AY 2022/2023

POLITECNICO
MILANO 1863

QuizTime

Design and Implementation
of Mobile Application:

Design Document

Andrea Paolo Arbasino Alberto Panzanini

Professor
Luciano Baresi

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide an exhausting explanation of the S2B, focusing in particular on the architecture that will be adopted, the modules of the system and their interfaces. Furthermore, a series of interaction diagrams will be provided to show the workflow of the application. Finally, some quick notes about the implementation, integration and testing processes will be described.

1.2 Project description

QuizTime is an application that empowers users to create, edit, share and take quizzes effortlessly. With QuizTime, users will have the ability to craft a diverse range of quizzes, from educational assessments to fun and interactive trivia games, reaching a wide spectrum of audiences.
Namely, the main goals of the application are:

- Allow users to create a quiz easily thanks to the **intuitive interface** and share it with everyone.
- Take a quiz and see the results anywhere, at any time.
- Target any age group thanks to the intuitive interface.
- Help educational institutions and individuals design and take quizzes to assess and reinforce people's learning.

1.3 Key features

1.3.1 Sign in through different services

A user can sign in (or sign up) with their email address or with the Single-Sign-On (SSO) functionality with some supported identity providers.

1.3.2 Create a new quiz

A user can create a new quiz, which consists in a series of questions, to test other users' knowledge about specific topics. Each question type can be either true/false or multiple-choice, to accommodate diverse content requirements.

1.3.3 Edit a quiz

A user can easily update the content of a quiz. The flexibility of the editing feature ensures that creators can continuously improve and refine their quizzes based on feedback and evolving requirements.

1.3.4 Share a quiz

A user can share a quiz with other users in different easy ways. They can share a quiz through a unique quiz ID, the user ID of the creator, or the name of the quiz.

1.3.5 Search for a quiz

A user can search for a quiz through the quiz ID, the ID of the author, or the name of the quiz.

1.3.6 Take a quiz

A user can take a quiz that was previously found as the result of a research.

1.3.7 Search for definitions during a quiz

A user can search for the definition of words inside the question text or answers to a question.

1.3.8 View the results of a quiz

A user can see the results and the solutions of a previously taken quiz, at any time. This feature ensures that participants can review quizzes at their convenience, making the learning experience more flexible and personalized.

1.3.9 View the results of the participants of a quiz

A user can see the results of every participant who took a quiz created by him/her.

1.3.10 Multi-theme support

A user can modify the theme of the application: it can either be light or dark.

1.4 Definitions, Acronyms, Abbreviations

- **Android**: It is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets.
- **iOS**: It is a mobile operating system created and developed by Apple Inc. exclusively for its hardware.
- **Dart**: It is a programming language designed for client development, such as for web and mobile apps. It is developed by Google and can also be used to build server and desktop applications.
- **Flutter**: It is Google's portable UI toolkit for crafting beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source.
- **Framework**: It is an abstraction in which software, providing generic functionality, can be selectively changed by additional user-written code, thus providing applications-specific software.
- **RESTful**: It is a software architecture style that defines a set of constraints for creating Web services
- **Tier**: It is a row or layer in a series of similarly arranged objects. In computer programming, the parts of a program can be distributed among several tiers, each located in a different computer in a network, each with its computational power.

1.4.1 Acronyms

- **API**: Application Programming Interface, it indicates an on-demand procedure that supplies a specific task.
- **BLoC**: Business Logic Component.
- **DB**: DataBase.
- **DBMS**: DataBase Management System.
- **DTO**: Data Transfer Object.

- **IdP**: Identity Provider.
- **REST**: Representational state transfer.
- **S2B**: Software to Be, is the one designed in this document and has not yet been implemented.
- **SDK**: Software Development Kit: It provides a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allow developers to create software applications on a specific platform.
- **UI**: User Interface.
- **UX**: User Experience.

1.4.2 Abbreviations

- **Rn**: requirement number n.
- **An**: assumption number n.

1.5 Document Structure

- **Section 1 : Introduction**
  This section offers a brief description of the document, including the definitions, acronyms and abbreviations that will be found while reading it.
- **Section 2 : Architectural Design**
  This section is addressed to the developer team and offers a more detailed description of the architecture of the system. The first part describes the chosen paradigm and how the system will be organized. Furthermore, a high-level description of the system is provided, together with a presentation of the modules composing its nodes. Finally, there is a concrete description of the tiers forming the S2B.
- **Section 3 : User Interface Design**
  This section is useful for graphical designers of the S2B and contains several screens of the application.

- **Section 4 : Requirements**
  This section contains the main requirements that the S2B must satisfy, together with a brief description of them.

- **Section 5: Implementation, Integration and Testing**
  The last section is again addressed to the developer team and describes the procedures to follow for implementing, testing and integrating the components of the S2B. There will be a detailed description of the core functionalities of it, together with a complete report about how to implement and test them.

2 Architectural Design

2.1 Overview

QuizTime is a distributed application that adheres to the widely recognized client-server paradigm. This architecture consists of two main components: the client and the server.

The client, represented by the QuizTime mobile application, serves as the primary interface for users to access and utilize all system functionalities. Through the mobile app, users can interact with the server and perform various quiz-related activities.

The server, implemented using Firebase, handles user authentication, ensuring secure access to the QuizTime application. Additionally, the server manages data storage and retrieval: it leverages the capabilities of Firestore, a Firebase database service. Firestore allows for efficient and real-time management of quiz data, ensuring a smooth user experience and synchronized updates across devices.

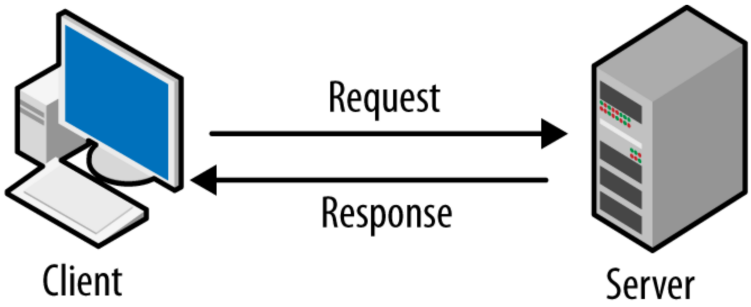

Figure 1: Client-server paradigm

The application is divided into three main layers:

- **Presentation layer**: It is the user interface of the application, through which the end-user interacts with it. Its main purpose is to collect and display information from the user.

- **Logic layer**: It processes the information collected from the Presentation layer using the business logic. The logic layer is entitled to add, delete or modify data in the data layer.
- **Data layer**: It is where the information processed by the application is stored and managed.

There is a clear separation of these three layers on different tiers to enhance performance and maintainability.
Specifically, it adopts a two-tier architecture, consisting in a thick client, where the first two layers described are located, and a server.
In this architecture, the business logic is primarily implemented within the client application. Data storage and exchange operations are managed on the server side.

