our design of the storage we can rely on a backup (using firebase) that enables us to play on different devices in a consisten manner. So in this way if you want to play with your phone when you are moving through Milan with the metro and then come back again playing on your tablet, here you can. The only limitation is that, as it is implemented now, you can only transfer your data regarding subject/decks and flashcards. But do not fear, the update is coming and will fix this !

2 Dependencies

The table presented below enumerates the most important components that we integrated into the project and elucidates their respective functions within the application. This comprehensive overview provides insight into how each imported element contributes to the functionality and performance of the app.

Packages	Usage
Firebase	Enables access to FirebaseCore, FirebaseAuth and FirebaseFirestore APIs used for the authentication part and the storing part
GoogleSignIn Authentication	Users securely access the app by logging in with Google, ensuring convenience and trusted authentication
ChatGpt API	We use massivelly the AI to create decks and to give more insights about our errors when playing
SharedPreferences	Wraps platform-specific persistent storage for simple data, allowing us to store locally data
IconPicker	This package provides an IconPicker with supported (or custom provided) Icons. We used to increase the pickable icons
CountryFlag	A Flutter package for displaying the SVG image of a country's flag, used to add a more nice view in the language selection part (when generating a deck with AI)

Table 1: This table explain all the most relevant dependency used in the project.

3 Architecture

3.1 Overview

Our application uses Flutter, a framework designed for building native apps across different platforms using Dart. Flutter is known for its efficiency in creating visually appealing interfaces and its support for fast development cycles and can also create cross-platform application. The last point, combined with ease of use and fast-learning, is the reason why we decide to choose this framework. Also because this was an interesting possibility to learn a very flexible framework. Then in practice we implemented only the Android version of the app, in the sense that can run also on iOs or on web, but it's not well supported.

Initially, our app was designed to store data locally on users' devices, ensuring reliability even without an internet connection. Later on, we decided to integrate Firebase database to provide cloud storage and backup capabilities. This shift required us to adapt our basic data model to support Firebase, allowing for real-time data synchronization and improved accessibility across devices.

By leveraging Flutter's multi-platform support and Firebase's cloud infrastructure, we've enhanced our app's reliability and scalability. This approach ensures that users can seamlessly access their data from anywhere while maintaining a responsive user experience.

3.2 Database structure

FlashcardsAI uses Firebase to store all the data of the users. So the organization is the one provided in the image:

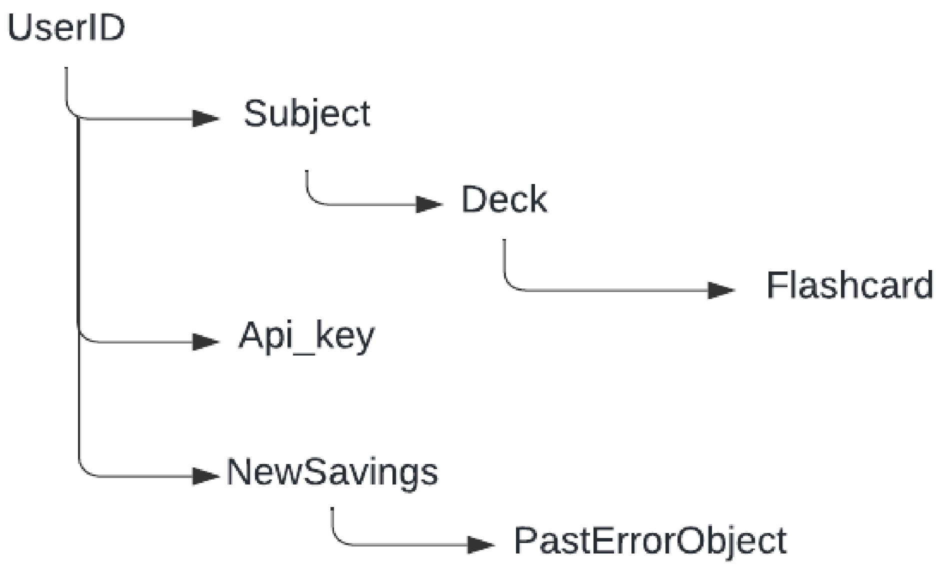


Figure 1: This is the structure of our firebase db

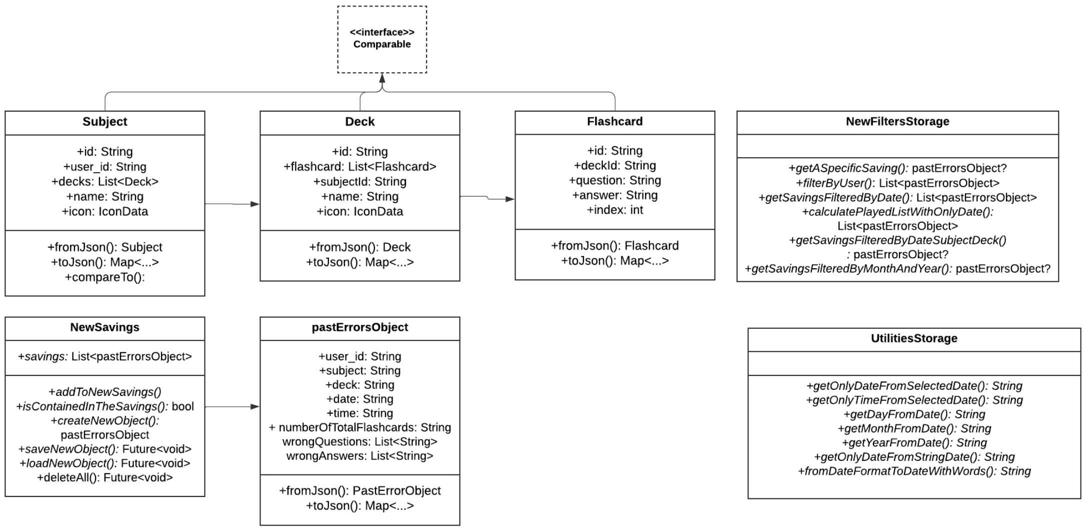
Since the structure is the same as the one created in the model, we recommend you to take a look to the app architecture documentation below to understand better the nesting storage.

3.3 App architecture

For the implementation we decide to rely, as much as possible, to the know design pattern MVVM (model, view, view-model). Basically it states that you should devide the application in 3 layer: the firs one, the model, that should store and mantain all the datas. The second one (the view) should contain all the widgets that manage the UI and then the third one (the view-model) should be a sort of bridge from the 2 part, managing the request made by the user from the UI and should update the model, compliant to the business logic. After this small and general description of the pattern let's dive deeper into the single components of each part.
(Note that:
-we represented each class with its attributes and methods. In particulars in all the diagrams I omitted all the parameters needed by each method, just to improve readability. So if needed I suggest to take a look to the code to have more details;
-for the same reason stated before we also ommitted the return values in all the widgets of the view part and view-model part)

3.3.1 Model

Here I will present the model components one by one. Figure (1) shows the main classes that we created to store and manage the data, with them attributes and methods.



<<interface>> Comparable
Subject
+id: String
+user_id: String
+decks: List<Deck>
+name: String
+icon: IconData
+fromJson(): Subject
+toJson(): Map<...>
+compareTo():
Deck
+id: String
+flashcard: List<Flashcard>
+subjectId: String
+name: String
+icon: IconData
+fromJson(): Deck
+toJson(): Map<...>
Flashcard
+id: String
+deckId: String
+question: String
+answer: String
+index: int
+fromJson(): Flashcard
+toJson(): Map<...>
NewFiltersStorage
+getASpecificSaving(): pastErrorsObject?
+filterByUser(): List<pastErrorsObject>
+getSavingsFilteredByDate(): List<pastErrorsObject>
+calculatePlayedListWithOnlyDate(): List<pastErrorsObject>
+getSavingsFilteredByDateSubjectDeck() : pastErrorsObject?
+getSavingsFilteredByMonthAndYear(): pastErrorsObject?
NewSavings
+savings: List<pastErrorsObject>
+addToNewSavings()
+isContainedInTheSavings(): bool
+createNewObject(): pastErrorsObject
+saveNewObject(): Future<void>
+loadNewObject(): Future<void>
+deleteAll(): Future<void>
pastErrorsObject
+user_id: String
+subject: String
+deck: String
+date: String
+time: String
+ numberOfTotalFlashcards: String
wrongQuestions: List<String>
wrongAnswers: List<String>
+fromJson(): PastErrorObject
+toJson(): Map<...>
UtilitiesStorage
+getOnlyDateFromSelectedDate(): String
+getOnlyTimeFromSelectedDate(): String
+getDayFromDate(): String
+getMonthFromDate(): String
+getYearFromDate(): String
+getOnlyDateFromStringDate(): String
+fromDateFormatToDateWithWords(): String

Figure 2: This is the UML diagram of the model part

3.3.1.1 Subject

The class Subject is the ancestor of our hierarchy of objects and contains: the ”userId” associated by firebase to the user after the login. This is required here to keep track of which subjects are of the current user logged. Each subjects has a peculiar name and an icon (both customized by the user) and contains all the list of decks assigned to the current subject.
The methods fromJson() and toJson() are needed to serialize and deserialize it (for istance for firebase).

3.3.1.2 Deck

The class Deck contains: a list of Flashcard, its name, the reference to the father subject and its icon. As before the methods fromJson() and toJson() are needed to serialize and deserialize.

3.3.1.3 Flashcard

The class Flashcard is the final class in the hierarchy and represent the actual flashcard used to play. So it contains 2 strings attributes :the ”question” and the ”answer” and an integer ”index” used to allow a particular ordering decide by the user. Then it has also the reference to its subject father. As before the methods fromJson() and toJson() are needed to serialize and deserialize the classes instance.

3.3.1.4 PastErrorsObject

This class is needed to store all the errors made by the user when playing a certain deck, allowing us to create a sort of history of the errors to be displayed to the user. So it contains the parameters to identify a flashcard (deck, subject, userId) and also contains the number of total cards in the deck and the list of all the questions and all the answers that the user failed during the game. This class present a different hierarchy with respect to the one described above and this is for performance purposes.

3.3.1.5 NewSavings

The newSavings class is the one used to store all the PastErrorObject and has all the methods to manage them. In particular: the ”createNewObject” is a factory method for the previous class, the ”isContainedInTheSavings” is used to scan all the ”savings” list and check not to insert duplicates.

3.3.1.6 UtilitiesStorage e NewFiltersStorage

Those 2 classes are useful filters on the ”savings” list and it’s not needed any additional comment since the names are clear and gives the exact idea of what they do.

3.3.2 ViewModel

Here I have to specify that in case of very simple interaction widget, we didn't create a custom view-model, but we handled it directly in the way. This is not to be interpreted as a violation of the MVVM pattern, instead is a way to avoid introducing more complexity than needed and also not to create a too much wide codebase that could easily become hard to manage.

Our ViewModel part uses a mix of BLoC and normal classes: the first are used in the core and more complex section of the code and the second more on the handling of simple and short updates of model and view related to the features added later on. Just to be on the same page recall that a Bloc (Business Logic Component) is a design pattern used in Flutter and Dart to manage state and logic in a predictable and testable way. It separates the business logic from the UI, allowing developers to build reusable, maintainable, and scalable applications. So, the working flow is the following:
catch of an Event or an update of the state - > execute the handler function in the bloc and produce a new state - > the UI listen to the changes and update the interface.

3.3.2.1 AuthenticationBloc

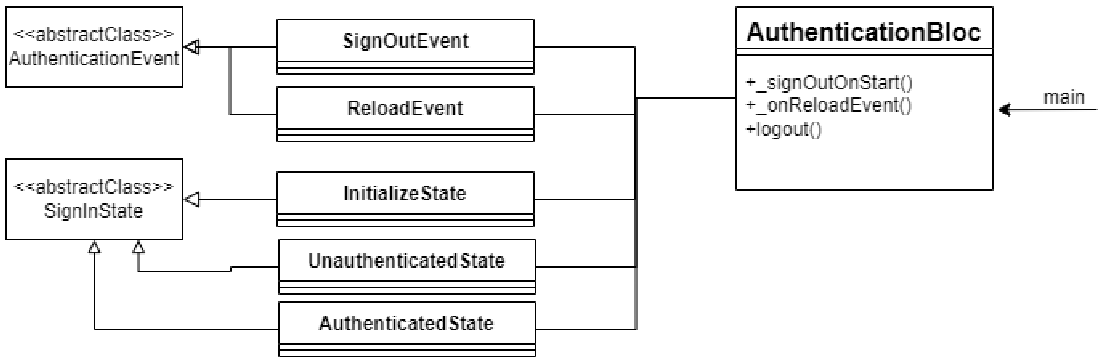


Figure 3: Diagram describing the AuthenticationBloc class

1. InitializationState
   The initial state of the AuthenticationBloc, representing the starting point of the authentication process.

2. UnauthenticatedState
   This state indicates that the user is not authenticated.

3. AuthenticatedState
   This state indicates that the user is successfully authenticated.

4. SignOutEvent
   This event is triggered when the user attempts to sign out.

5. _ReloadEvent

This event is triggered to reload the user's authentication state. It includes an optional user parameter.

6. AuthenticationBloc

The AuthenticationBloc manages the authentication state by handling events and emitting states. It includes:

- The constructor sets the initial state to InitializationState and triggers a sign-out on start.
- The _signOutOnStart method signs out the user when the app starts and listens for authentication state changes.
- The on¡SignOutEvent¿ method handles the sign-out process by signing out the user from Firebase Authentication.
- The _onReloadEvent method updates the state based on the user's authentication status, emitting UnauthenticatedState if the user is null, and AuthenticatedState if the user is present.
- The logout method triggers the sign-out process and state change.

3.3.2.2 SignInBloc

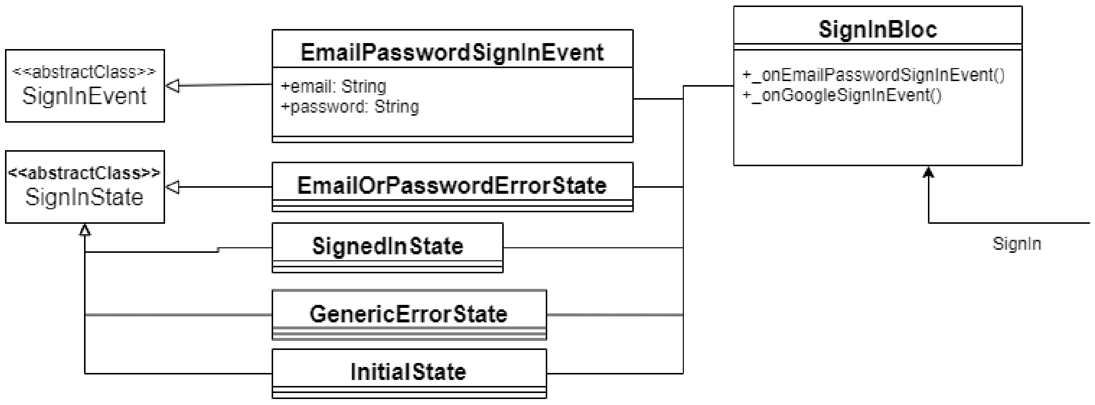


Figure 4: Diagram describing the SignInBlock class

1. EmailPasswordSignInEvent

This event is triggered when the user attempts to sign in using an email and password. It contains the email and password fields as parameters.

2. GoogleSignInEvent

This event is triggered when the user attempts to sign in using Google Sign-In. It does not require any parameters.

3. InitialState

The initial state of the SignInBloc, representing that no sign-in action has been taken yet.

4. EmailOrPasswordErrorState

This state indicates an error due to incorrect email or password during the email/password sign-in process.

5. GenericErrorState

This state represents a generic error state that includes an error message to provide more context about the error.

6. SignedInState

This state indicates that the user has successfully signed in.

7. SignInBloc

The SignInBloc manages the sign-in process by handling events and emitting states. It includes:

7.1 The constructor sets the initial state to InitialState.

7.2 The _onEmailPasswordSignInEvent method handles EmailPasswordSignInEvent by attempting to sign in with Firebase Authentication. It emits SignedInState on success, or an error state on failure.

7.3 The _onGoogleSignInEvent method handles GoogleSignInEvent by attempting to sign in with Google Sign-In. It emits SignedInState on success, or an error state on failure.

3.3.2.3 SignUpBloc

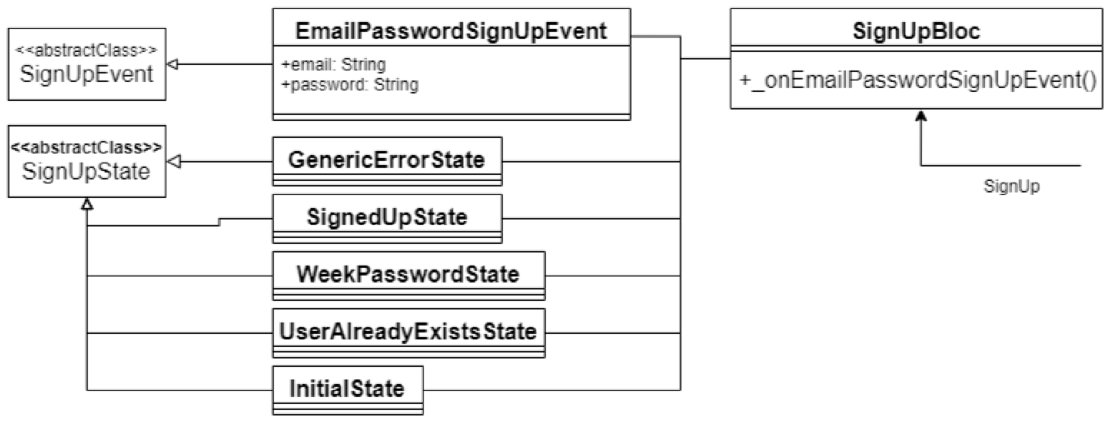


Figure 5: Diagram describing the SignUpBloc

1. EmailPasswordSignUpEvent

This event is triggered when the user attempts to sign up using an email and password. It contains the email and password fields as parameters.

2. InitialState

The initial state of the SignUpBloc, representing that no sign-up action has been taken yet.

3. UserAlreadyExistsState

   This state indicates an error due to the email already being in use during the sign-up process.

4. WeakPasswordState

   This state indicates an error due to the provided password being too weak during the sign-up process.

5. GenericErrorState

   This state represents a generic error state that handles various errors not specifically categorized, including an error message.

6. SignedUpState

   This state indicates that the user has successfully signed up.

7. SignUpBloc

   The SignUpBloc manages the sign-up process by handling events and emitting states. It includes:

   - The constructor sets the initial state to InitialState.
   - The _onEmailPasswordSignUpEvent method handles EmailPasswordSignUpEvent by attempting to sign up with Firebase Authentication. It emits SignedUpState on success, or an error state on failure.