The application, therefore, takes the form of a thick client which relies on the server to store and retrieve data. As a result, an internet connection is necessary to use the application.

The core of the software, encompassing the Presentation and Logic layer, resides on the client side. On the other hand, the Data Layer finds its place on the server tier, utilizing Firebase Server to handle data storage and retrieval operations.

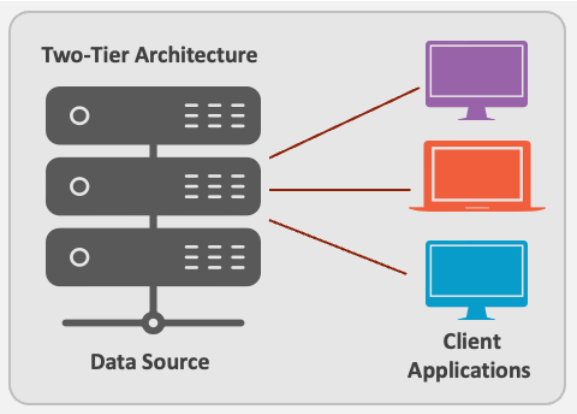

Figure 2: Two tiers application

2.2 RESTful architecture

The RESTful architecture is based on the stateless principle, in which the server does not contain any information about the state of the client, which is managed directly on the client side.

This behaviour guarantees less computational load on the server and also a dynamic attitude of the services.

The application is intended to be developed through client-side programming, which means that all the requests and updates of the page are made on the client side.

3 User Interface Design

The purpose of this section is to present the design of the main screens of the mobile application, highlighting the flow of the primary functionalities it serves. The flow is tailored to the specific inputs and preferences of the user.

The primary objective is to create an intuitive and user-friendly design, ensuring accessibility and ease of use for users of diverse age groups, making it universally approachable.

It is important to note that the user experience will be slightly adapted for both tablet and smartphone devices, aiming to optimize the user's interaction and overall experience.