3.3.2.4 SubjectBloc

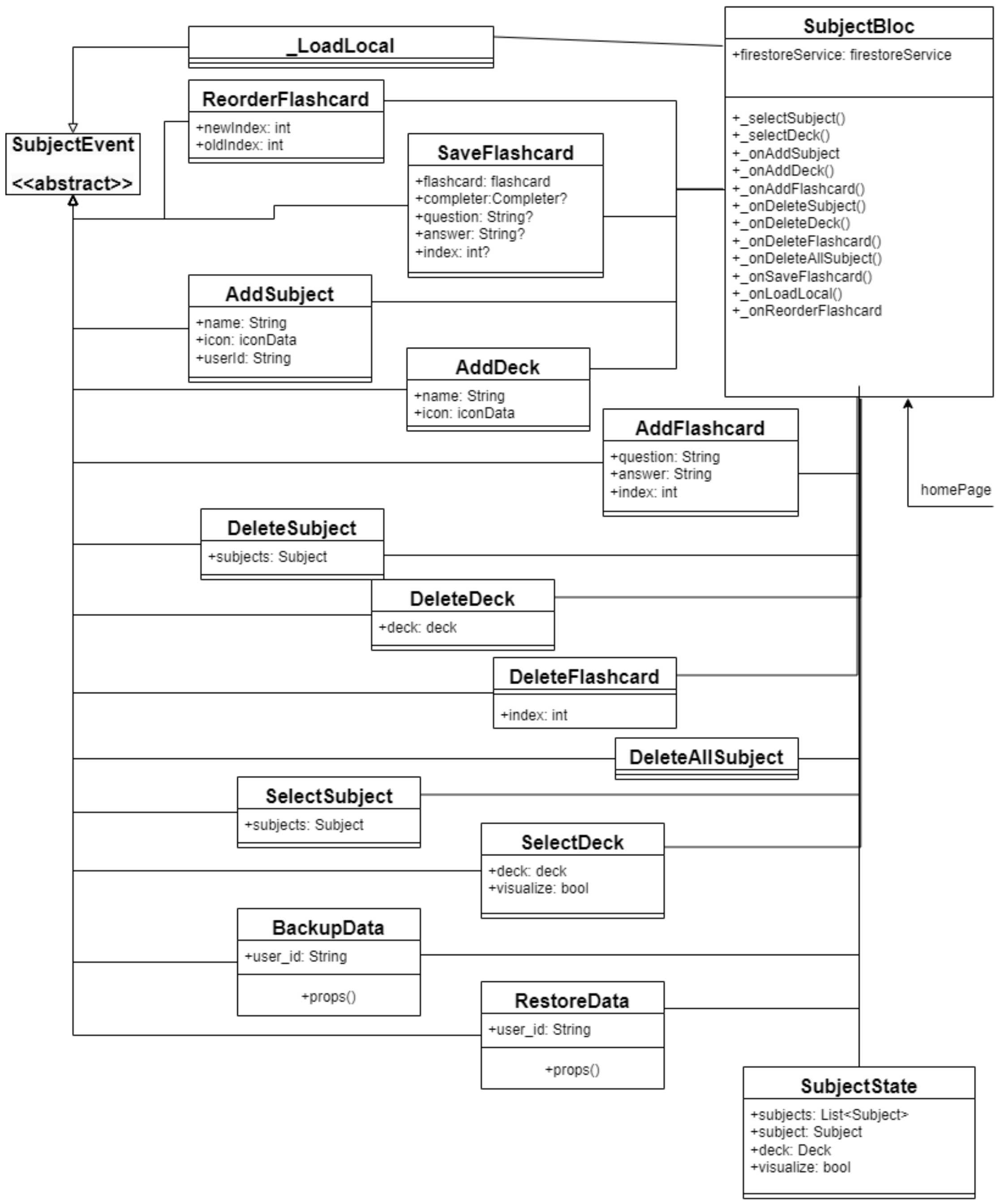


Figure 6: Diagram describing the SubjectBlock class

1. SubjectEvent

   Abstract class representing all possible events related to subjects, decks, and flashcards.

2. _LoadLocal

   Event to load subjects from the local repository.

3. SaveFlashcard

   Event to save or update a flashcard with new question and answer. Optionally completes a completer for async operations.

4. ReorderFlashcard

   Event to reorder flashcards within a deck based on new and old indices.

5. BackupData

   Event to backup all subject data to Firestore for a specific user.

6. RestoreData

   Event to restore all subject data from Firestore for a specific user.

7. AddSubject

   Event to add a new subject with a name, icon, and user ID.

8. AddDeck

   Event to add a new deck with a name and icon to the selected subject.

9. AddFlashcard

   Event to add a new flashcard with a question and answer to the selected deck.

10. DeleteFlashcard

    Event to delete a flashcard at a specific index from the selected deck.

11. DeleteDeck

    Event to delete a specified deck or the currently selected deck.

12. DeleteSubject

    Event to delete a specified subject.

13. DeleteAllSubjects

    Event to delete all subjects from the local repository and Firestore

14. SelectSubject

    Event to select a specific subject.

15. SelectDeck

    Event to select a specific deck with an option to visualize it.

16. SubjectState

    State representing the current subjects, selected subject, and selected deck. It includes:

    - subjects: List of all subjects.
    - subject: Currently selected subject.
    - deck: Currently selected deck.

- visualize: Boolean indicating if the deck should be visualized.

17. SubjectBloc

Bloc handling SubjectEvents and updating SubjectState.

- The constructor initializes with an empty subjects list and loads local subjects.
- _onAddDeck: Adds a new deck to the selected subject and updates Firestore.
- _onSelectSubject: Selects a specific subject.
- _onSelectDeck: Selects a specific deck and optionally visualizes it.
- _onDeleteDeck: Deletes a specified or currently selected deck and updates Firestore.
- _onDeleteSubject: Deletes a specified subject and updates Firestore.
- _onAddFlashCard: Adds a new flashcard to the selected deck and updates Firestore.
- _onSaveFlashcard: Saves or updates a flashcard in the selected deck and updates Firestore.
- _onLoadLocal: Loads subjects from the local repository.
- _onAddSubject: Adds a new subject to the local repository and Firestore.
- _onDeleteAllSubject: Deletes all subjects from the local repository and Firestore.
- _onReorderFlashcard: Reorders flashcards within a deck.
- _onDeleteFlashcard: Deletes a flashcard from the selected deck and updates Firestore.
- _onBackupData: Backs up all subject data to Firestore for a specific user.
- _onRestoreData: Restores all subject data from Firestore for a specific user.

3.3.2.5 PlayBloc

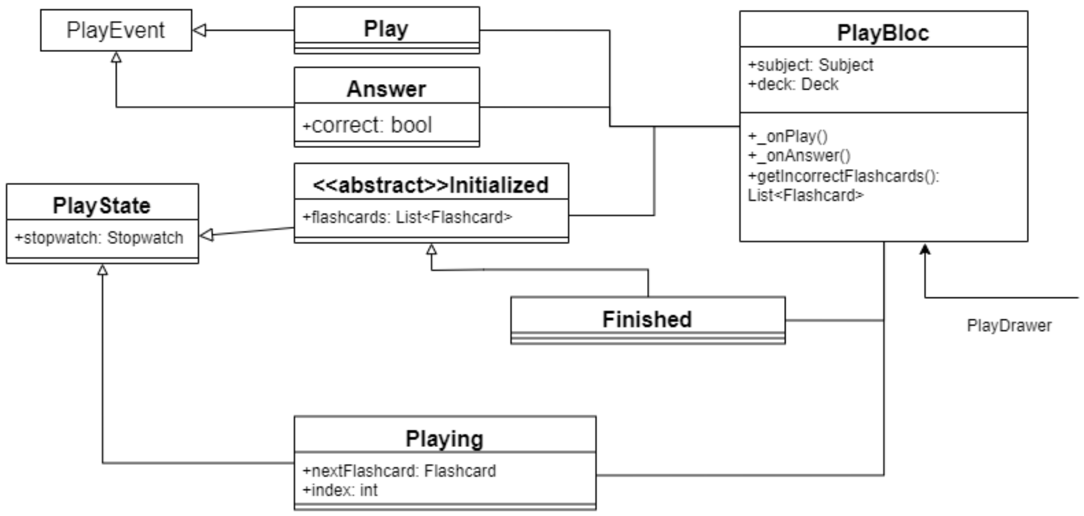


Figure 7: Diagram describing the PlayBlock class

1. PlayEvent

   Class representing all possible events related to the play session of flashcards.

2. Play

   Event to start a new play session. It initializes the flashcards and starts the stopwatch.

3. Answer

   Event to submit an answer for the current flashcard. It takes a boolean indicating whether the answer was correct.

4. PlayState

   State representing the current play session, including a stopwatch to track elapsed time.

5. Initialized

   Abstract state representing the initialized state of the play session, including a list of flashcards with their answer status.

6. Playing

   State representing an ongoing play session. It includes:

   - nextFlashcard: The next flashcard to be shown.
   - index: The current index of the flashcard being shown.
   - flashcards: List of flashcards with their answer status.
   - stopwatch: Stopwatch tracking the elapsed time.

7. Finished

   State representing a finished play session. It includes:

   - flashcards: List of flashcards with their answer status.
   - stopwatch: Stopwatch tracking the total time of the session.

8. PlayBloc

   Bloc handling PlayEvents and updating PlayState. It includes:

   - The constructor initializes with an empty play state and assigns event handlers.
   - _onPlay: Handles the Play event, shuffling flashcards, and starting the stopwatch.
   - _onAnswer: Handles the Answer event, updating the answer status of the current flashcard, and transitioning to the next flashcard or finishing the session.
   - getIncorrectFlashcards: Returns a list of flashcards that were answered incorrectly in the finished session.

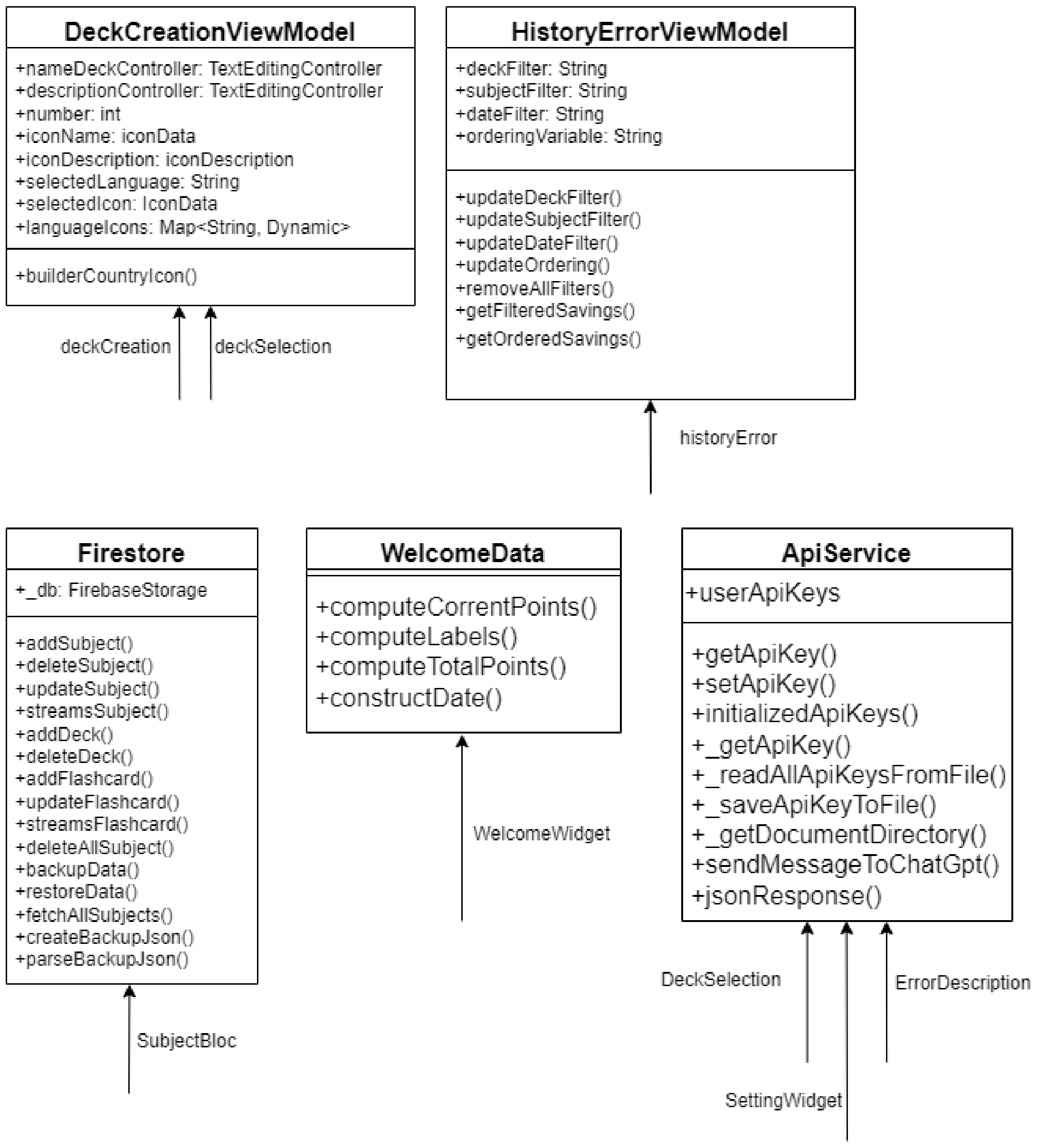


Figure 8: Diagram describing the other classes uses as VM

3.3.2.6 ApiService

1. ApiService

   This is a singleton class managing API keys and interactions with the OpenAI API, ensures a single instance of the service. In particular it stores the key of each user and provide the methods to send and receive Json request to ChatGpt

2. getApiKey

   Method to retrieve the API key for a specific user. Returns a default key if none is found.

3. setApiKey

Method to set and save a new API key for a specific user. Updates the in-memory storage and persists the key to a file.

4. initializeApiKeys

   Method to initialize user API keys from files on app startup. Reads keys from the documents directory.

5. _getApiKeyFile

   Private method to get the file object for storing a user's API key. Generates the file path based on the user's ID.

6. _readAllApiKeysFromFiles

   Private method to read all API keys from files in the documents directory. Loads keys into a map.

7. _saveApiKeyToFile

   Private method to save a user's API key to a file. Writes the key to the appropriate file in the documents directory.

8. _getDocumentsDirectory

   Private method to get the path to the application's documents directory. Used for file storage.

9. sendMessageToChatGPT

   Method to send a message to ChatGPT using the OpenAI API. Constructs the HTTP request, handles the response, and returns the message content.

3.3.2.7 FirestoreService

1. FirestoreService

   This is a classes that act as a single controller on the calls to firebase and manage the updates and downloads on the data

2. FirestoreService

   A class managing interactions with Firebase Firestore, including CRUD operations for subjects, decks, and flashcards.

3. addSubject

   Method to add a new subject to Firestore and update it with the generated ID.

4. updateSubject

   Method to update an existing subject's data in Firestore.

5. deleteSubject

   Method to delete a subject from Firestore by its ID.

6. streamSubjects

   Method to stream subjects from Firestore, returning a list of subjects.

7. addDeck

   Method to add a new deck to Firestore and update it with the generated ID.

8. deleteDeck

   Method to delete a deck from Firestore given the subject ID and deck ID.

9. addFlashcard

   Method to add a new flashcard to Firestore and update it with the generated ID.

10. updateFlashcard

    Method to update an existing flashcard's data in Firestore given the deck ID.

11. deleteAllSubjects

    Method to delete all subjects from Firestore.

12. deleteFlashcard

    Method to delete a flashcard from Firestore given the deck ID and flashcard ID.

13. backupData

    Method to back up data to Firestore for a specific user by storing JSON data.

14. restoreData

    Method to restore data from Firestore for a specific user by retrieving JSON data.

15. _fetchAllSubjects

    Private method to fetch all subjects, decks, and flashcards from Firestore.

16. createBackupJson

    Method to create a JSON string representing a backup from a list of subjects.

17. parseBackupJson

    Method to parse a JSON string into a list of subjects.

3.3.2.8 WelcomeData

1. WelcomeData

   This is a utility class providing methods to compute data for the welcome screen, such as correct points, labels, and total points based on user activity for a given month.

2. computeCorrectPoints

   Computes the number of correct answers per day for a given month by determining the number of days in the month, iterating through each day, retrieving filtered data, calculating correct answers, and returning an array of correct points for each day.

3. computeLabels

   Generates labels for each day of the month by determining the number of days in the month and creating a list of day numbers as strings.

4. computeTotalPoints

   Calculates the total number of flashcards played per day for a given month by determining the number of days in the month, iterating through each day, retrieving filtered data, summing total flashcards played, and returning an array of total points for each day.

5. constructDate

   Constructs a date string for a specific day of the month by parsing the given date string to a DateTime object, creating a new DateTime object for the specified day, and formatting and returning the new DateTime as 'yyyy-MM-dd'.

6. retrieveNumberOfDaysInAMonth

   Retrieves the number of days in a specific month and year by creating a DateTime object for the first day of the next month, subtracting one day to get the last day of the current month, and returning the day number of the last day.

3.3.2.9 DeckCreationViewModel

1. DeckCreationViewModel

2. DeckCreationViewModel

   A view model class for managing the state and behavior of the deck creation screen. It initializes controllers, icons, selected language, and a map of language icons.

3. DeckCreationViewModel Constructor

   Initializes the view model with default values for text controllers, number, icons, selected language, and selected icon. It also initializes a map of language icons using the builderCountryIcon method.

4. builderCountryIcon

   Creates a CountryFlag widget for the given language code, setting the height, width, and border radius. This method is used to populate the languageIcons map with flag icons for different languages.

3.3.2.10 HistoryErrorViewModel

1. HistoryErrorViewModel

   A view model class extending ChangeNotifier that manages filters and sorting for historical error data. It includes properties for deck, subject, date filters, and an ordering variable.

2. updateDeckFilter

   Updates the deck filter and notifies listeners of changes.

3. updateSubjectFilter

   Updates the subject filter and notifies listeners of changes.

4. updateDateFilter

   Updates the date filter and notifies listeners of changes.

5. updateOrdering

   Updates the ordering variable based on the provided ordering string. It validates the input against the OrderingEnum values and notifies listeners if a match is found.

6. removeAllFilters

   Clears all filters (deck, subject, and date) and notifies listeners of changes.

7. getFilteredSavings

   Returns a filtered list of pastErrorsObject based on the current filters. It iterates through allSavings, applying deck, subject, and date filters.

8. getOrderedSavings

   Returns an ordered list of pastErrorsObject based on the current ordering variable. It sorts the list according to the specified ordering criteria in the orderingVariable.

9. OrderingEnum

   Enum defining the possible ordering options: subjectNameAZ, subjectNameZA, deckNameAZ, deckNameZA, dateIncrease, and dateDecrease.

3.3.3 View

Here I will present the model components one by one. In particular here we will adopt this approach: in order to describe better the architecture of the view we decided to divide it in sections. Each section will start with an image describing the classes among that section and will have a list of paragraphs to describe in details what each widget does.
Note that:
In the following diagrams the syntax `<<StatelessWidget>>` means extend `StatelessWidget` and the same for the syntax `<<State<MyApp>>>` means "extend `State<MyApp>` ". We followed this convention that is not totally compliant with the standard UML policy just not to introduce too much useseless arrows towards the StatefulWidgets and StatelessWidgets.

So without losing other time let's start !

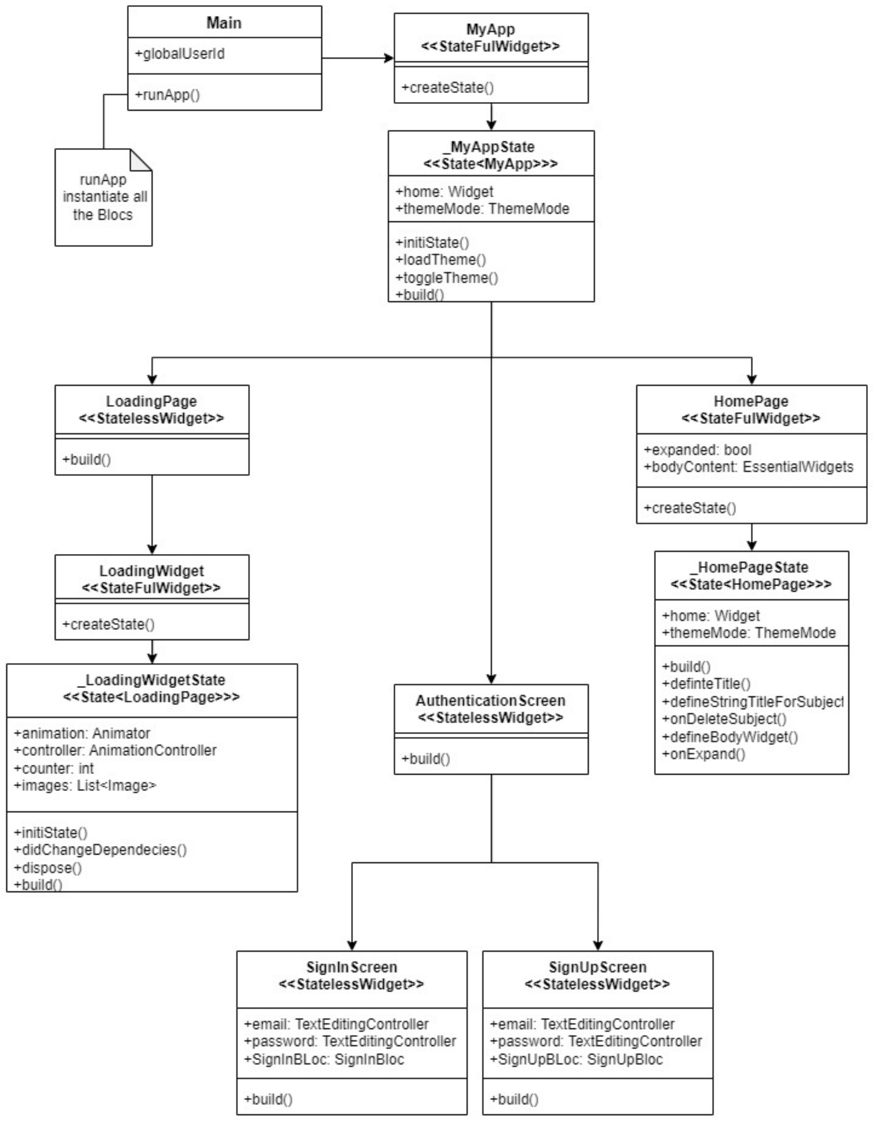


Figure 9: First part (MyApp, SignIn, SignUp)

This figure present the process of starting the app and loading the first screens used for the authentication. In particular now let's break down all the components and their roles:

3.3.3.1 Main

It contains all the initialization of the basic services such as firebase for the login, the apiService that is used to store the api keys associated to each user, the globalUserId that is the Id provided by firebase that is associated with the account that will be used to authenticate in the app. This value will be associated to each subject and will be used as a key for the filtering part of subjects/decks/flashcards. Then in the runApp function there is also all the initialization of the "MultiBlocProvider" that contains all the other blocs used. In particular we decide to use this component since "improves the readability and eliminates the need to nest multiple BlocProviders", as stated by the flutter_bloc_package_documentation. Finally, the multibloc component has also a field "child" that allow us to call the main start widget of the entire app that is the following MyApp Widget. Note that all those operations performed in this section are not directly visible from the diagram because are all executed in the runApp function, that hides the complexity of those instructions, enhancing the readability of this schema.

3.3.3.2 MyApp

This is, as stated before, the main basic widget of the app. As we can see it extends "StatefulWidget" and has 2 important parameters:
-"home", that keep tracks of which widget among the 3 specified in this part ("LoadingPage", "AuthenticationScreen" and "Homepage") is the current displayed widget.
-"themeMode", that is used to set the dark-mode or bright.mode using the method specified after ("loadTheme" and "toggleTheme".
Finally the method build set some parameters and load the screen selected. As default the one selected at the beginning is LoadingPage, that is immediately swapped to the AuthenticationScreen as soon as the AuthenticationBloc is instantiated.

3.3.3.3 LoadingPage

This widget, as I was saying in the prevoius point, is a sort of splashScreen used to make the user wait till the initialization of the blocs, and in particular, the one of the authenticationBloc is concluded. So it simply provide a basic widget where is nested also the "LoadingWidget".

3.3.3.4 LoadingWidget

This widget has to make a sort of animation to make the screen nicer to look at. So it has a loop of 5 images of a humanized memo that is walking.

3.3.3.5 AuthenticationScreen

This widget is triggered after the initialization of the "AthenticationBloc" and it provides a simple welcome screen with the possibility to choose between 2 types of accessing mechanism: -sign in; -signup.

3.3.3.6 SignInScreen

After hitting the SignIn button in the previous screen you directly access to thsi widget where you can access to your user by using a simple Email-Password mechanism provided by firebase or using the

simpler login mechanism based on SignIn with Google. In the first case we simply wait for the user to insert the 2 data requested (email and password) and we send them to Google using the function "_onEmailPasswordSignInEvent" described in the "signInBloc" and waiting for the result.
In the second case we do the same thing but using the "_onGoogleSignInEvent" function in the same bloc used before. In case of wrong data we display a String "generic error" in the page.

3.3.3.7 SignUpScreen

Intuitively, this screen has the same way of working of the SignIn one. So, using the "SignUpBloc" we are able to manage the validation of email-password using the "firebase authenticator service " and also bad states such as "weak password" or "email already in use" or "generic error", the ascreen display the errors.

3.3.3.8 Homepage

This is the most important widget in out hierarchy. In particular this stateful widget has 2 attributes:
-"expanded" will control the button located on the top-left part of the screen thanks to the "onExpand()" method.
-"bodyContent" will controll what will be displayed in the central part of our screen.

3.3.3.9 HomepageState

Here we have a buil method that paint the drawer on the left called "SubjectSelection", the title of the central widget (located in the top part of the screen) and the actual content of the central screen. As I was saying in the Homepage widget, this part will be controlled by the value of the bodyContent variable. In particular this values is set in the "defineBodyWidget" method and assign a new widget to this parameter according to the current selected widget from the drawer. Then there are some other methods like "defineStringTitleForSubject" and "defineTitle" that will manage the title based on the bodyContent selected and also based on the available space (in the "defineStringTitleForSubject" method in fact we calculate how many words of the title can fit in the title space).

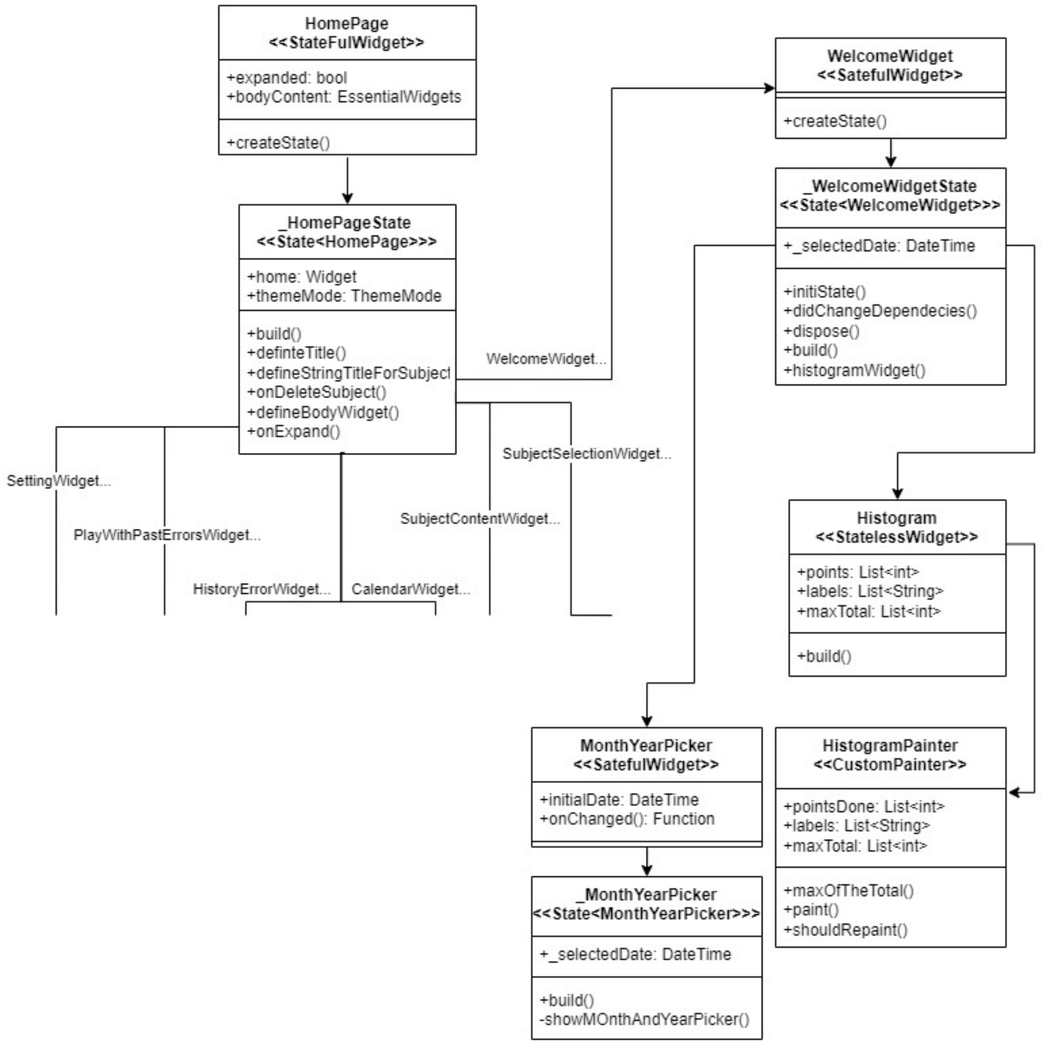


Figure 10: Second part(WelcomeWidget,Histogram,MonthYearPicker)

This figure present the basic schema of the welcomeWidget that manages all the introduction part to the app and also the histogram painting. Let's see in details:

3.3.3.10 WelcomeWidget, WelcomeWidgetState

The first is a simple widget that create the basic state for the screen. In the second instead there are all the data that are needed to create the Welcome screen. So basically there are some Text fields to print the initial info about the app and then, under them, there is an Histogram. In particular, this widget enables the user to select a date through the "MonthYearPicker" and then this date is stored here. The date is used to filter the errors through the "WelcomeViewModel" class that are then passed as parameter to the histogram.

3.3.3.11 Histogram

This widget take 3 parameters from the "_WelcomeWidgetState" already filtered by the "WelcomeViewModel" class and pass them to the "HistogramPainter" in the build method.

3.3.3.12 HistogramPainter

The histogramPainter simply paint the histogram. In order to do that it computes the maximum value among the total points with the "maxOfTheTotal()" method and then it paints: the axis, the labels and all the data. In particular it uses the "RRect.fromRectAndRadius" that allows to create a rectangle of a parametric height and a parametric rounded corner.. The method "shouldRepaint()" is called when a new instance of the class is provided, to check if the new instance actually represents different information

3.3.3.13 MonthYearPicker

This widget allows to take the month and the year to build the _selectedDate needed by the Histogram

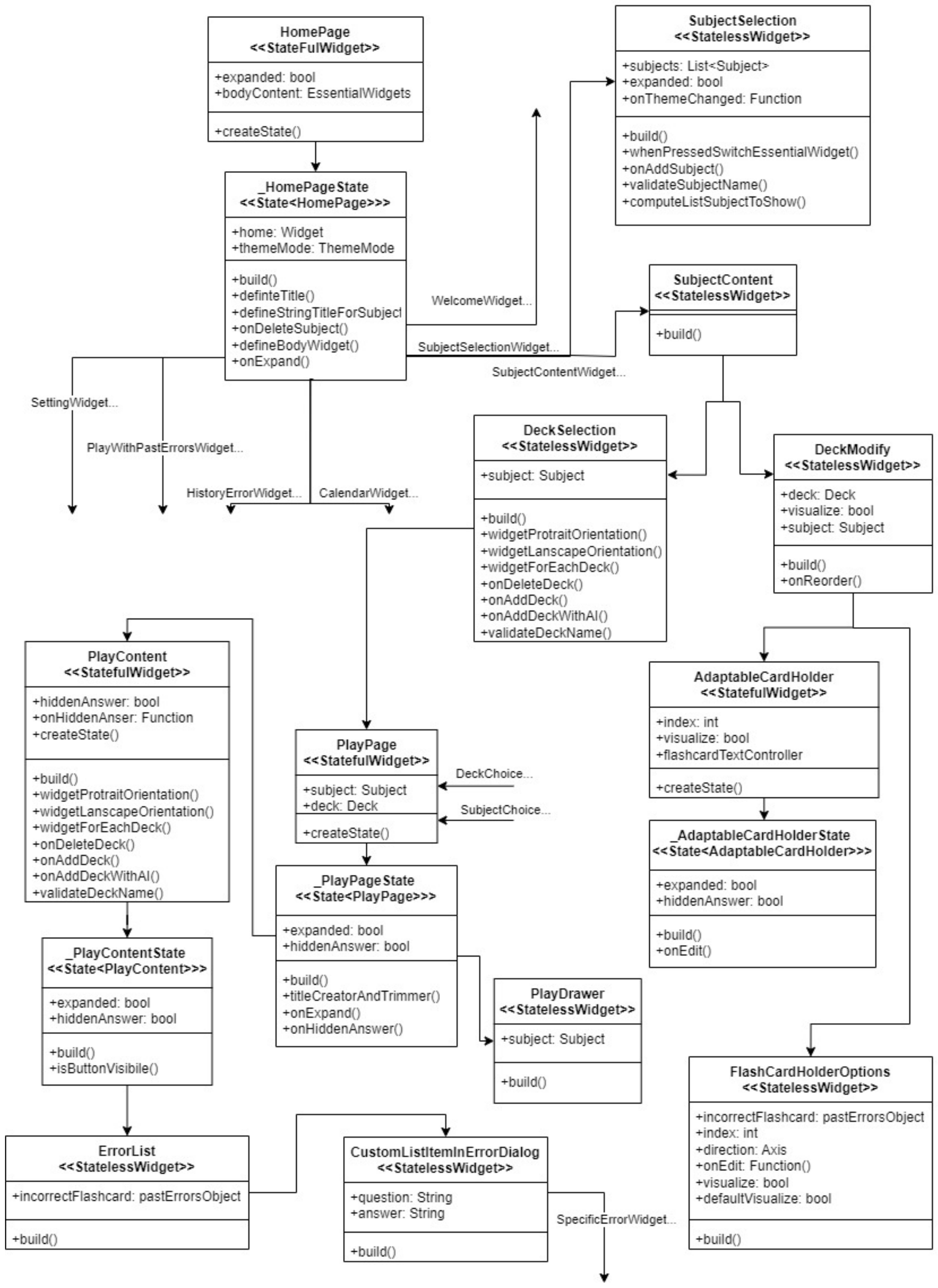


Figure 11: Third part (SubjectSelection, SubjectContent, DeckSelection, DeckModify, AdaptableCardHolder, FlashCardHolderOptions, PlayPage, PlayContent, PlayDrawer, ErrorList, CustomListItemInErrorDialog)

The figure present all the long part of the subject selection and playing. This schema is full of

things but the follow, even if a bit intricated, is logical straight forward. You select a subject, then a deck, then you play, then you can see the results and also the errors. Then there is also the possibility to manage the deck or the subject to add/modify/remove flashcards/decks.
Let's see more in details:

3.3.3.14 SubjectSelection

This is a widget that will be used a drawer on the left

3.3.3.15 SubjectContent

This widget displays the main content for a subject. It uses BlocBuilder to listen to state changes from SubjectBloc and update its display accordingly. If no subject is selected, it shows a message prompting the user to select a subject. If a subject is selected but no deck is available, it displays the DeckSelection widget. If both a subject and deck are selected, it displays the DeckModify widget.

3.3.3.16 DeckSelection

This widget provides an interface for selecting and managing decks within a subject. It displays the decks in a list format, which adapts to portrait and landscape orientations. In portrait mode, each deck is displayed in a column with its details and actions using the "widgetPortraitOrientation()" method. In landscape mode, the actions are aligned horizontally next to the deck details, using the "widgetLandscapeOrientation()" method. Then users here can also add new decks, either manually or using AI, and perform actions such as play, view, edit, and delete on existing decks. This is possible using the methods "onaddDeck()", "onaddDeckWithAI()": the first displays a dialog for creating a new deck. Users can enter the deck name and select an icon. The deck name is validated for uniqueness. If valid, an event is dispatched to the SubjectBloc to add the new deck.
The second instead displays a dialog for users to specify deck details and communicates with an AI service to generate the deck content. The generated deck and flashcards are then added to the SubjectBloc. Finally the method "validateDeckName()" do a check on the deck name preventing the insertion of an empty name or a duplicate one (w.r.t the already stored decks).

3.3.3.17 DeckModify

This widget manages the modification of an existing deck. It takes a deck, a boolean flag visualize, and a subject as parameters. It provides the functionality to visualize or edit the deck associated with the given subject. The visualize flag determines if the deck is displayed for viewing or editing.

3.3.3.18 AdaptableCardHolder

This widget holds the structure for displaying and editing flashcards. It adapts its layout based on screen width, displaying options and cards either in a row or column. It initializes with flashcard-TextEditingController, index, and visualize to manage flashcard data and visualization state.

3.3.3.19 _AdaptableCardHolderState

Manages the state for AdaptableCardHolder. It toggles the visualization state of the flashcards and uses LayoutBuilder to adapt the layout based on screen size, switching between vertical and horizontal alignments. The onEdit method toggles the visualize state.

3.3.3.20 FlexCardHolderOptions

A stateless widget displaying options for each flashcard. It includes the card index, a drag handle, and buttons for toggling visualization, editing, and deleting. It adjusts its layout direction (horizontal/vertical) based on the provided direction parameter.

3.3.3.21 PlayPage

This widget represents the main interface for playing through a deck of flashcards. It initializes the PlayBloc with the given subject and deck, and starts the play session. The page layout is managed by AdaptablePage, which includes an expandable drawer (PlayDrawer), the main content (PlayContent), and a dynamic title generated by titleCreatorAndTrimmer. Users can toggle the answer visibility and expand or collapse the drawer.

3.3.3.22 _PlayPageState

This class manages the state of PlayPage. It handles toggling of the expanded state for the drawer and visibility of the flashcard answers. It includes methods onExpand to toggle the drawer and onHiddenAnswer to toggle answer visibility. The titleCreatorAndTrimmer method creates a shortened title for display.

3.3.3.23 PlayDrawer

This widget displays the drawer content for the PlayPage. It uses BlocBuilder to listen to state changes from PlayBloc and update the drawer's content accordingly. It shows the status of each flashcard (correct, incorrect, or unanswered) using colored icons and includes an AdaptableStopwatch to track the session duration.

3.3.3.24 PlayContent

This widget manages the main content for playing flashcards. It uses BlocBuilder to listen to the PlayBloc state and update the UI accordingly. If in the Playing state, it shows the current flashcard with the option to reveal the answer. If the answer is revealed, it provides buttons for marking the answer as correct or wrong. If in the Finished state, it displays the results of the session, including the number of correct and wrong answers, and provides options to view errors or play again.

3.3.3.25 _PlayContentState

This class manages the state of the PlayContent widget. It updates the UI based on the current state of the PlayBloc. It handles button presses to reveal the answer, mark the flashcard as correct or wrong, and reset the answer visibility. It also manages displaying the session results and showing a dialog with the incorrect flashcards.

3.3.3.26 ErrorList

This widget displays a list of incorrect flashcards within a SingleChildScrollView. It takes a pastErrorsObject as input, which contains lists of wrong questions and answers. It constructs a column layout with a fixed container size of 600x600 pixels, ensuring the content fits within this space. The ListView.builder renders each incorrect flashcard pair using CustomListItemInErrorDialog, presenting the question and its corresponding wrong answer.

3.3.3.27 CustomListItemInErrorDialog

This widget presents a single item in the error dialog list. It displays the question and the corresponding incorrect answer received from the ErrorsList. The layout is padded to provide spacing between items in the list.