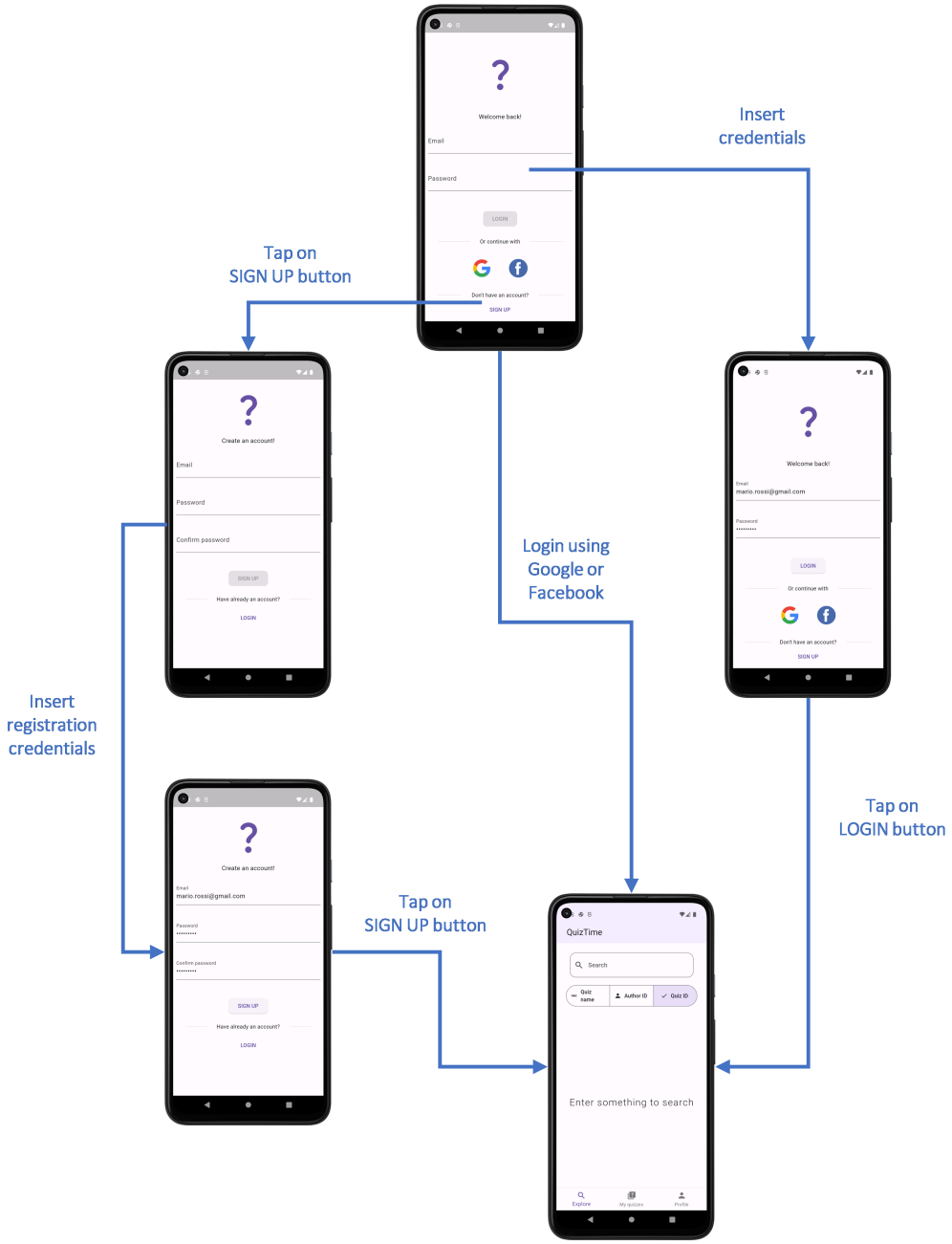

Figure 3: Login and sign up

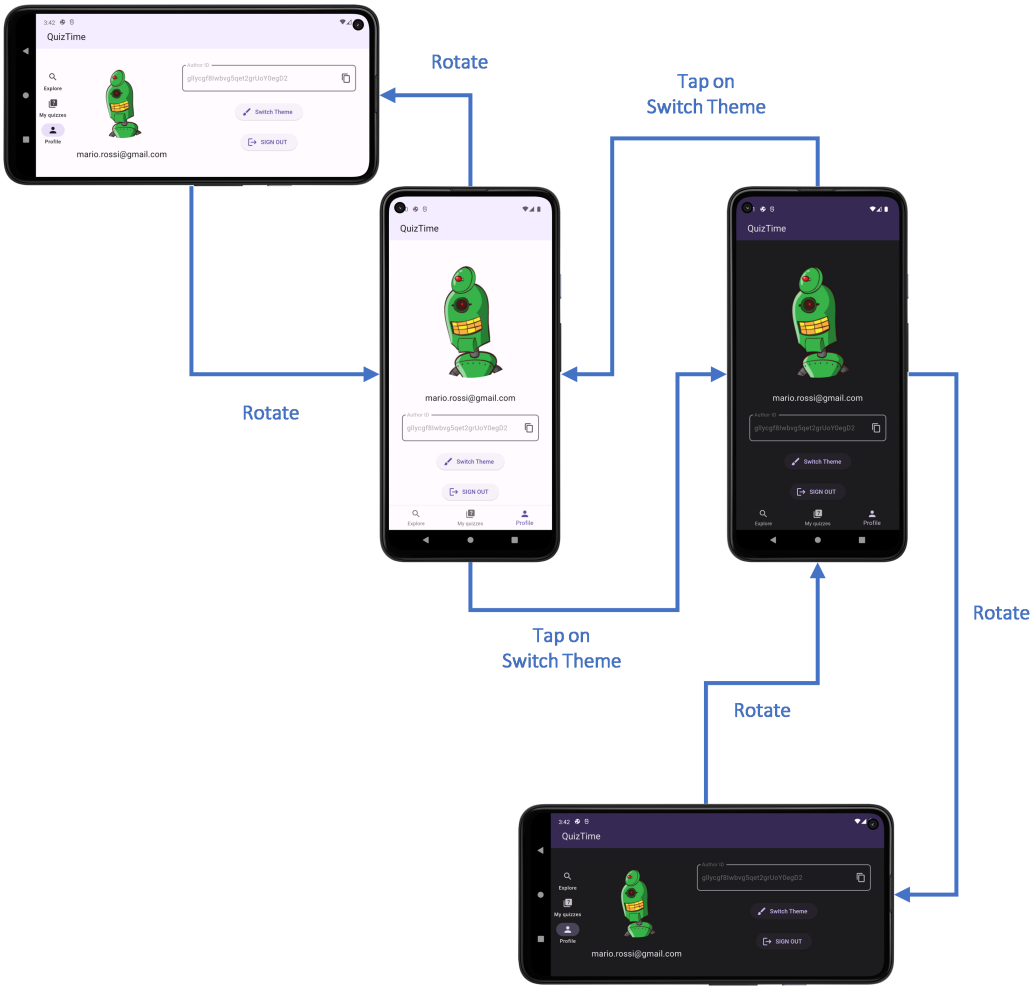

Figure 4: Profile page: light theme vs. dark theme

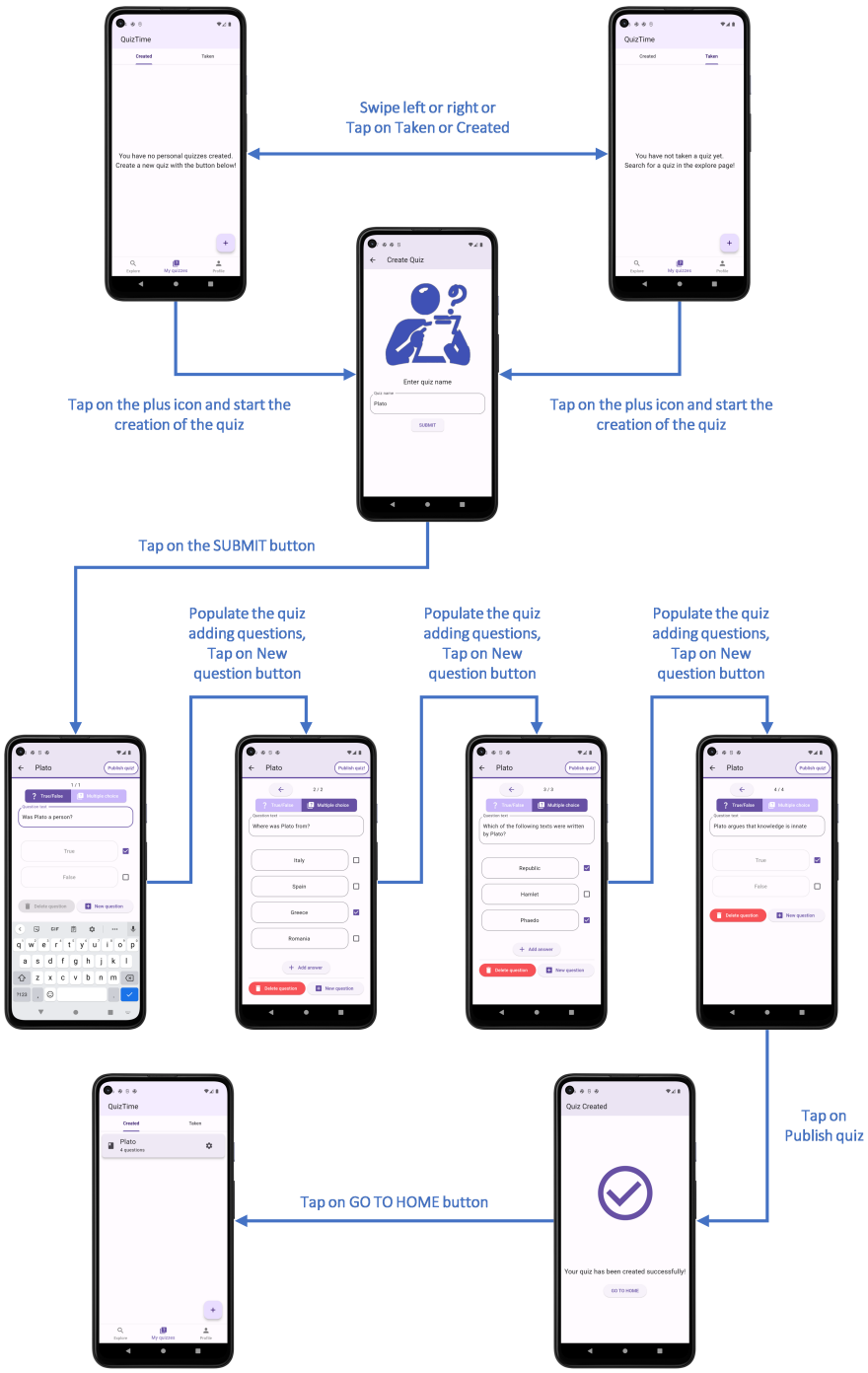

Figure 5: Quiz creation

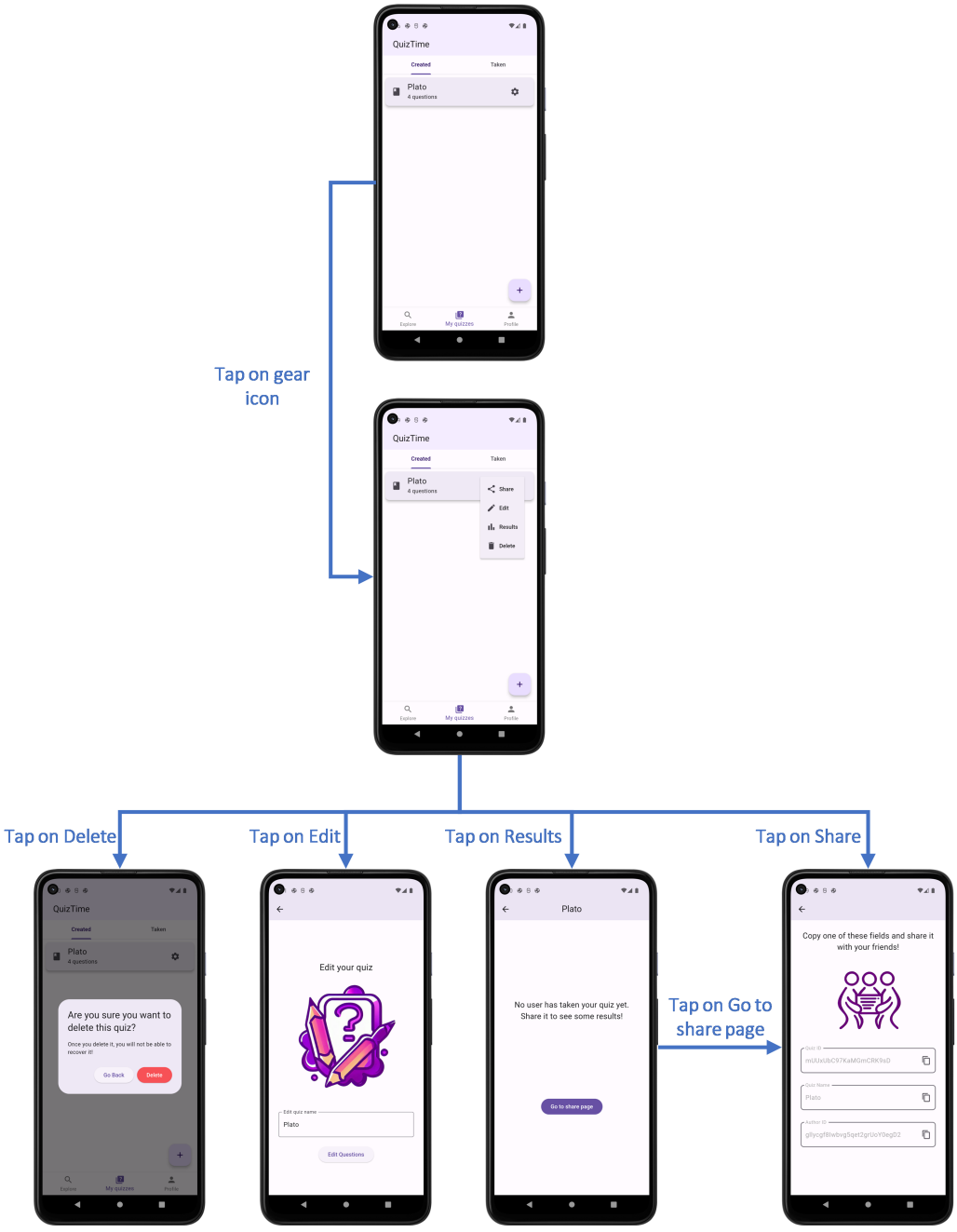

Figure 6: Options for managing created quiz

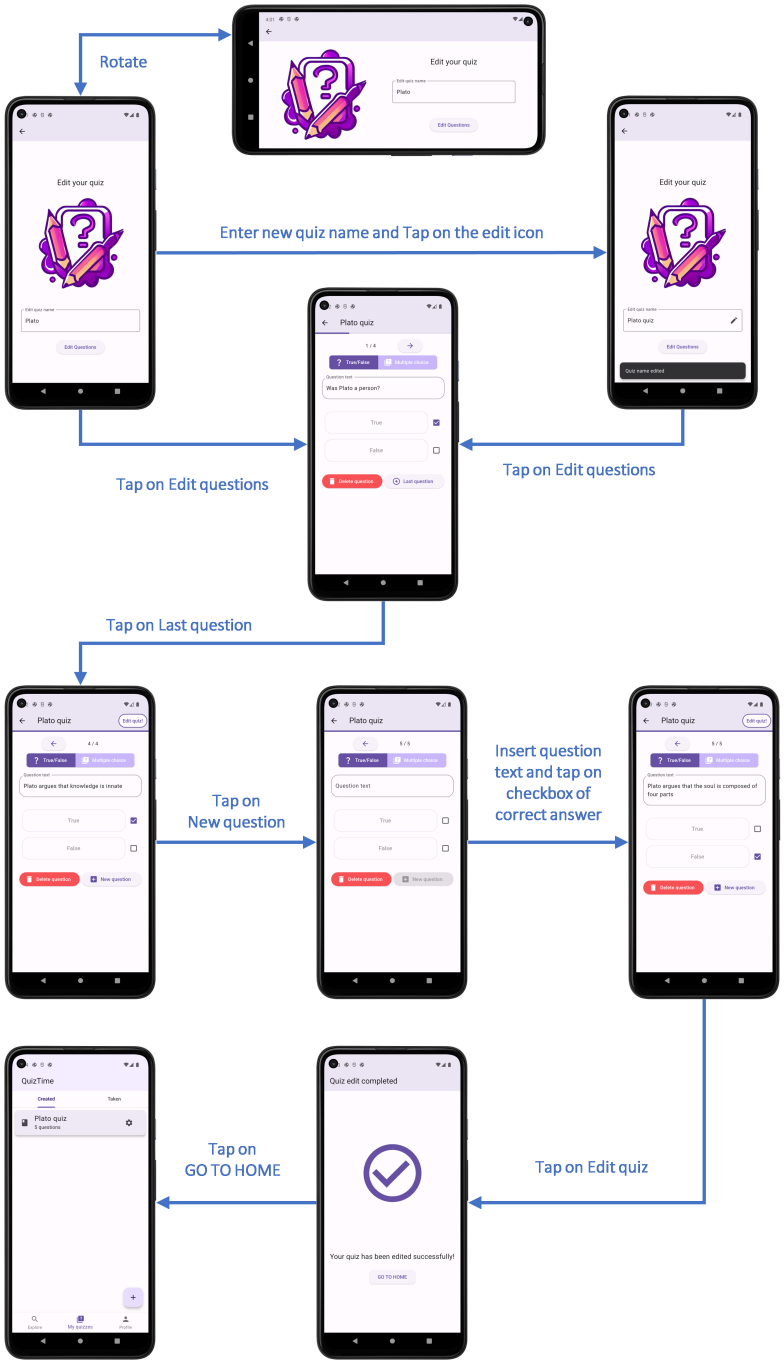

Figure 7: Edit a quiz

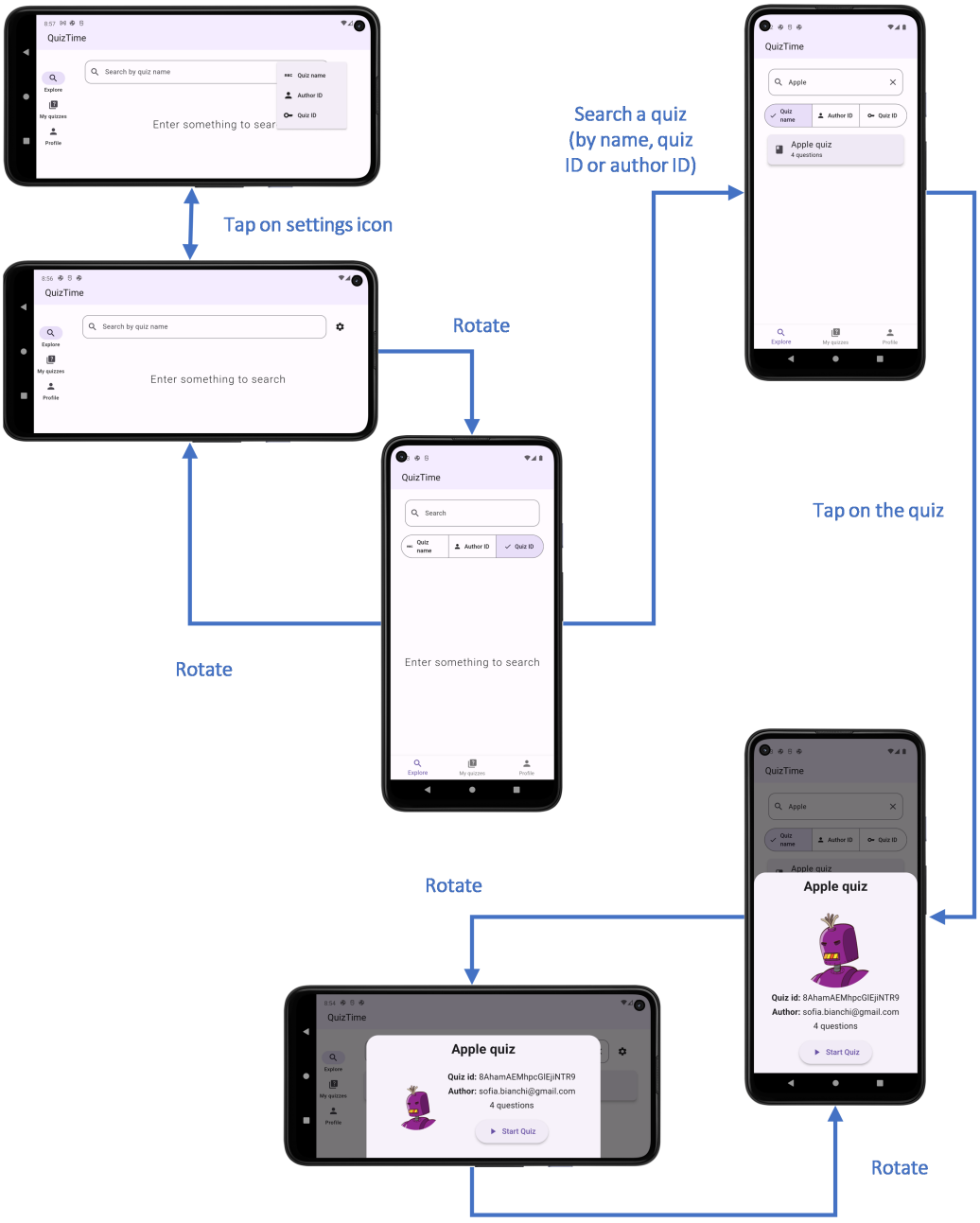

Figure 8: Search for a quiz

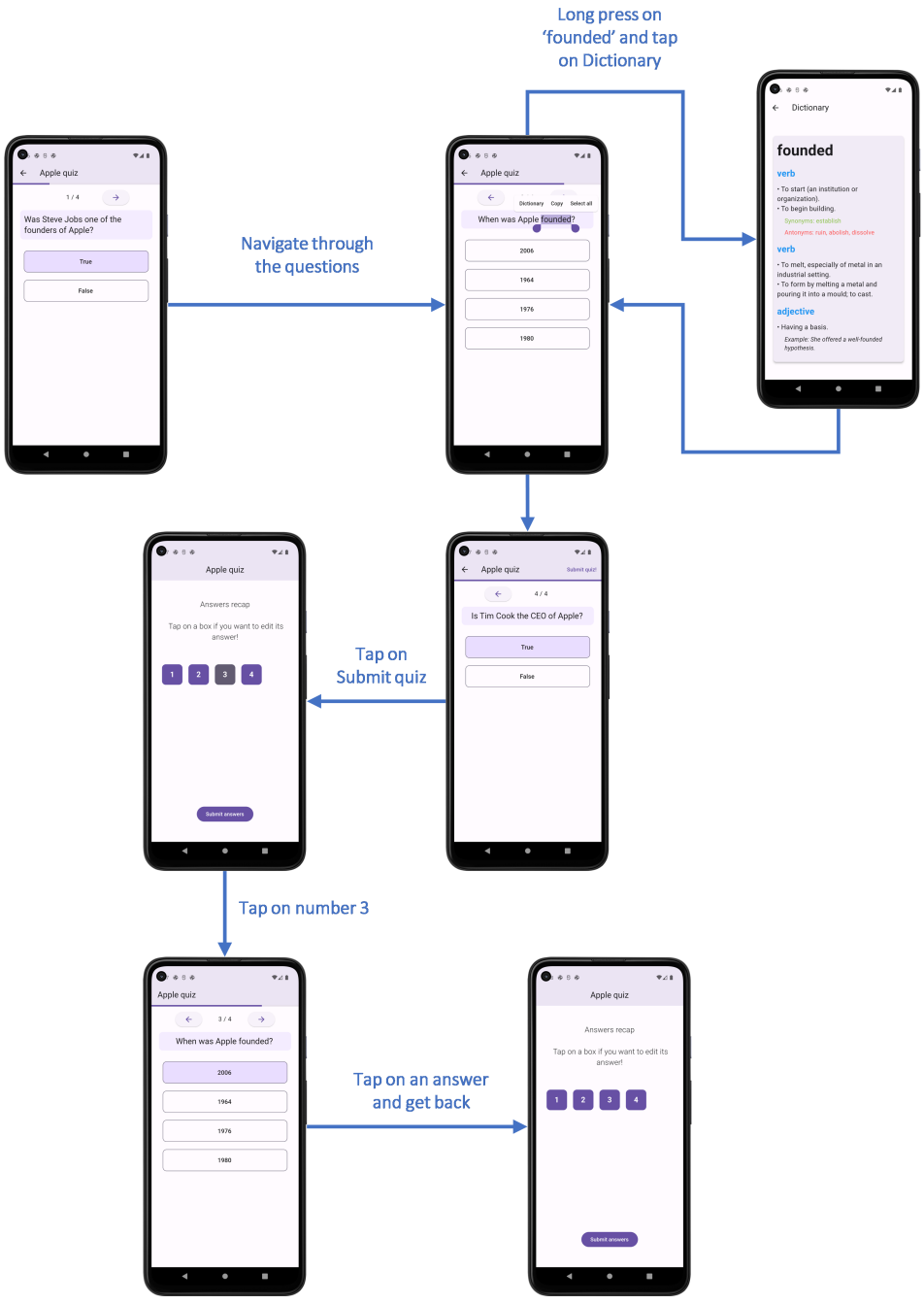

Figure 9: Take a quiz

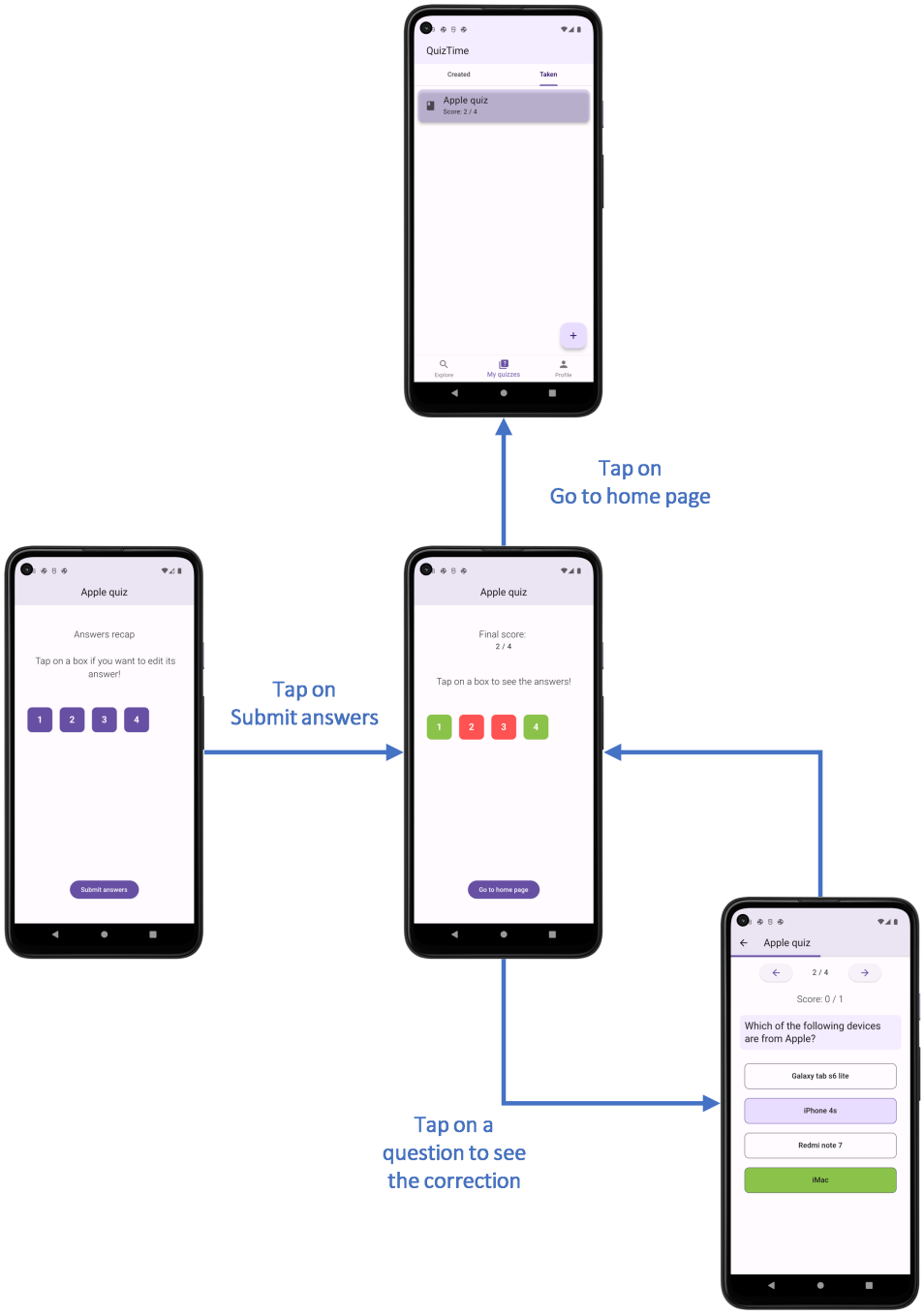

Figure 10: Submit taken quiz and display results

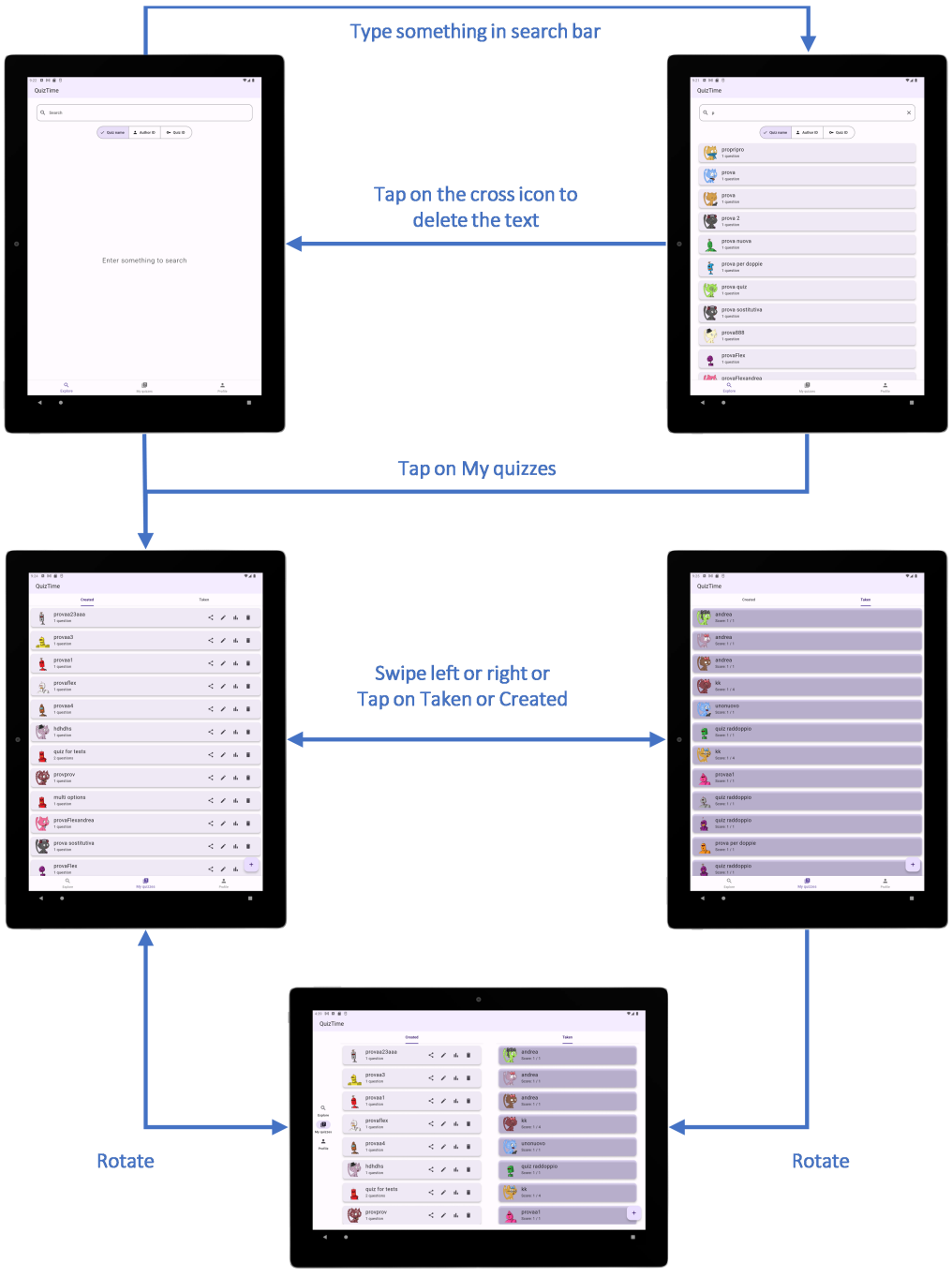

Figure 11: Tablet UI tailoring

3.1 Mobile phone Interface

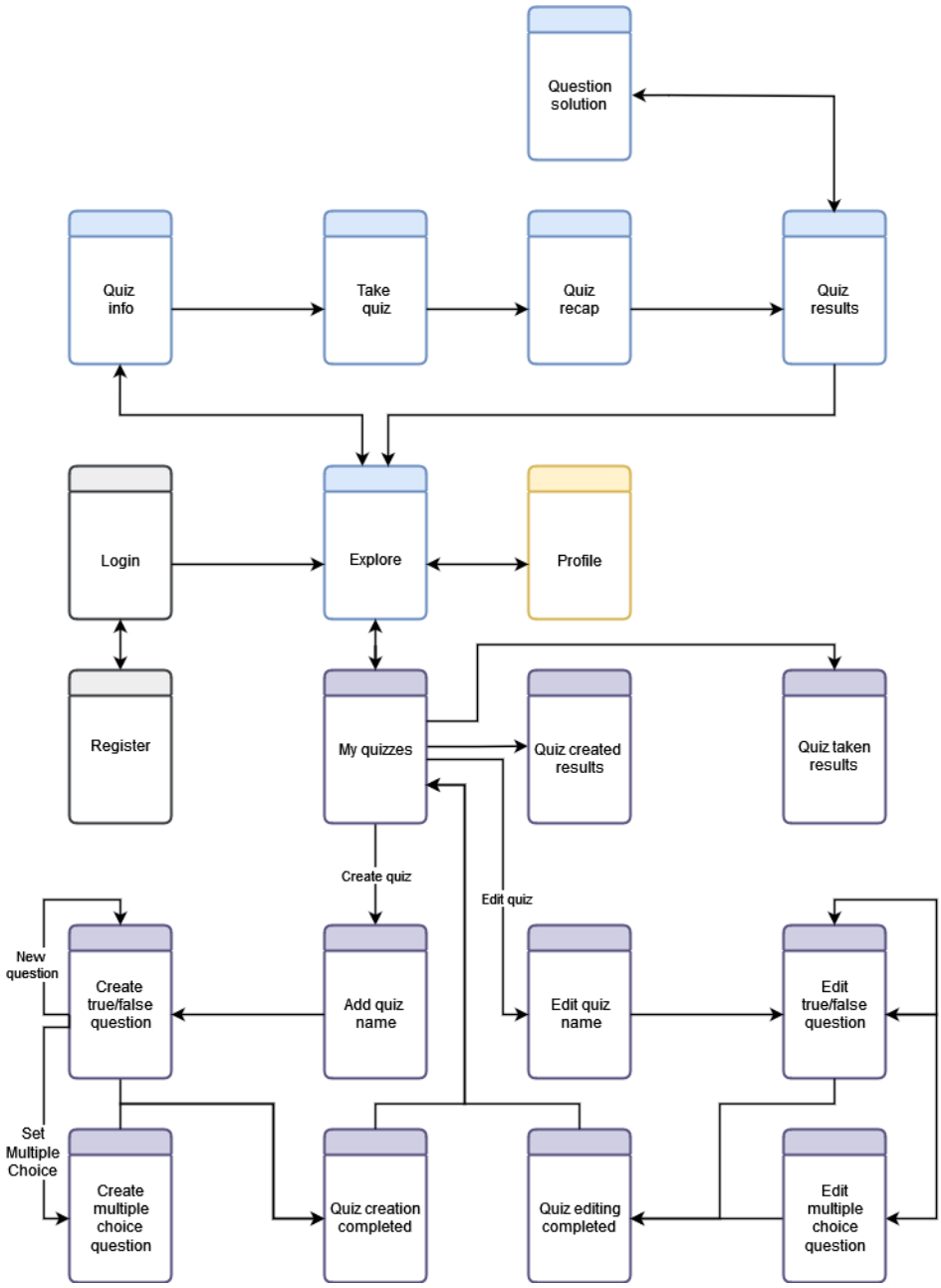

Figure 12: Mobile phone UX diagram

3.2 Tablet Interface