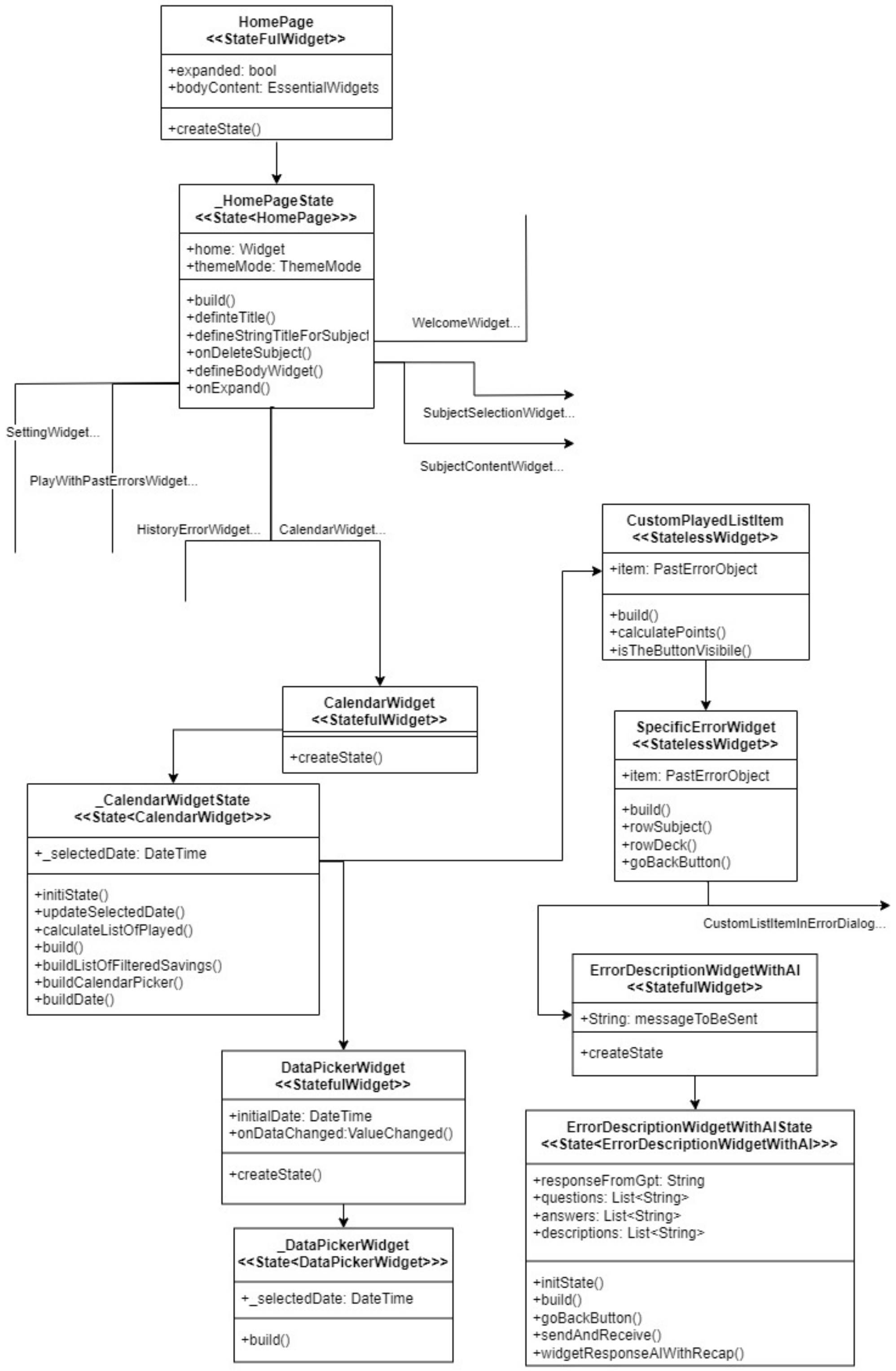


Figure 12: Fourth part (CalendarWidget, DataPickerWidget, CustomPlayedListItem, SpecificError-Widget, ErrorDescriptionWwidgetWithAI,)

The figure present the calendarWidget flow. Basically you can choose a date and see the errors made on that date. Then you can also ask the ai to generate more details on the answer to dive deeper

into the learning.
Let's see in details

3.3.3.28 CalendarWidget

This stateful widget manages the calendar-based interface for displaying filtered savings. It initializes with the current date, updating _selectedDate via _updateSelectedDate. calculateListOfPlayed filters data based on the selected date and user ID. The widget conditionally builds its UI based on the device orientation, utilizing methods buildCalendarPicker, buildDate, and buildListOfFilteredSavings to structure the display elements dynamically.

3.3.3.29 DatePickerWidget

This stateful widget provides a calendar date picker interface. It accepts initialDate and onDate-Changed as parameters, initializing _selectedDate with the provided initial date. The widget contains a CalendarDatePicker to handle date selection, updating both its internal state and the parent state through the onDateChanged callback whenever a new date is picked.

3.3.3.30 CustomPlayedListItem

This stateless widget displays past error information for a flashcard session. It takes a pastErrorsObject as an input and calculates the user's score by subtracting the number of errors from the total number of flashcards using the calculatePoints method. The isTheButtonVisible method determines if the "See errors" button should be shown. If visible, the button navigates to SpecificErrorWidget to display detailed errors.

3.3.3.31 SpecificErrorWidget

A stateless widget that displays detailed error information for a specific flashcard session. It accepts a pastErrorsObject as a parameter. The widget includes rowSubject and rowDeck methods to display the subject and deck information. It lists all the wrong answers using CustomListItemInErrorDialog. A button allows users to get more details from an AI using ErrorDescriptionWidgetWithAI, constructed by constructionOfTheMessageForDetails. The goBackButton provides navigation back to the previous screen.

3.3.3.32 CustomListItemInErrorDialog

A custom widget (not provided in the original code) likely used within SpecificErrorWidget to display individual wrong questions and answers from the pastErrorsObject. It takes a question and answer as inputs to display them in the error details list.

3.3.3.33 ErrorDescriptionWidgetWithAI

A custom widget (not provided in the original code) that likely displays AI-generated detailed explanations for the errors in a flashcard session. It takes a message generated by constructionOfTheMessageForDetails and displays the AI response for further insights into the errors.

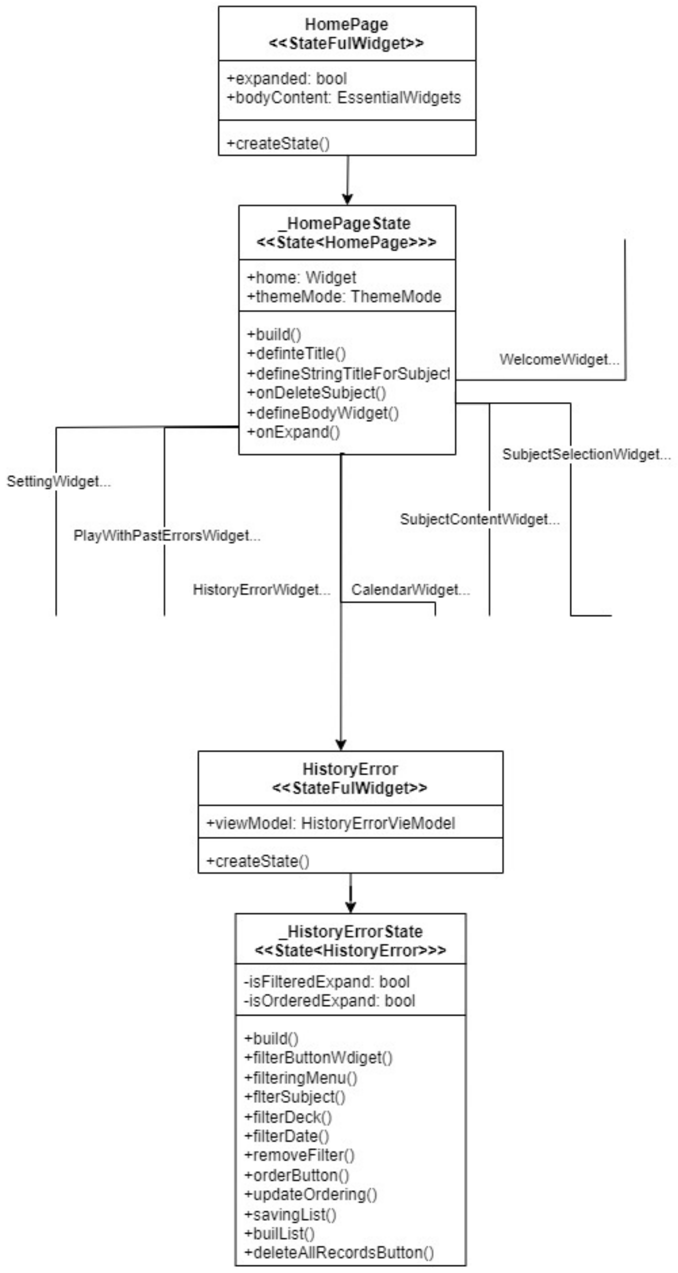


Figure 13: Fifth part (HistoryError)

3.3.3.34 HistoryError

The figure present the historyWidget. It's used to access to all the records of errors made. It offers also some filtering parameters (subject, deck, date) and also some ordering options. Let's see:
The "HistoryError" widget is a stateful widget that displays and manages historical error records using a "HistoryErrorViewModel". It maintains the state for filter and order expansion. The build method constructs the UI based on the current theme and orientation, incorporating a filter button, order button, list of filtered savings, and a delete button.
The "FilterButtonWidget" renders a button that toggles the filter menu's visibility. When expanded, the filter menu displays dropdowns for selecting a subject and deck, and a date picker for date filtering. These inputs update the "viewModel" with the chosen filters, which are applied to the savings data.
The "orderButton" toggles the visibility of an order menu. When expanded, the order menu provides sorting options. The selected ordering is applied to the filtered data through the viewModel.
The "savingList" widget retrieves filtered and ordered savings records from the viewModel and displays them in a scrollable list. This list is built using the "buildList" method, which creates a ListView of error records, with each record rendered by the "CustomErrorListItem" widget.
The "deleteAllRecordsButton" renders a button that, when pressed, shows a confirmation dialog. If confirmed, it deletes all records by calling NewSavings.deleteAll and updates the state to reflect the changes.
The overall layout adapts to theme and orientation changes, ensuring a consistent user experience across different device settings.

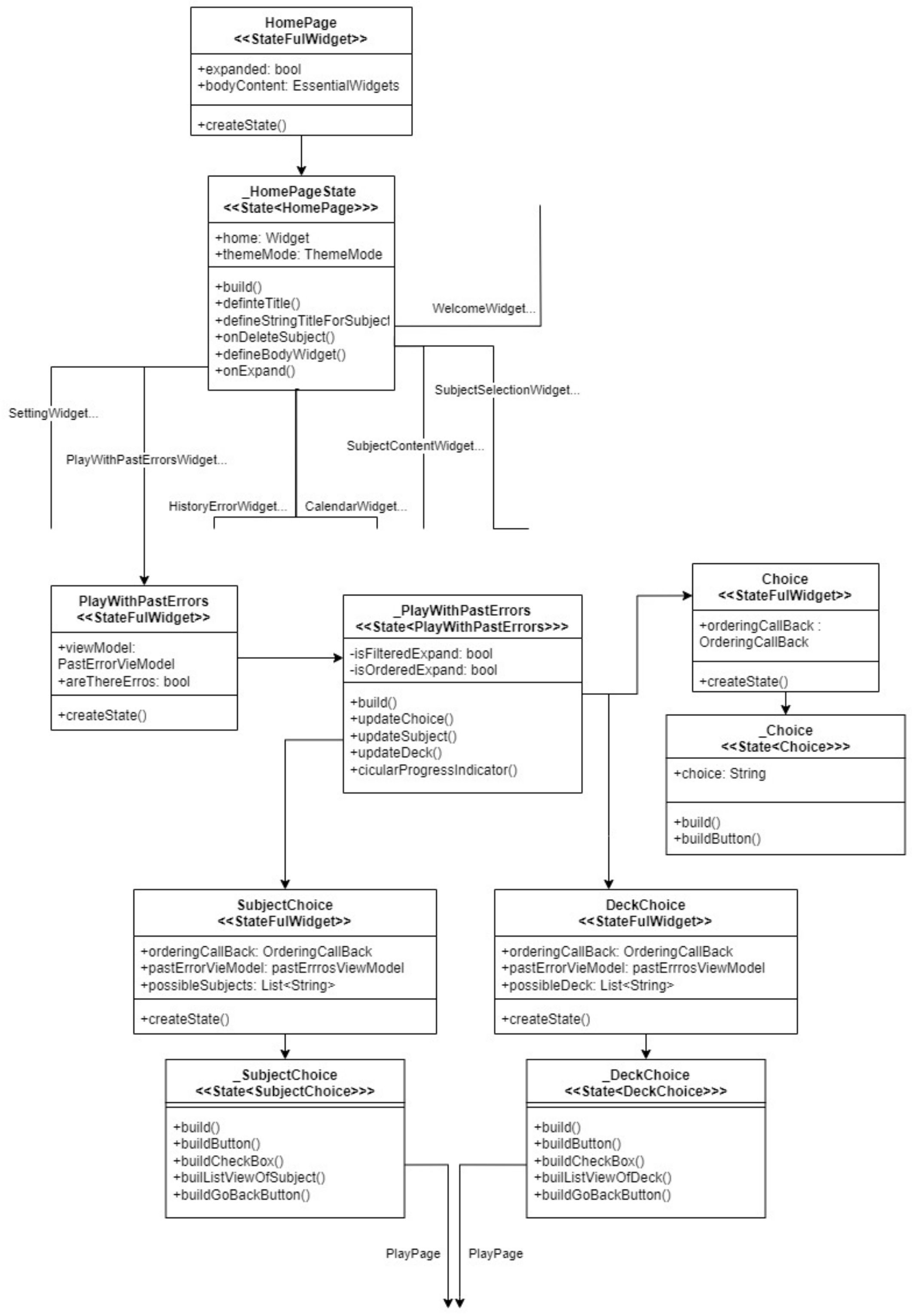


Figure 14: Sixth part (PlayWithPastErrors, Choice, SubjectChoice, DeckChoice)

This figure present the flow of the PlayWithErrors functionality. Here you can decide if you want to revise a deck or a subject and which one. If you select the check box then you will paly with all the wrong flashcards.

3.3.3.35 PlayWithPastErrors

This widget manages the state and behavior for reviewing past errors. It interacts with a PastErrorsViewModel to determine the current phase of play and displays corresponding UI elements. If in the "choice" or "play" phase, it shows a Choice widget and a message if there are no errors. Otherwise, it displays a loading indicator or widgets for selecting subjects or decks, depending on the phasePlayChoice value.

3.3.3.36 _PlayWithPastErrors

This state class handles the logic for updating UI elements based on user interactions and the current phase of the PastErrorsViewModel. It contains methods for updating choices (updateChoice), subjects (updateSubject), and decks (updateDeck). These methods modify the view model's state and trigger UI updates. It also includes a method to display a circular progress indicator during loading.

3.3.3.37 Choice

This widget allows users to choose between reviewing subjects or decks. It receives a callback function (orderingCallback) to handle the user's selection. It visually highlights the selected choice and provides two buttons for "Review a subject" and "Review a deck".

3.3.3.38 _Choice

This state class maintains the current choice selection and updates the UI accordingly. It uses the orderingCallback to notify the parent widget of the user's choice. The buildButton method creates buttons with dynamic styles based on the selection state.

3.3.3.39 DeckChoice

This widget displays available decks for review and interacts with PastErrorsViewModel to fetch these decks. It also provides an option to replay all wrong flashcards. Key attributes include orderingCallback for handling selection, pastErrorsViewModel for accessing data, and possibleDecks to store deck options. Methods include buildCheckBox for rendering a checkbox to replay all flashcards, buildListViewOfDeck to display deck options, and buildGoBackButton for a navigation button to return to the previous screen.

3.3.3.40 _DeckChoice

This state class initializes the list of possible decks from the view model and updates the UI based on user interactions. Key methods include buildButton, which creates buttons for each deck and handles navigation to PlayPage on selection; buildCheckBox, which toggles the option to replay all flashcards; and buildListViewOfDeck, which renders a list of selectable decks.

3.3.3.41 SubjectChoice

This widget displays available subjects for review, interacting with PastErrorsViewModel to fetch these subjects. It provides an option to replay all wrong flashcards. Key attributes include orderingCallback for handling subject selection, pastErrorsViewModel for accessing subject data, and possibleSubjects to store subject options. Methods include buildCheckBox for rendering a checkbox to replay all flashcards, buildListViewOfSubjects to display subject options, and buildGoBackButton for a navigation button to return to the previous screen.

3.3.3.42 _SubjectChoice

This state class initializes the list of possible subjects from the view model and updates the UI based on user interactions. Key methods include buildButton, which creates buttons for each subject and handles navigation to PlayPage on selection; buildCheckBox, which toggles the option to replay all flashcards; and buildListViewOfSubjects, which renders a list of selectable subjects.

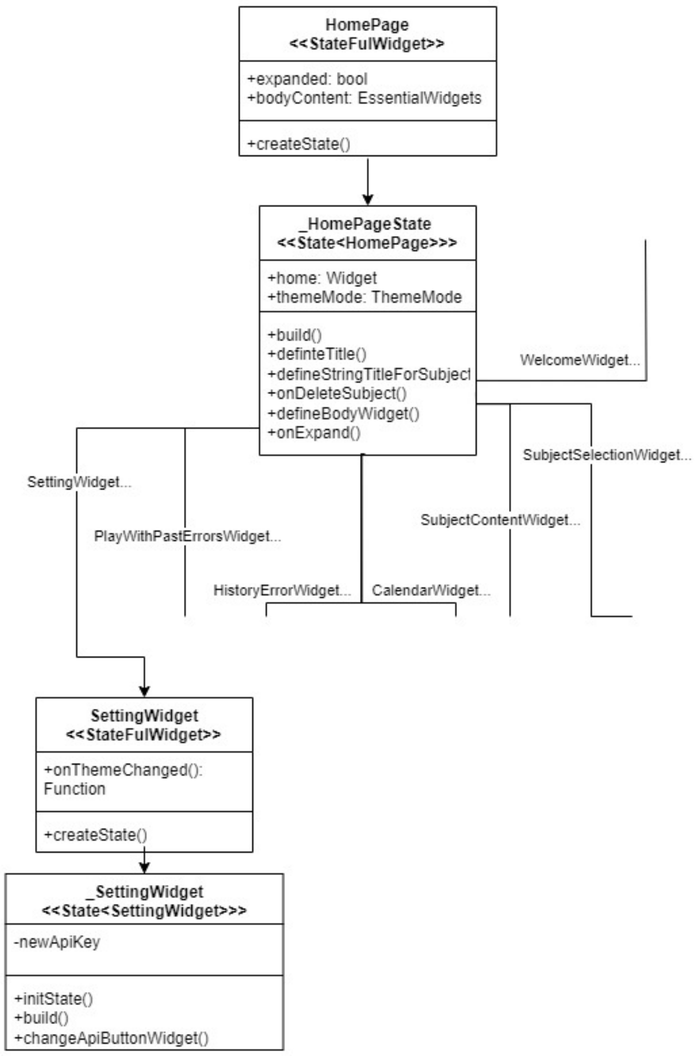


Figure 15: Seventh part (SettingWidget)

The figure present a settingWidget that allows you to see and change the Api_key and also to enable or disable the dark mode.

3.3.3.43 SettingsWidget

This widget just create the states and a callback function (onThemeChanged) to notify the parent widget of theme changes.

_SettingsWidgetState
This state class manages the UI and logic for SettingsWidget. It initializes with the current API key, provides a switch to toggle dark mode, and includes functionality to display and change the API key. The important methods are changeApiButtonWidget which shows a dialog for updating the API key, and it interacts with ApiService to get and set the API key using the functions provided in the ApiService.dart file. In particular they uses the path_provider dependency to store the couple <user,key> in local so that each user can manage its own key.

4 User Interface

4.1 UI Prototyping

We must admit that we understood the importance of a good initial prototyping phase hardly and quite lately. In fact we started the development having clear in mind the most important features that our app should have. But we completely overlooked through some important decisions such as UI. So, for this reasons, we started to having some issue on the placement of new object in the middle of the development stage and this started to be painful... In the end we manage to create, anyway, still a good UI that is able to help a lot the user visually, guiding it and supporting him/her through the use of colors, line, styles, shapes and icons.

4.2 Goals of the UI

The main aim of the UI design should be to provide a familiar and intuitive experience by incorporating native UI elements while keeping a unique and cohesive appearance.
In fact the app extensively utilizes Material Design Widget to enforce consistency between screens and also a nice look (since it's one of the most used) Important elements of the UI are :

-action buttons, that are easy to spot and clearly state their purpose, improving user interaction by making the app's features readily accessible;

-icons, that are used consistently across different screens, with each icon accurately depicting its function. For instance, a "plus" icon clearly suggests that pressing it will allow users to add new content in that section.

This design approach ensures the app remains user-friendly and easy to navigate, combining a sense of familiarity with a distinctive style.

-also a lot of other graphical elements do their part in order to make the user feel guided and supported by the UI.

4.3 Screenshots and examples

Here we put some screenshot to give a sort of overview on the app and guide a bit the reader not onnly through the architectural part (as we did till now) but also on the real front-end part. Moreover we wanted also to prove here that:

1. we were still able to do a good job with the UI design;
2. we developed an app that supports also multi-devices (and adapt the UI accordingly);
3. we also gifted our app with a nice darkmode for the low light conditions.

Now we will presented briefly some of the most important widget with a small caption, but we apologize in advance for the difference in the size of the images. We tried as much as possible not to waste too much time, so we had to change the normal dimensions.

4.3.1 Authentication, SignIn and SignUp

This screen appears when the Firebase Authentication APIs, via the AuthViewModel, indicate that the user is not logged in. By pressing the appropriate button, the user is given the option to either log in with Google through a WebView or use the standard authentication method provided by Google. Below are presented the clean authentication methods

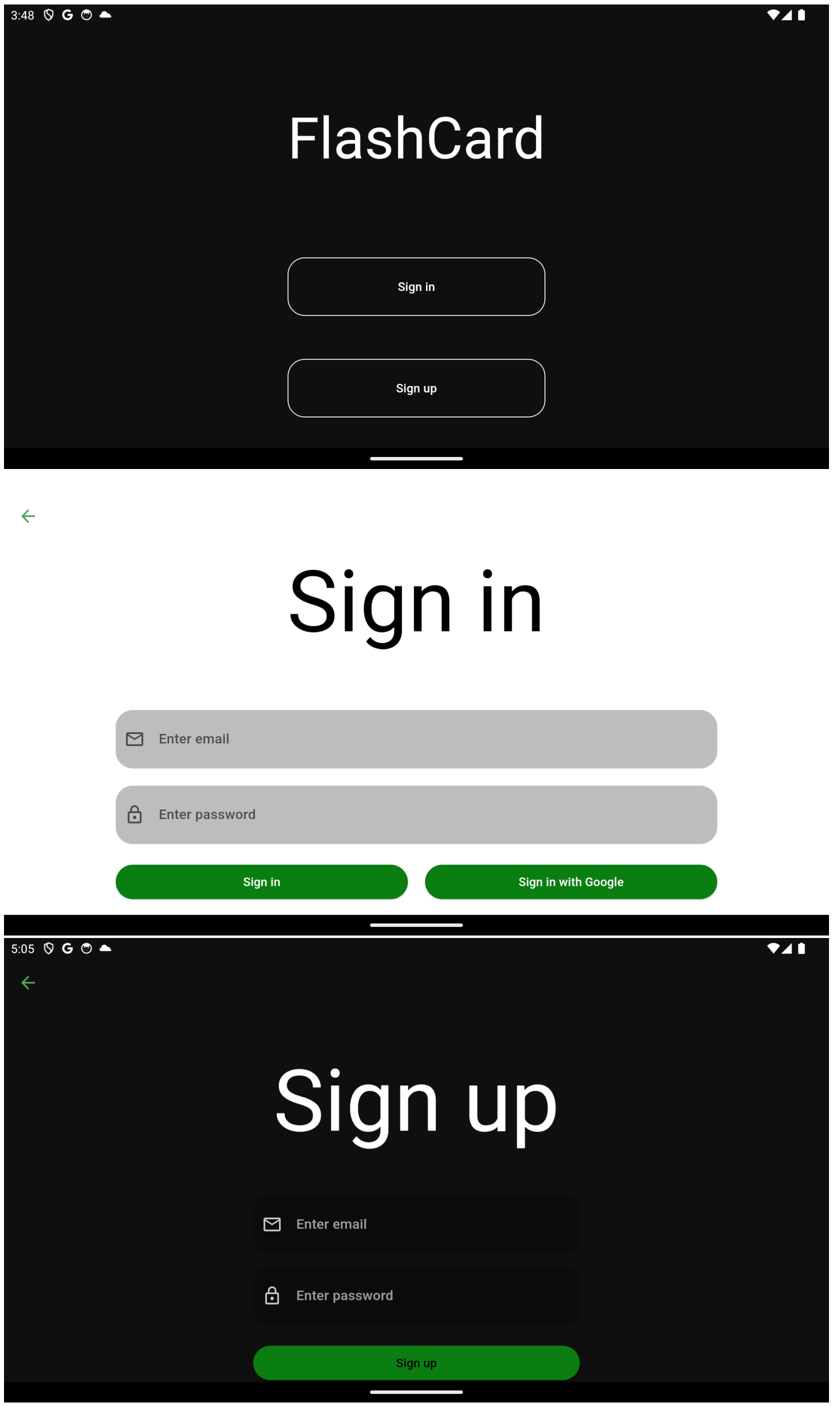
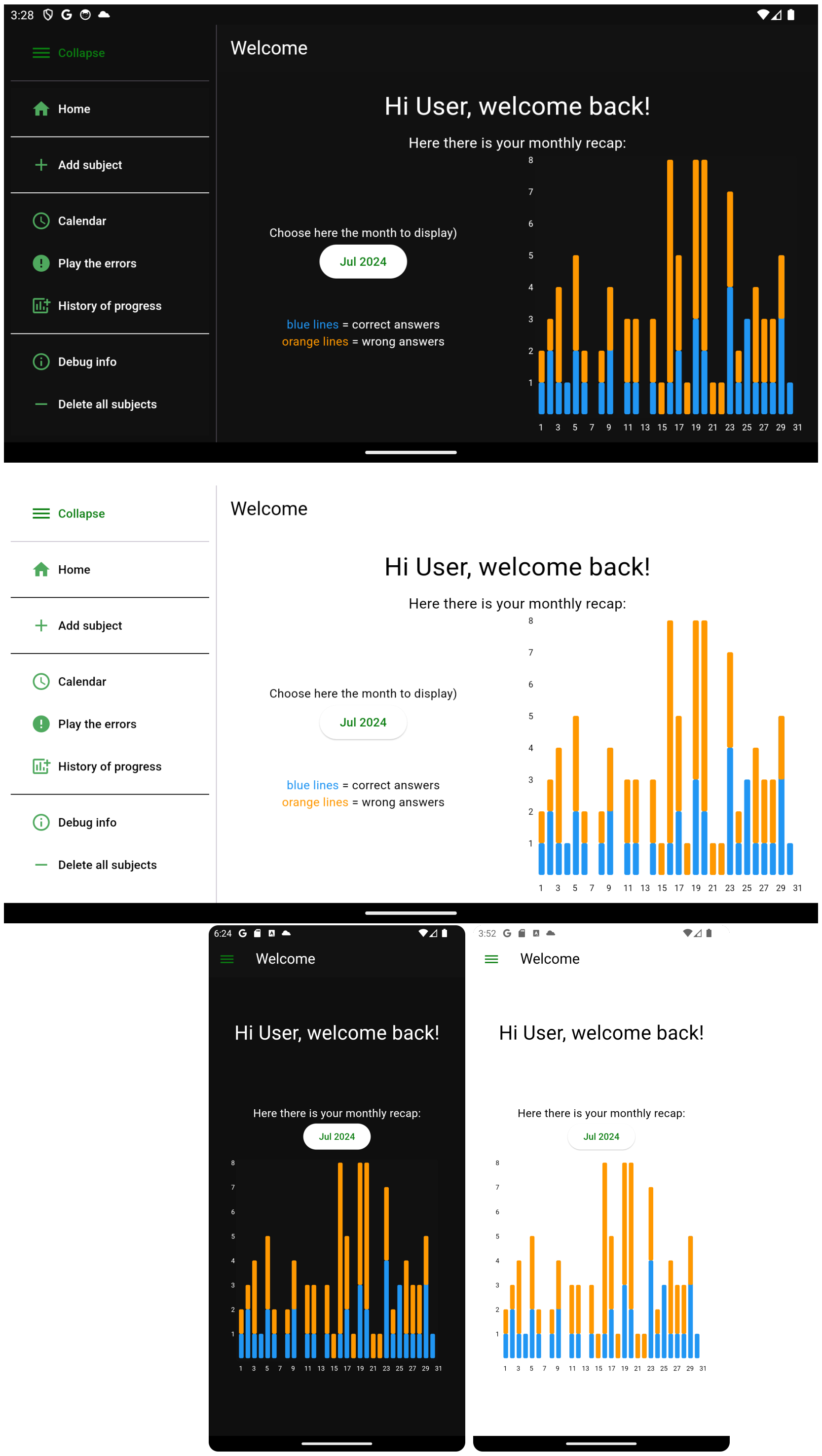


Figure 16: Authentication, SignIn and SignUp screens

4.3.2 WelcomeWidget

This is the first screen of the real app that the user sees. Here it can play with the histogram choosing which data he wants to see (according to the month and year choosen).



Hi User, welcome back!

Here there is your monthly recap:

Choose here the month to display)

blue lines = correct answers
orange lines = wrong answers

Figure 17: WelcomeWidget in all the modes and devices

4.3.3 SubjectSelection

This is the lateral drawer of the app that allows the user to select which page he/she/it wants to see. In particular is composed by the home tab (that take it back to the welcomeWidget), the section of all the subjects created (with also the button to add them). Then there is the calendarWidget button, the PlayWithErrors button and the HistoryOfErrors button (those are the name of the widget classes). Then there is the delete all subjects that is auto-explicative and the settingsWidget button that gives to the user the possibility of changing API key and changing DarkMode.
Lastly there are the backup data full and the restore data full that allows the user respectivelly to do a backup on firebase and to fetch from it (those are used to exchange data with firebase allowing consistency among devices).Finally there is the logout button.

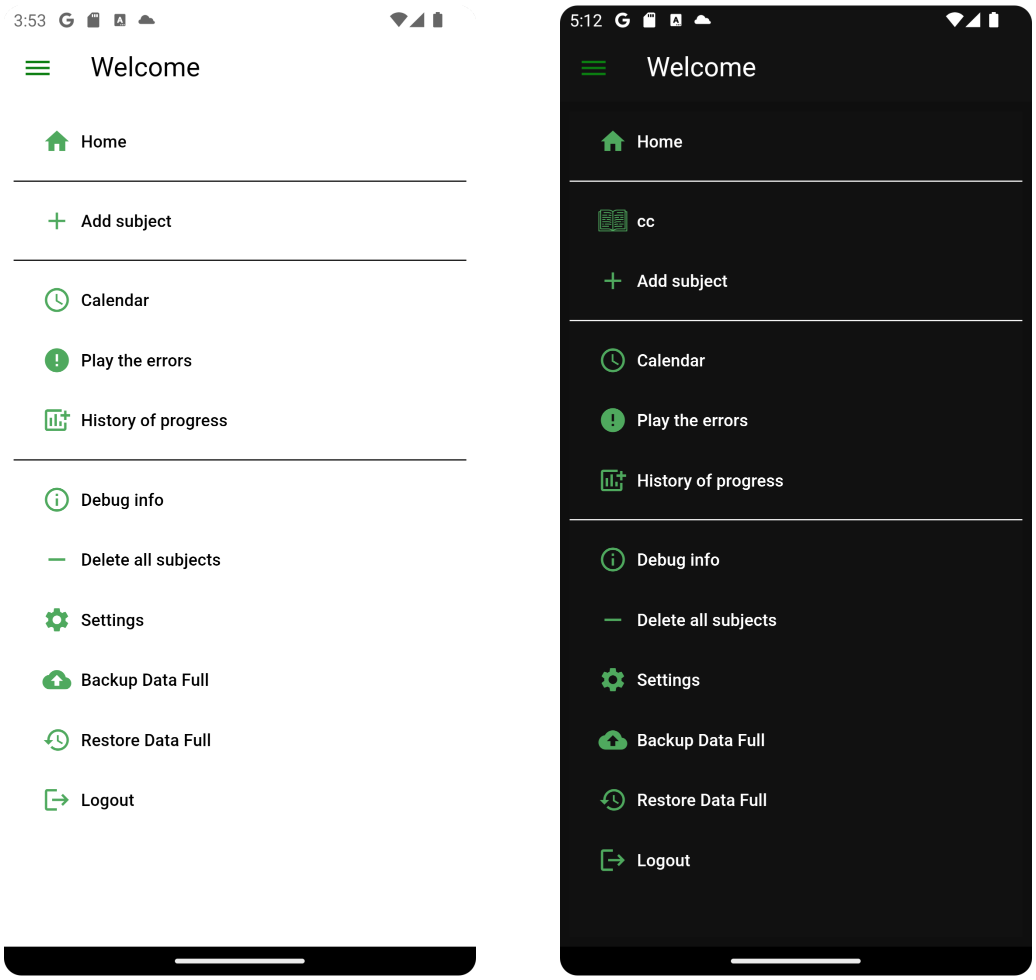
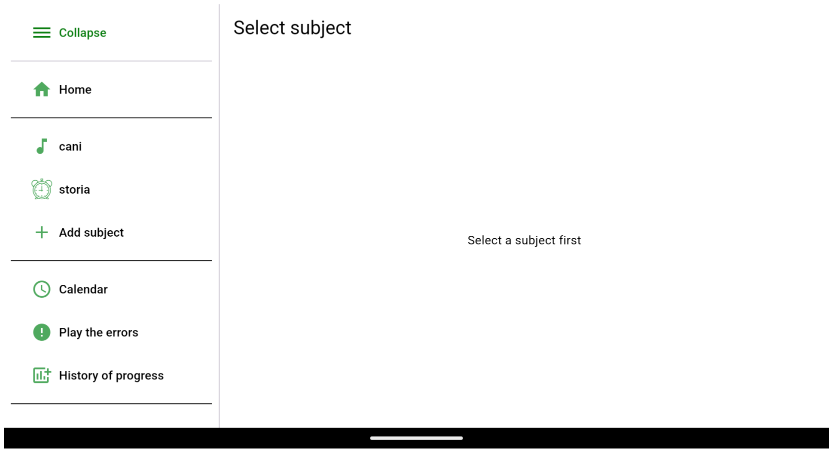


Figure 18: Subject selection (a sort of drawer of our app)



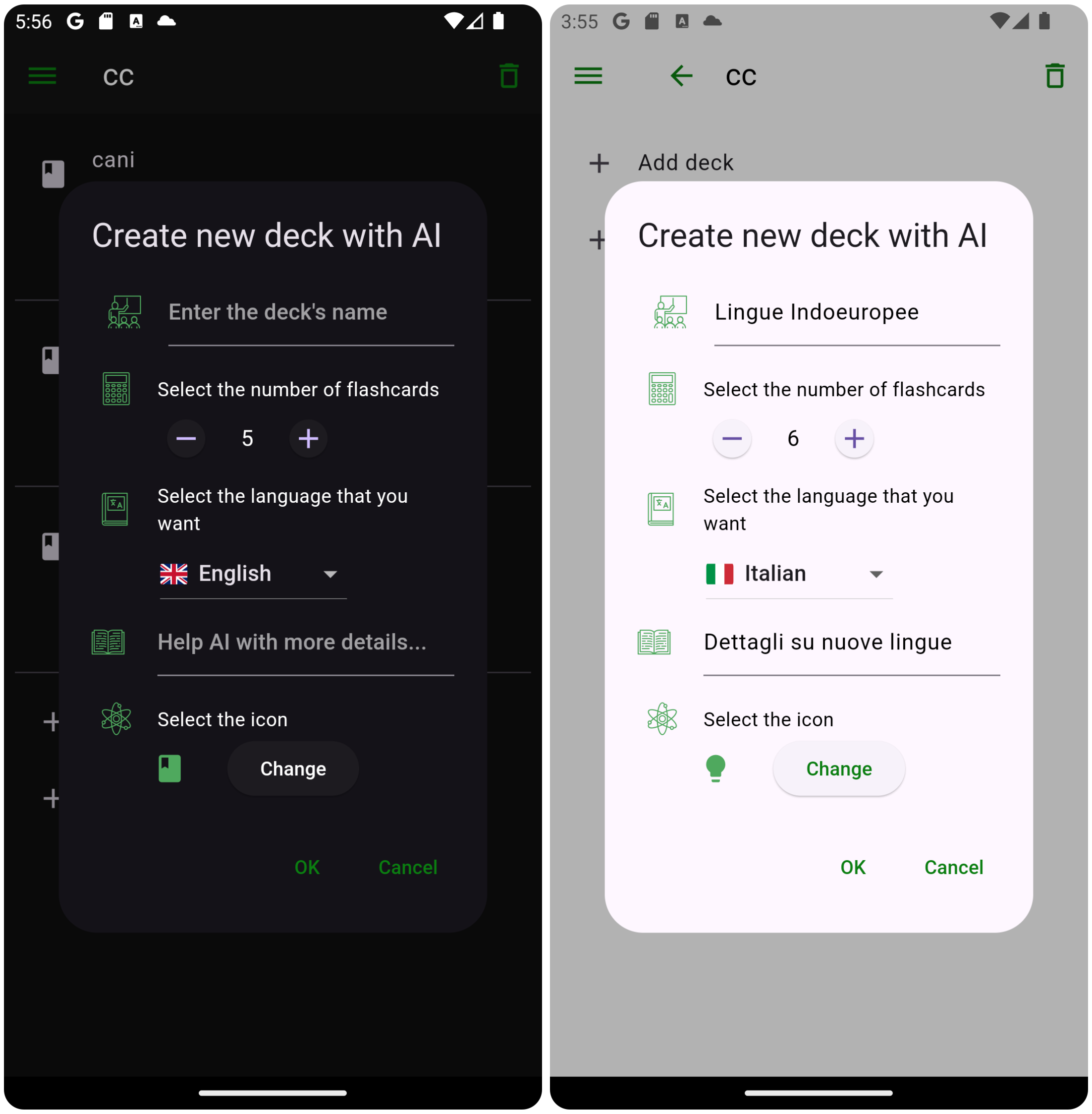
Select subject

Select a subject first

Figure 19: Here there is the screen when you don’t select any subject

4.3.4 CreateDeckWithAI

Here there is the createDeckWithAI alert dialog that allows the user to set the parameters used by chatGPT to create automatically a deck and store it in memory. In particular here it requests: the deckName, the number of flashcards to create, the language choosen, the optional description that helps ChatGPT to create a more need-specific set and finally the icon to be used to represent it in the deck selection page.


5:56

CC

cani

Create new deck with AI

Enter the deck's name

Select the number of flashcards

5

Select the language that you want

English

Help AI with more details...

Select the icon

Change

OK Cancel

3:55

CC

Add deck

Create new deck with AI

Lingue Indoeuropee

Select the number of flashcards

6

Select the language that you want

Italian

Dettagli su nuove lingue

Select the icon

Change

OK Cancel

Figure 20: AddDeckWithAI (the dialog that allow us to ask ChatGpt to create a deck

4.3.5 DeckSelection and modify

Here you can see, create, edit and remove the decks of each subject.