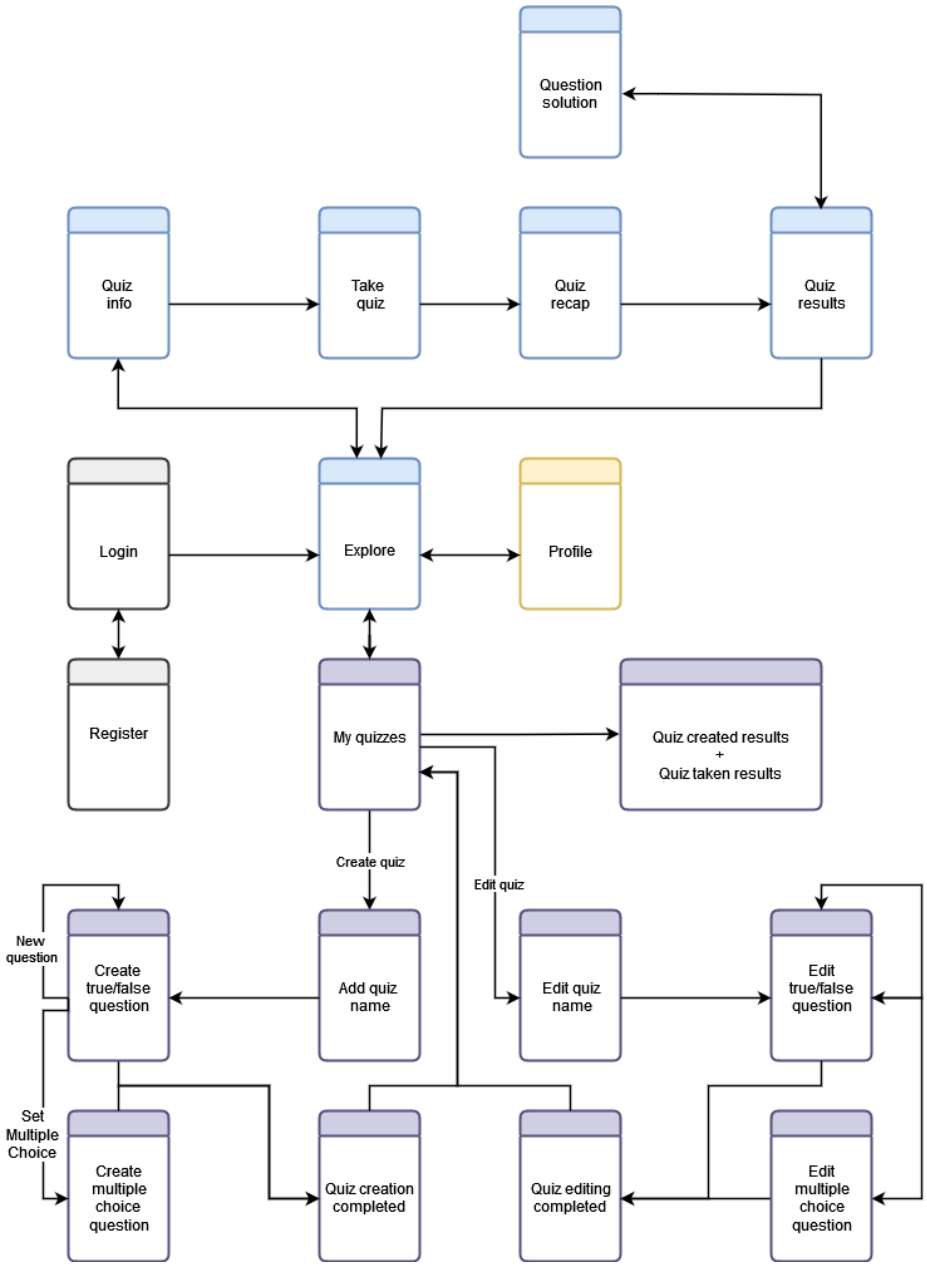

Figure 13: Tablet UX diagram

4 Requirements

The following list contains the requirements that the application should satisfy, together with a brief explanation of them.

R1: The system allows the login/registration of a user. This can be done with the methods specified in section 5.4.2.

R2: The system allows users to create a new quiz, which consists of a series of questions.

R3: The system allows users to create a question either of type true/false or multiple choice.

R4: The system allows users to edit and update their quizzes after creation.

R5: The system allows users to delete a previously created quiz.

R6: The system allows users to share their quizzes.

R7: The system allows users to search for a quiz by name.