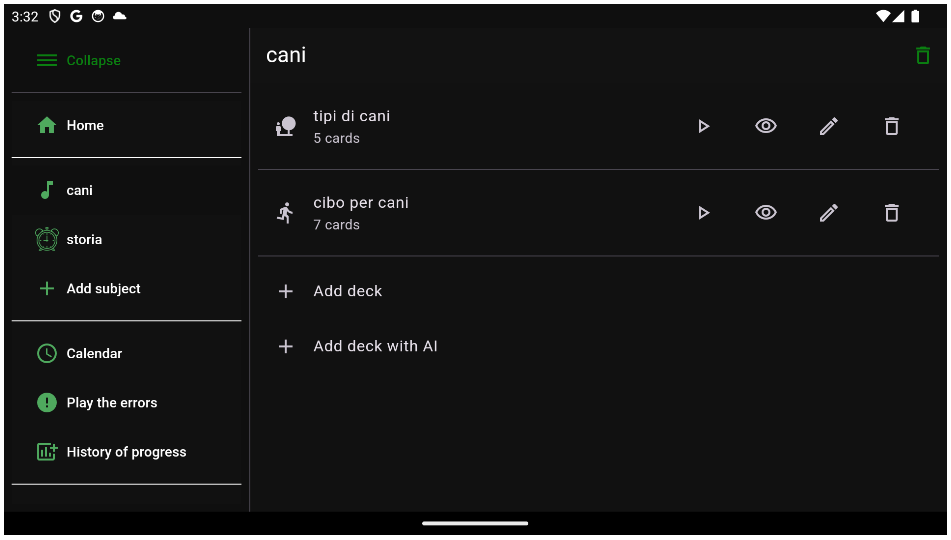
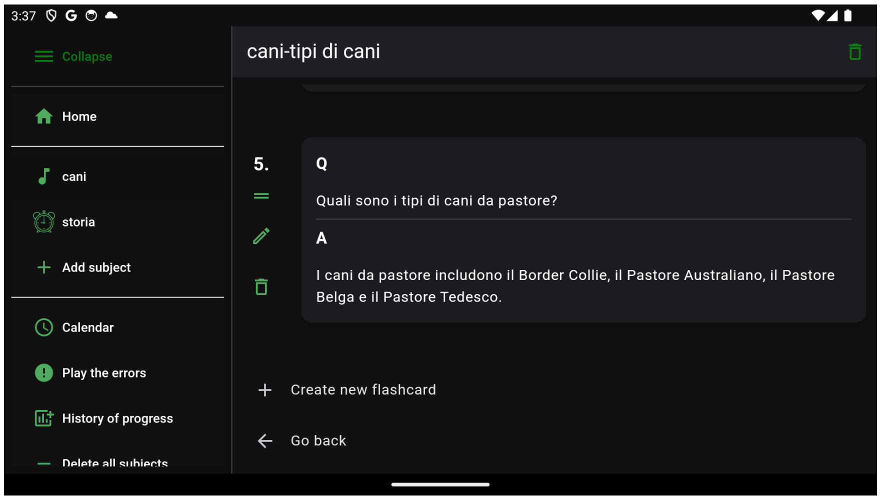


Figure 21: Deck selection (here you can seem all the decks and create new ones



Q

Quali sono i tipi di cani da pastore?

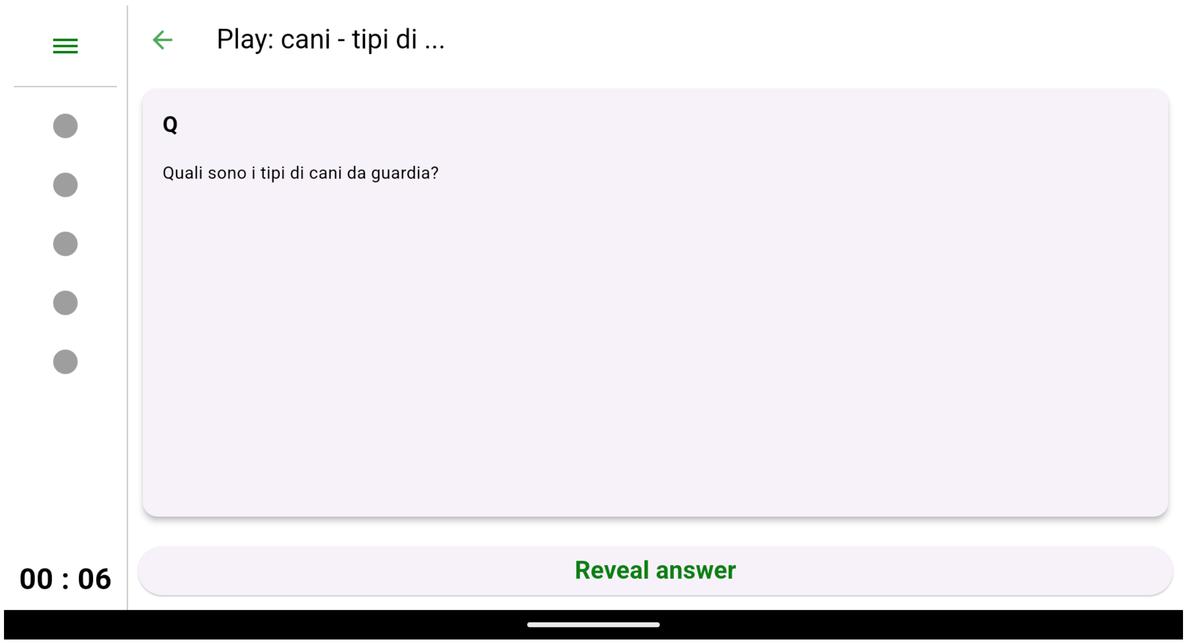
A

I cani da pastore includono il Border Collie, il Pastore Australiano, il Pastore Belga e il Pastore Tedesco.

Figure 22: Deck modify (here you can change order, modify and create new Flashcard

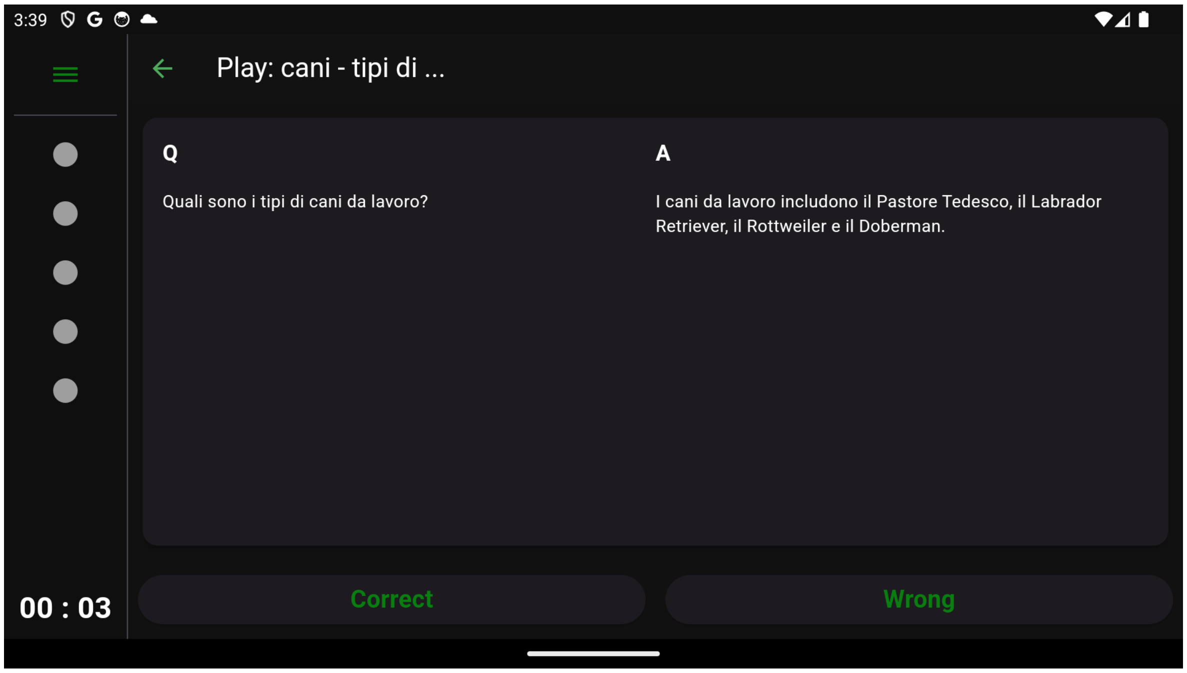
4.3.6 PlayContent

Here you can actually play the content. You will visualize the flashcard and you will reveal it after. Then you go to the next one saying if the previous one was correct or not.



Quali sono i tipi di cani da guardia?

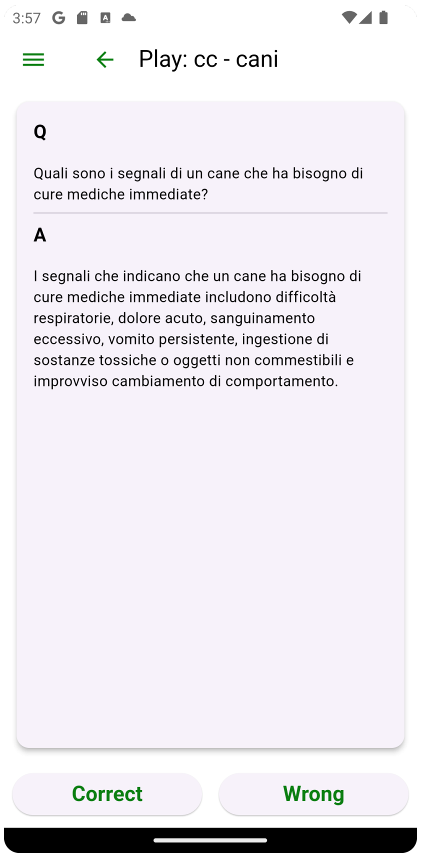
Figure 23: PlayContent (the widget that show us the flashcaard)



Quali sono i tipi di cani da lavoro?

I cani da lavoro includono il Pastore Tedesco, il Labrador Retriever, il Rottweiler e il Doberman.

Figure 24: Still in the PlayContent screen, selecting the revealing answer option



3:57

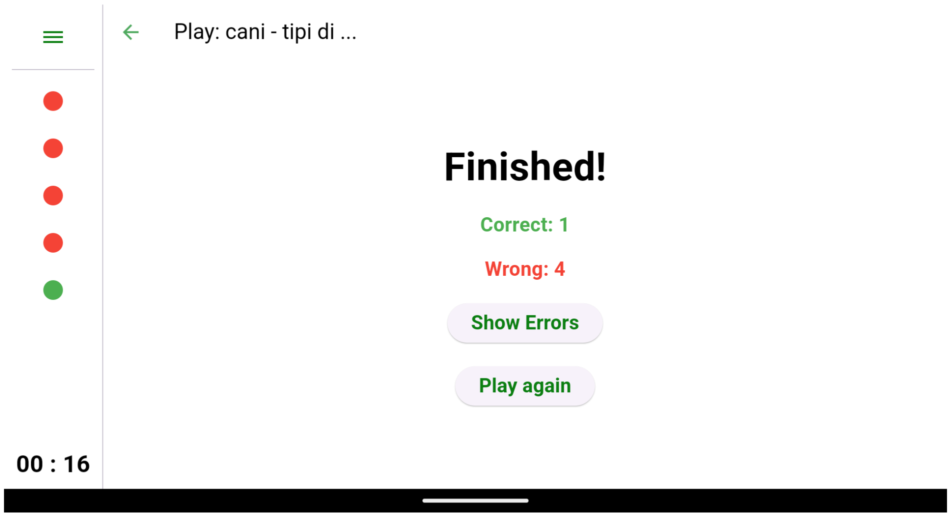
Play: cc - cani

Q

Quali sono i segnali di un cane che ha bisogno di cure mediche immediate?

A

I segnali che indicano che un cane ha bisogno di cure mediche immediate includono difficoltà respiratorie, dolore acuto, sanguinamento eccessivo, vomito persistente, ingestione di sostanze tossiche o oggetti non commestibili e improvviso cambiamento di comportamento.



Play: cani - tipi di ...

Finished!

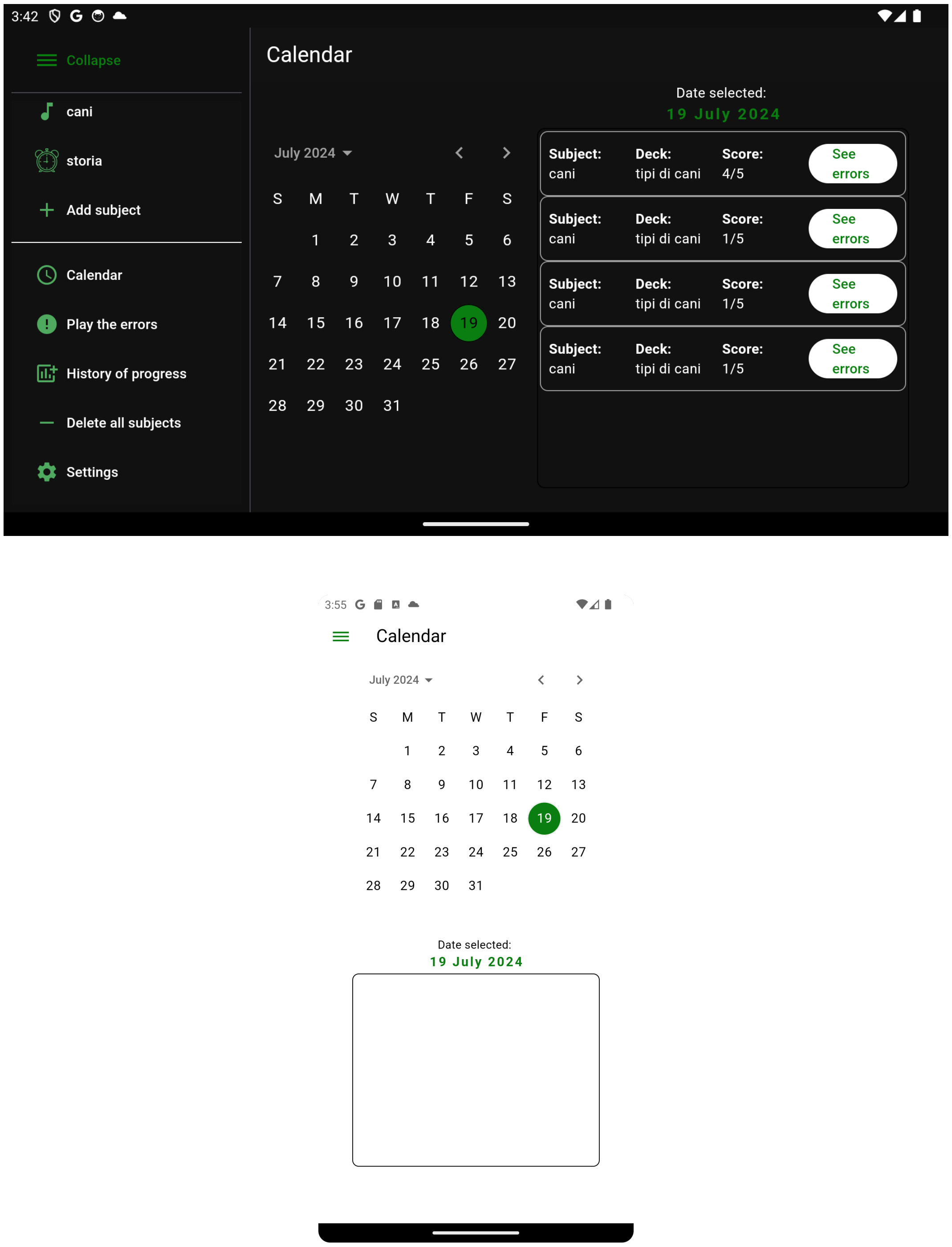
Correct: 1

Wrong: 4

Figure 25: Still in PlayContent screen, reading the correct answer and finishing the deck

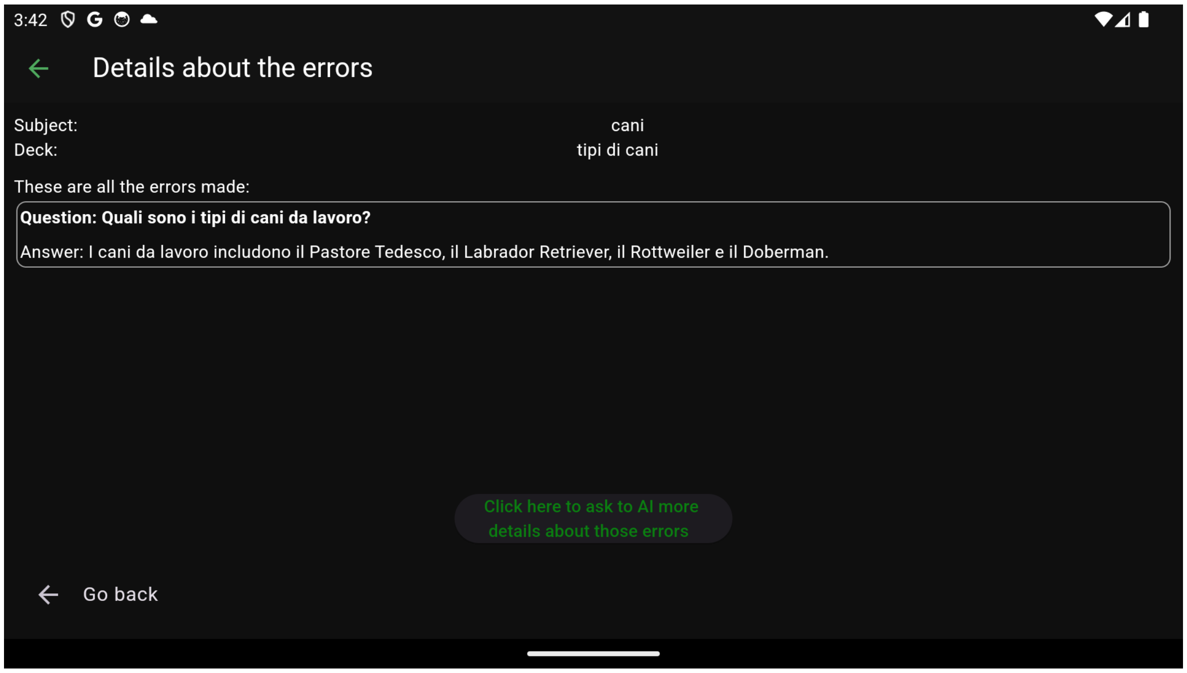
4.3.7 Calendar

The calendar widget is the entry point first to see the scores for each deck each day, but also allows the user, after selecting the deck played to see the score and to ask the AI more details about the errors. The AI will give the insights for each questions, making easier to us to memorize our mistakes and learn from it.



Subject:	Deck:	Score:	
cani	tipi di cani	4/5	See errors
cani	tipi di cani	1/5	See errors
cani	tipi di cani	1/5	See errors
cani	tipi di cani	1/5	See errors

Figure 26: Calendar widget that shows the deck played each day and the corresponding score



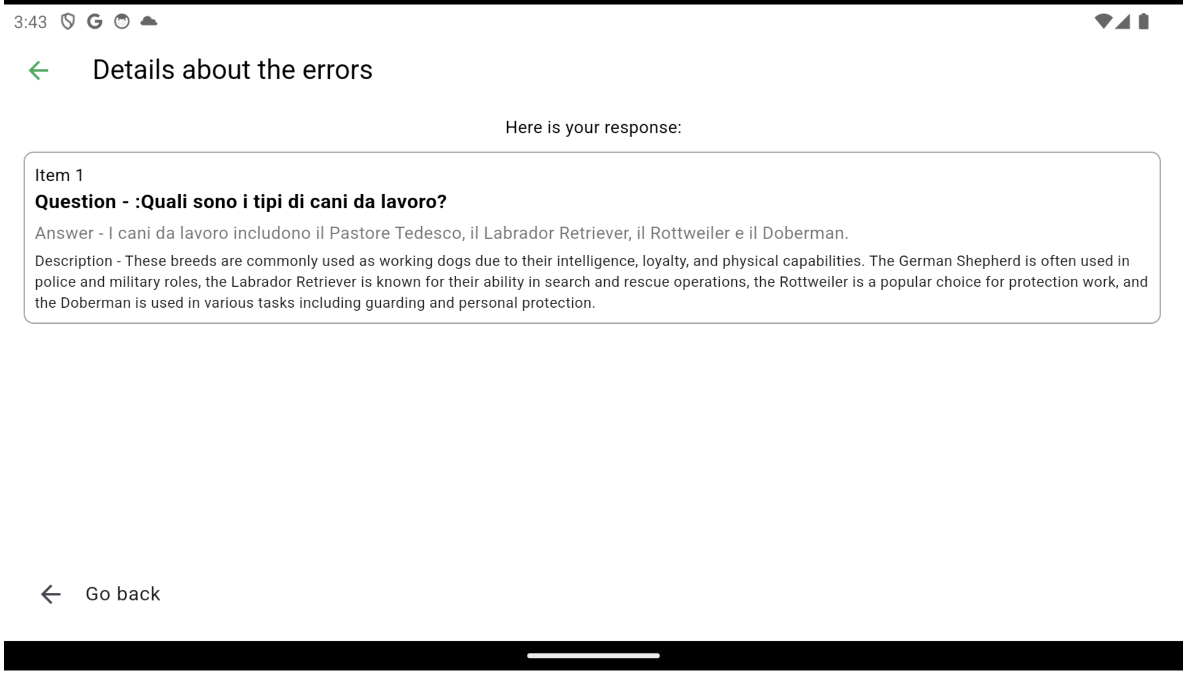
Details about the errors

Subject: cani
Deck: tipi di cani

These are all the errors made:

Question: Quali sono i tipi di cani da lavoro?

Answer: I cani da lavoro includono il Pastore Tedesco, il Labrador Retriever, il Rottweiler e il Doberman.



Details about the errors

Here is your response:

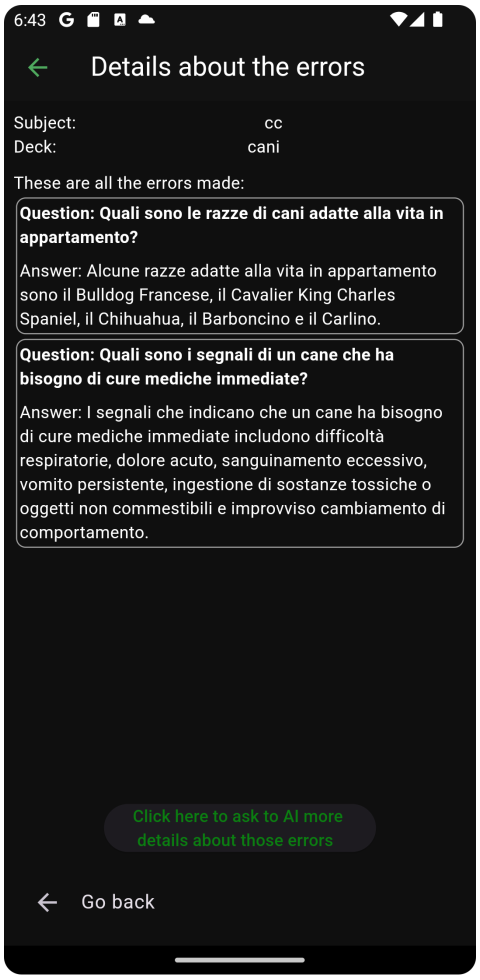
Item 1

Question - :Quali sono i tipi di cani da lavoro?

Answer - I cani da lavoro includono il Pastore Tedesco, il Labrador Retriever, il Rottweiler e il Doberman.

Description - These breeds are commonly used as working dogs due to their intelligence, loyalty, and physical capabilities. The German Shepherd is often used in police and military roles, the Labrador Retriever is known for their ability in search and rescue operations, the Rottweiler is a popular choice for protection work, and the Doberman is used in various tasks including guarding and personal protection.

Figure 27: Here are respresented both the normal description of the errors and also the one created by AI


6:43

Details about the errors

Subject: cc
Deck: cani

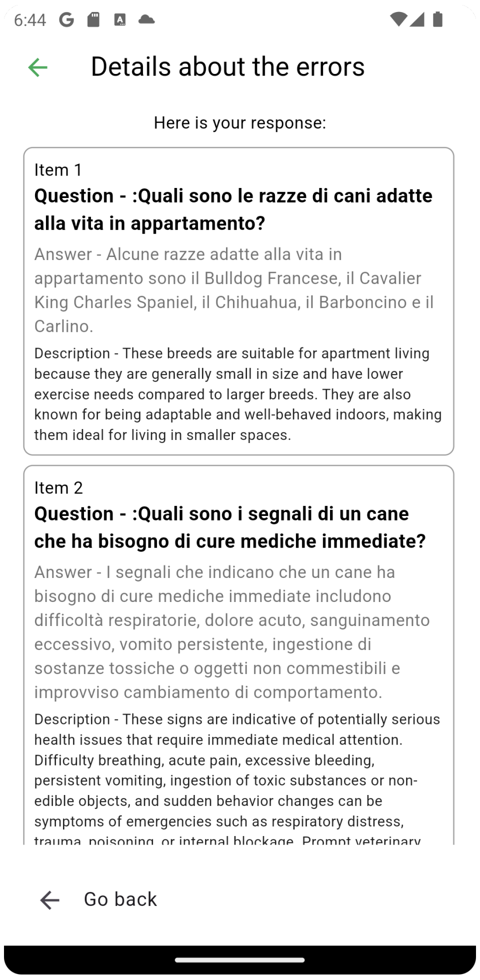
These are all the errors made:

Question: Quali sono le razze di cani adatte alla vita in appartamento?

Answer: Alcune razze adatte alla vita in appartamento sono il Bulldog Francese, il Cavalier King Charles Spaniel, il Chihuahua, il Barboncino e il Carlino.

Question: Quali sono i segnali di un cane che ha bisogno di cure mediche immediate?

Answer: I segnali che indicano che un cane ha bisogno di cure mediche immediate includono difficoltà respiratorie, dolore acuto, sanguinamento eccessivo, vomito persistente, ingestione di sostanze tossiche o oggetti non commestibili e improvviso cambiamento di comportamento.


6:44

Details about the errors

Here is your response:

Item 1

Question - :Quali sono le razze di cani adatte alla vita in appartamento?

Answer - Alcune razze adatte alla vita in appartamento sono il Bulldog Francese, il Cavalier King Charles Spaniel, il Chihuahua, il Barboncino e il Carlino.

Description - These breeds are suitable for apartment living because they are generally small in size and have lower exercise needs compared to larger breeds. They are also known for being adaptable and well-behaved indoors, making them ideal for living in smaller spaces.

Item 2

Question - :Quali sono i segnali di un cane che ha bisogno di cure mediche immediate?

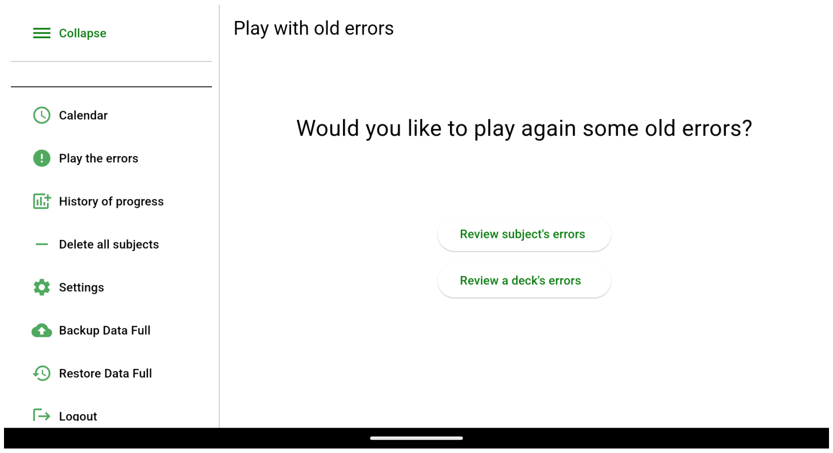
Answer - I segnali che indicano che un cane ha bisogno di cure mediche immediate includono difficoltà respiratorie, dolore acuto, sanguinamento eccessivo, vomito persistente, ingestione di sostanze tossiche o oggetti non commestibili e improvviso cambiamento di comportamento.

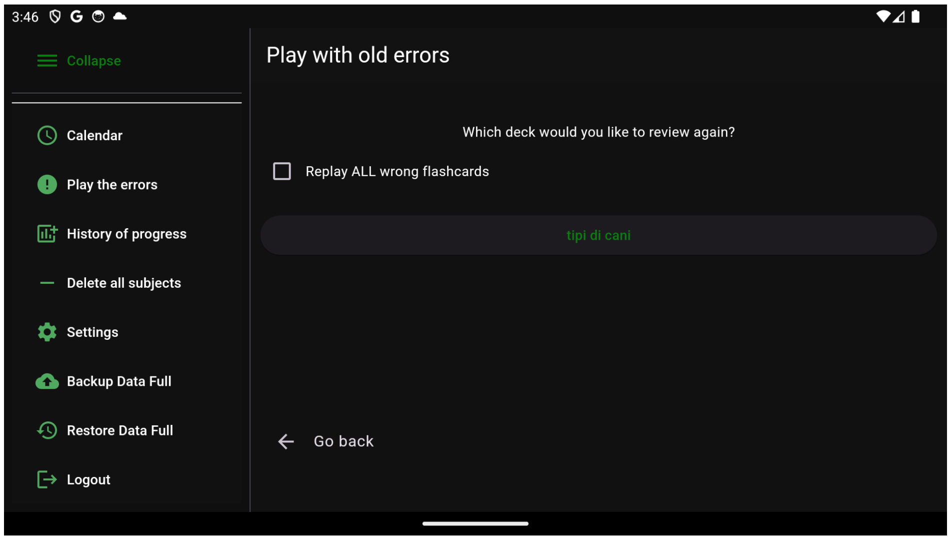
Description - These signs are indicative of potentially serious health issues that require immediate medical attention. Difficulty breathing, acute pain, excessive bleeding, persistent vomiting, ingestion of toxic substances or non-edible objects, and sudden behavior changes can be symptoms of emergencies such as respiratory distress, trauma, poisoning, or internal blockage. Prompt veterinary

Figure 28: The same also here (but with the phone)

4.3.8 Play the errors

Allows the user to play again with the mistakes that he/she/it did, grouped by decks or subjects.





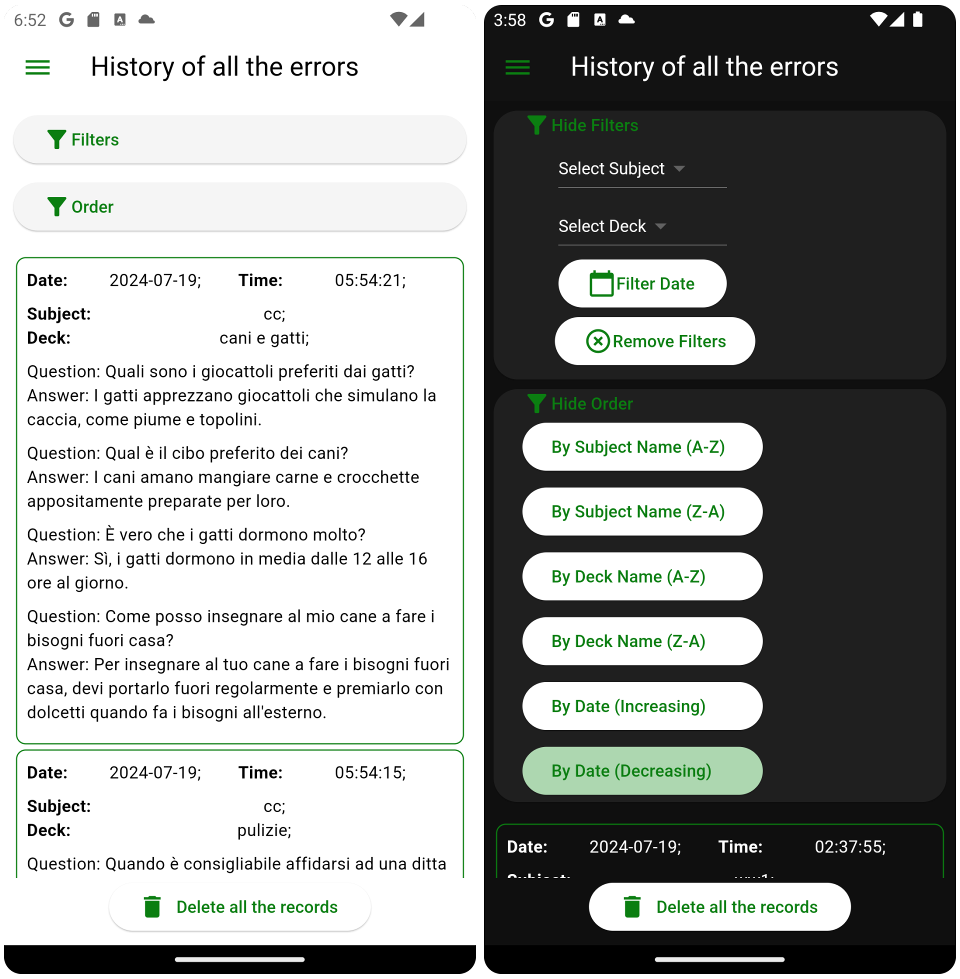
Which deck would you like to review again?

Replay ALL wrong flashcards

Figure 29: PlayWithPastErrors

4.3.9 History errors

This widget allow the user to go back again to the errors and to filter and order them by some values.



Date: 2024-07-19; **Time:** 05:54:21;
Subject: cc;
Deck: cani e gatti;

Question: Quali sono i giocattoli preferiti dai gatti?
Answer: I gatti apprezzano giocattoli che simulano la caccia, come piume e topolini.

Question: Qual è il cibo preferito dei cani?
Answer: I cani amano mangiare carne e crocchette appositamente preparate per loro.

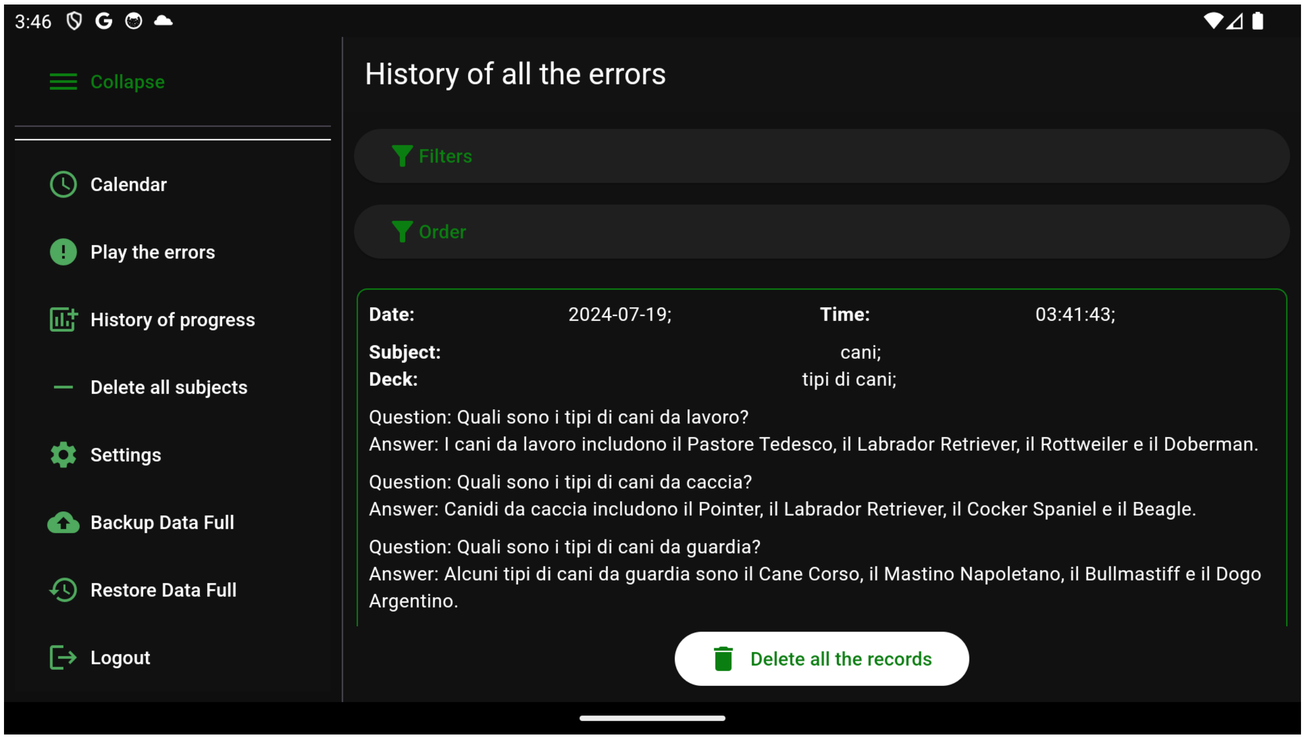
Question: È vero che i gatti dormono molto?
Answer: Sì, i gatti dormono in media dalle 12 alle 16 ore al giorno.

Question: Come posso insegnare al mio cane a fare i bisogni fuori casa?
Answer: Per insegnare al tuo cane a fare i bisogni fuori casa, devi portarlo fuori regolarmente e premiarlo con dolcetti quando fa i bisogni all'esterno.

Date: 2024-07-19; **Time:** 05:54:15;
Subject: cc;
Deck: pulizie;

Question: Quando è consigliabile affidarsi ad una ditta

Date: 2024-07-19; **Time:** 02:37:55;


3:46

Collapse

Calendar

Play the errors

History of progress

Delete all subjects

Settings

Backup Data Full

Restore Data Full

Logout

History of all the errors

Filters

Order

Date: 2024-07-19; **Time:** 03:41:43;

Subject: cani;

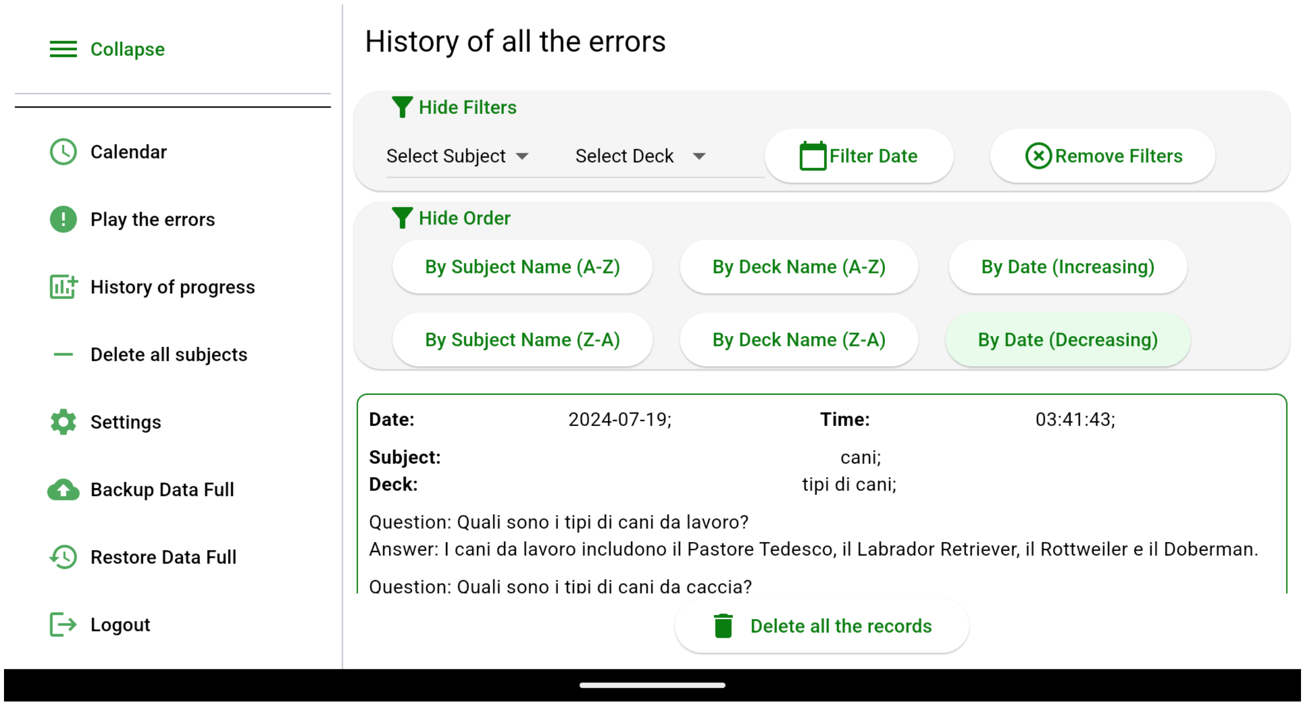
Deck: tipi di cani;

Question: Quali sono i tipi di cani da lavoro?
Answer: I cani da lavoro includono il Pastore Tedesco, il Labrador Retriever, il Rottweiler e il Doberman.

Question: Quali sono i tipi di cani da caccia?
Answer: Canidi da caccia includono il Pointer, il Labrador Retriever, il Cocker Spaniel e il Beagle.

Question: Quali sono i tipi di cani da guardia?
Answer: Alcuni tipi di cani da guardia sono il Cane Corso, il Mastino Napoletano, il Bullmastiff e il Dogo Argentino.

Delete all the records


Collapse

Calendar

Play the errors

History of progress

Delete all subjects

Settings

Backup Data Full

Restore Data Full

Logout

History of all the errors

Hide Filters

Select Subject | Select Deck | Filter Date | Remove Filters

Hide Order

By Subject Name (A-Z) | By Deck Name (A-Z) | By Date (Increasing)

By Subject Name (Z-A) | By Deck Name (Z-A) | By Date (Decreasing)

Date: 2024-07-19; **Time:** 03:41:43;

Subject: cani;

Deck: tipi di cani;

Question: Quali sono i tipi di cani da lavoro?
Answer: I cani da lavoro includono il Pastore Tedesco, il Labrador Retriever, il Rottweiler e il Doberman.

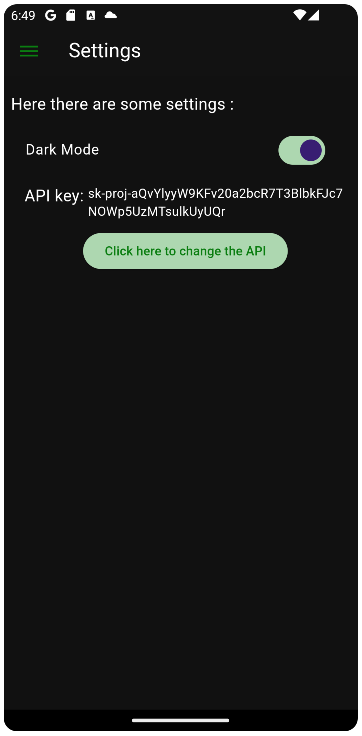
Question: Quali sono i tipi di cani da caccia?

Delete all the records

Figure 30: HistoryOfProgress with also the filtering and ordering buttons activated/not activated

4.3.10 Setting

This widget allow the user to see or modify it's own API_KEY and also to decide which UI to see: dark or light.


6:49

Settings

Here there are some settings :

Dark Mode

API key: sk-proj-aQvYlyyW9KFv20a2bcR7T3BlbkFJc7NOWp5UzMTsulkUyUQr

Figure 31: Setting screen

5 Testing and Feedbacks

5.1 Overview

In our development strategy, we decided to focus mainly on user testing, though we did implement some basic tests. Specifically, we performed tests on our models and local repository, as well as widget testing for key components. However, we faced challenges in developing extensive tests due to the strong coupling with Firebase. The necessity of fetching data from Firebase for every request impeded the testing process, making it difficult to isolate components for integration testing. Consequently, our core testing approach has been user testing.

While unit and integration testing can provide important boundaries within which to develop safely, we deemed them less crucial for this project due to the straightforward and uncomplicated nature of the app's functionalities. Implementing extensive unit testing would involve setting up and instantiating Bloc components repeatedly, which we found to be time-consuming and inefficient given the app's predictable behavior.

Our priority was to gather direct feedback from our target audience through user testing. This method allows us to adapt and refine the app based on real-world usage scenarios and user interactions quickly. By focusing on user testing, we aim to iterate more efficiently and align our development efforts with the specific needs and expectations of our users in an agile manner. This iterative process ensures that we can deliver a more polished and user-friendly app experience while acknowledging the inherent risks of not conducting extensive unit and integration testing.

5.2 Unit, Widget and Integration Testing

As we was saying, we did Unit Testing on all the model part and we also tested the LocalRepository-Service class with all the methods.

This is a small table to recap each functionality tested :

Name of the class tested	Name of the test	What it tests
Subject	fromJson should return a valid model	Verify that Subject model is correctly instantiated from JSON data
Subject	toJson should return a valid JSON map	Ensure Subject model is properly converted to JSON format
Subject	copyWith should return a valid copy with updated fields	Check that Subject.copyWith method returns a correct copy with specified field updates
Subject	compareTo should compare by name	Test Subject.compareTo method to ensure it sorts Subjects by name correctly
Subject	toString should return a JSON string representation	Confirm that Subject.toString method returns a JSON string representation
Deck	toJson should return a valid JSON map	Ensure Deck model is properly converted to JSON format
Deck	copyWith should return a valid copy with updated fields	Check that Deck.copyWith method returns a correct copy with specified field updates
Deck	compareTo should compare by name	Test Deck.compareTo method to ensure it sorts Decks by name correctly
Deck	toString should return a JSON string representation	Confirm that Deck.toString method returns a JSON string representation
Flashcard	fromJson should return a valid model	Verify that Flashcard model is correctly instantiated from JSON data
Flashcard	toJson should return a valid JSON map	Ensure Flashcard model is properly converted to JSON format
Flashcard	toJson should return a valid JSON map	Ensure Deck model is properly converted to JSON format
Flashcard	copyWith should return a valid copy with updated fields	Check that Flashcard.copyWith method returns a correct copy with specified field updates
Flashcard	compareTo should compare by index	Test Flashcard.compareTo method to ensure it sorts Flashcards by index correctly
LocalRepositoryService	addNewSubject should add a subject to local storage	Ensure LocalRepositoryService.addNewSubject correctly stores a new subject
LocalRepositoryService	getSubjects should return a list of subjects from local storage	Verify LocalRepositoryService.getSubjects returns all stored subjects
LocalRepositoryService	addNewDeck should add a deck to a subject	Ensure LocalRepositoryService.addNewDeck correctly adds a new deck to a subject
LocalRepositoryService	addNewSubject should add a subject to local storage	Ensure LocalRepositoryService.addNewSubject correctly stores a new subject
LocalRepositoryService	addNewFlashcard should add a flashcard to a deck	Verify LocalRepositoryService.addNewFlashcard correctly adds a new flashcard to a deck

Table 2: This table explain all the test made to test the model

Then we also did some Widget Testing. I will report below the most importan ones:

Name of the widget tested	Name of the test	What it tests
PlayWidget	Shows start button initially	Ensure the 'Start' button is displayed initially
PlayWidget	Shows first flashcard when started	Verify the first flashcard is displayed when the game starts
PlayWidget	Shows second flashcard when answered correctly	Check that the second flashcard is shown after the first one is answered correctly
PlayWidget	Shows finished when all flashcards answered	Confirm that 'Finished!' is displayed when all flashcards are answered
SubjectWidget	Shows Add Subject button initially	Ensure the 'Add Subject' button is displayed initially
SubjectWidget	Adds subject and shows subject name	Verify that adding a subject displays the subject name and 'Add Deck' button
SubjectWidget	Adds deck and shows deck name	Check that adding a deck displays the deck name

Table 3: This table explain all the test made to test the model

5.3 User Testing

We split our testing into two phases.
In early March, we asked a small group of less than 10 users to give us feedback. By that time, our app had all the basics features: creating, editing, and deleting decks, subjects, and flashcards and AI features. So we were a bit blocked and that's why we needed to know what more we could add to make the app even better.
Then, in June, we did another round of testing to fix any remaining issues and polish the app based on the earlier feedback. This phase focused on ironing out bugs and making sure everything ran smoothly before the final release. So in this second phase fifteen individuals, primarily students aged 18 to 25, were invited to try out our flashcard app for a week and share their feedback. Some testers were briefed on the app's features beforehand, while others approached it without prior knowledge. Technical expertise varied among the participants, affecting their ability to spot and describe issues they encountered.

Our beta testing focused on key goals: assessing app quality, gathering feedback to improve user experience—especially from less experienced users and identifying and fixing hard-to-find bugs. Testers provided feedback through screenshots or detailed descriptions of any errors they encountered. This feedback has been crucial for refining and enhancing the app before its official release.

Through the 2 phases of beta-testing we where able to evaluate new suggestion and to detect some hidden bug.
Here I will report some tables to list them:

1. Here there is the first table summerizing all the possible feature to implement and the corresponding changes done that we collected after the first feedback phase.

Possible features to add or changes	Evaluation and changes made
Try to think about a way to keep track of errors and progress	Solved adding 3 features: 1- adding the histogram in the WelcomeWidget 2- adding the history of progress tab, to have the complete log of all the errors made 3- adding of the calendar widget and also the details about the error made
Try to think about some way of personalizing the learn process	Solved adding the possibility to customize the icons, so that, just with a quick glance, we could understand the deck or subject that we need
Possibility to play the flashcard offline, without a connection	At the beginning, as stated in the Overview part of the Architecture, we implemented all the app and all the functionalities storing the data on the disk, but then with the support to multiple users and the consequent need of using the authentication with firebase, the app become no more suitable to act in a offline environment. Hence we decide to still do some investigation on how we could do it. We understood that more or less there is a simple way of doing it with flutter. But then, since at that point we have already decided to start implementing the AI part we decide to postpone in the roadmap this idea because it would conflict with the need of internet connection to generate the new flashcards. Moreover the probability of being offline in the current history period is only truth when travelling by plane, going to very high mountains or other small causes. So we decide to start developing other feature a

Possible features to add or changes	Evaluation and changes made
Possibility to add photo to the flashcards	This idea was delayed in the roadmap because would require to save all the images and could become difficult to handle in case of scaling of the app. Moreover if we would decide to implement the firebase part, this feature would require to store also online the pictures and will be again difficult to scale (compare instead to the string fields). Still we decide not to discard the idea because we recognize the importance of being able to save an image instead of a text. Also because sometimes is also more expressive (with an image I can understand quicker something)
Possibility to add a generated photo by the AI	We admit that would be a good idea, but at that time the generation process of images by the AI was more complex and Dall-E wasn't embedded yet in the Chat-Gpt package as it is now. Moreover that would still require to save it and this could be a problem, as stated in the previous point
Change the colors of the UI because are a bit too much bright and make the text different to have	We changed the palette to accomodate the request, because we realized that this could be a serious problem in case for example of not confortable condition of high brightness or low one (when sleeping or when going out with the sun)
Possibility to have an app that support multiple languages	This feature was also delayed in the roadmap because of more urgent features to develop. Moreover the app till that point was very simple so it was not so required to support languages. Also the introduction of icons made the app easier to understand and so with less needs of texts.

Possible features to add or changes	Evaluation and changes made
Integration of audio messages embedded in the flashcards	We admit that was a nice suggestion, also considering the final goal in a log term: to become an hub of learning. On the other hand however they are less accessible compared to text or images and also they has the same problem of storing presented in the "Possibility to add a generated photo by the AI" feature (so the storing problem).
Presence of buttons to record audio and transcript to text	This was a very interesting idea. I (Giacomo) tried personally in another app to do that and I completed the task. The only problem was that the flutter package used couldn't set the maximum time of recording and the normal Android period is linked to the time in which the sensor doesn't ear anything. So in case also of a small pause (less than 2 seconds) the service stop the acquisition of audio, making the process of transcripting less efficient. So at the end we decide not to integrate it here.
Possibility to customize the icons	We decided that was a good idea for the personalizabilty and so we decided to implement it. Also because it was very ez.

Table 4: This table explain all the suggestion received by the users.

Then also we had some proper testing results on what to debug or solve, so I will provide also a small table below:

Problems/Bugs	Resolution
When generating the deck with the AI it was often an issue, so that it didn't complete the process and didn't generate the deck	simply try to be more specific in the request. So we designed a message that it doesn't produce an accurate solution only in less then the 5% of cases
Creation of subject and deck with the same name or empty name was allowed	We implemented a check to be sure that the states of the app remains always correct (so no empty name allowed and also no duplication among deck or subjects)
Once generated the response it wasn't impossible to cancel it	Implementation of checks to avoid this situation, but still not completely avoidable since the not synchronized operation between UI and external services (ChatGPT). So it can still happen this situation, but at least we reduced the problem.
Overflows on the long titles	Implemented a sort of trimming function to cut the title displayed ONLY in the title part so that there is no way to exceed the space available.
Each time to see the subject from the drawer screen (our SelectSubject) we had to click on the expanded button in the top-left part of the screen (distant) and then also to the subject	We simply implement the trigger for the subject so that it was automatic

Table 5: This table explain all the bugs found by the users in the first beta-testing phase and the corresponding solutions

2. Then in June we did a more generic and wide sperimentation, because also we had more feature and complexity that could hide bugs. So here there are all the bugs/issues founded and the corresponding solutions.

Problems/Bugs	Resolution
Issue with the rendering of the recap of the wrong ansswers at the end of a game.	We didn't notice this bug since we always tested with small numbers of flashcard, so we didn't sa that it was cutting the widget, denying the possibility of seen the corrections. We simply tried to understand which was the component that gave that problem and it was the singleChildScrollView nested in a wrong hierarchy

Problems/Bugs	Resolution
A lot of overflows	Because of the different configurations of the devices there where inevitably issues with overflows. Legends told that still now they can be seen sometimes in the corners and in the edges of the most unpredictable widgets. To mitigate this fact I tried to manage as much as possible all the configurations.
Issues with the history filtering	Thanks to a user I noticed that we where filtering on the savings not by the actual subjects/decks already played, but on the entire set of decks and subjects. That required to change a bit the list on which we where iterating.
Issues with the API_KEY usage	The api_key is the key that we need to access the ChatGPT endpoint to retrieve the data that we need. So the problem, more in details, was on the association between each user and it's own key. We did use at the beginning only 1 key since it was the only one that we had. And this is ok. The issue was that we didn't recall that we have still to implement this function and we went on with our abstaction of 1 key. Obviously we understood at the first glance where there was the problem and we simply implemented a Map <String,String > that was able to retrieve the key assigned to each user. In particular the key was stored in local for ease of coding (using the path_provider) service.

Table 6: This table highlights the bug founded by the users and their solutions in the second beta-testing phase

3. Then, to conclude, we want to insert some of the most important user tests tried and the results :

Task	Description	Resolution
Verify Authentication	-Press signup (User1), Enter the email and password to create an account	All correct, no errors detected
Verify Api_Key Null	-User1 clicks on the setting button, Check no API key is set	All correct, no errors detected
Verify Api_Key Insertion	-User1 adds the key by clicking on change the API, -Check key is inserted (User1)	All correct, no errors detected
Verify Key Online and Fetch	-Backup (User1), -Restore in another device (User2), -Check the Api_ is updated (User2)	All correct, no errors detected
Create Subject and Sharing	-Create Subject (User1), -Backup (User1), -Restore (User2)	All correct, no errors detected
Create Deck and Sharing	-Create deckWithAI (User2), -Backup (User2), -Restore (User1)	All correct, no errors detected
Check Delete All	-Delete all with deck/subject (User2), -Check no data saved : no errors, no subjects, no decks, but Api_Key still setted) (User2)	All correct, no errors detected
Error Saving Online and Fetch	-Add deck (User1), -Backup (User1), -Reload (User2), -Play deck (User1), -Backup (User1), -Restore (User2), -Check for history (User2), -Filters available (User2)	All correct, no errors detected
Check Update Play (Again)	-Play (User1), -Backup (User1), -Restore (User2), -Check for updates in errors (User2)	All correct, no errors detected
Check Delete Locally and Sharing Online	-DeleteAllRecords (User2), -Backup (User2), -Login (User1), -Restore (User1), -DeleteRecords (User2), -Backup (User2), -Logout (User2), Restore (User1), -Check delete all the errors (User1)	All correct, no errors detected
Backup with Changes on the Subject	-Delete Subjects (User2), -Backup (User2), -Restore (User1), -Check all the subjects are deleted (User1)	All correct, no errors detected
Check Backup After Add Subject and Play	-Add subject (User2), -Add deck with AI (User2), -Play (User2), -Check show all the errors (User2), -Check history with previous play (User2), -Backup (User2), -Restore (User1), -Check errors arrived (User1)	All correct, no errors detected

Task	Description	Resolution
Check All Things Deleted with Backup	-Delete all errors present (User1), -Backup (User1), -Restore (User2), -Check no errors inside the historyErrors (User2)	All correct, no errors detected
Check Update Histogram	Play (User2), -Verify histogram (User2)	Errors in the drawing part
Check Parameters Stored in Add Subject and Add Deck with AI	-Create subject (User2) (choosing icons), -Check icon set (User2), -Add deck with AI (User2) (with parameters), -Check all parameters (User2)	All correct, no errors detected
Check Backup and Restore Consistency	-Backup (User2), -Restore (User2), -Check subjects/decks/errors (User2)	All correct, no errors detected
Check Date Filter on History Errors	-Check no results in history for other dates (User2)	ERROR detected, unexpected clean after backup
Consistency Between Calendars and Errors	-Play (User1), -Check calendar with errors (User1), -Check history with errors (User1), -Backup (User1), -Restore (User2), -Check calendar with errors and history (User1, User2)	All correct, no errors detected
Check Play Behavior and Error Savings	-Play (User1), -Check error in calendar (User1), -Click show errors (User1), -Check errors correct (User1), -Click askDescriptionAI, -Check working (with all the responses/Language), -Go back twice (User1), -Backup (User1), -Restore (User2), -Check errors in calendar and history	All correct, no errors detected
Play the Errors	-Review deck errors (User1), -Choose deck (User1), -Play deck (User1), -Check in the calendar (User1), - ...Same for the subject, -Check name Review	All correct, no errors detected

Task	Description	Resolution
Check Functionality Filtering of History Errors	-Create subject and deck (User1), -Backup (User1), -Restore (User2), -Check decks and subjects updates (User2), -Add flashcard to some of them (User1), -Backup (User1), Restore (User2), -Check flashcard (User2), -Check filters do not show the new ones (since not played yet) (User2), -Play (User2), -Check subject filtering (User2), -Check deck filtering (no mixing allowed, correct filter) (User2), -Check ordering (with also filtering) (User2)	All correct, no errors detected
Logout	-Logout (User2), -Check all the errors/calendar/key/subject/deck (User2), -Logout (User2), -Sign in (User1), -Check all (User1), -Create new things and play, check errors (User1), -Logout (User1), -Sign in (User2), -Check all new things updated (User2)	All correct, no errors detected

Table 7: This table summerize the most important user tests done

5.4 Future development

Looking ahead, while our app currently meets its core objectives, we see opportunities to enhance its functionality with a few key features:

1. -Chat Interface with AI: Introducing a chat feature will allow users to interact more dynamically with AI, seeking insights and explanations tailored to their learning needs. This addition aims to provide deeper engagement and support, especially for complex topics.

2. -Multi-Language Support: Integrating the flutter_localizations package will enable the app to support multiple languages. This enhancement is essential for broadening accessibility and accommodating users worldwide.

3. -Image Support for Flashcards: users to embed images into flashcards—captured from the camera or uploaded from their devices—will enrich learning materials. This feature is particularly beneficial for visual learners and can enhance the app's utility as a comprehensive learning tool.

These planned enhancements aim to elevate the app beyond its current capabilities, making it not only a functional tool but also a versatile platform for daily learning activities. Each feature aligns with our roadmap for future development, aimed at improving user experience and expanding the app's utility in educational contexts.
Now it remains only to start creating again!