R8: The system allows users to search for a quiz by quiz ID.

R9: The system allows users to search for a quiz by author ID.

R10: The system allows users to take a quiz.

R11: The system allows users to search for a word in a dictionary, through an API, while taking a quiz.

R12: The system allows users to see the results of a previously taken quiz.

5 Implementation, Integration and Testing

The S2B will be divided as follows:

- **Application Server** (which contains the database)
- **Client**
- **External Services**

5.1 Implementation assumptions

In order to properly implement the S2B, there are different assumptions to be made:

A1: The users have access to compatible mobile devices with internet connectivity in order to use the app.

A2: The users have basic knowledge of using mobile applications and interacting with intuitive interfaces.

A3: The creators are responsible for the content they create and share on the app, ensuring it adheres to legal regulations.

A4: The implemented API for searching for a word on a dictionary works mainly with English terms.

A5: The app is primarily developed in English, but there is a potential for future support for other languages.

5.2 Application Server

As mentioned earlier, the server side of the application will be implemented using Firebase due to its comprehensive tools and services that simplify backend development and maintenance.
Firebase platform provides a range of ready-to-use services that make it easier for developers to build and manage the app's backend. These services simplify the effort needed to set up and maintain the server. Here are the selected services for the application, along with brief descriptions for each:

- **Database**: as database, we will choose Cloud Firestore. Cloud Firestore is a NoSQL database that organizes data into collections and documents. It enables efficient indexed queries with compound sorting and filtering capabilities.
- **Firebase Auth**: In order to ensure secure and user-friendly authentication, we've integrated Firebase Auth. This service supports multiple platforms and can seamlessly work with other Firebase services, providing our users with a smooth and consistent experience.

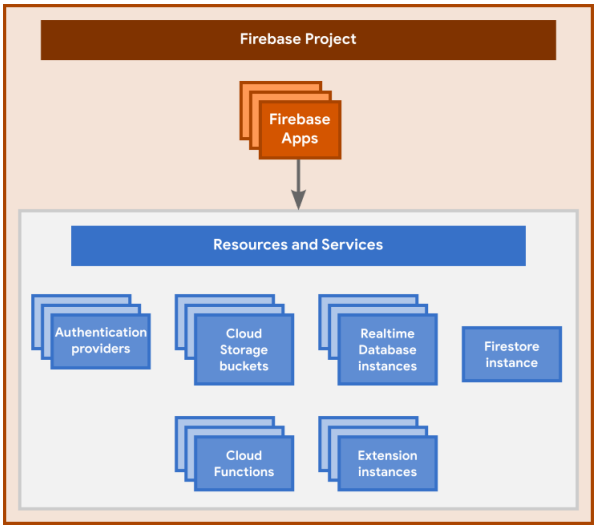

Figure 14: Firebase main resources

5.3 Client

5.3.1 Distribution environment

The client application will be distributed on the end user's mobile phone or tablet.
The supported operating systems will be Android and iOS and the application will be published on the respective stores, *Google Play* and *App Store*.

5.3.2 Framework and programming language

We will use Google's Flutter as our mobile app framework, whose architecture is described in Figure 15.

Flutter is an open-source UI software development kit created by Google. It is used to develop cross-platform applications for Android, iOS, Linux, Mac, Windows and Web platforms from a single codebase.

Flutter apps are written in the Dart language and make use of many of the language's more advanced features. Flutter's engine, written primarily in C++, provides low-level rendering support using Google's Skia graphics library.
Additionally, it interfaces with platform-specific SDKs such as those provided by Android (Kotlin) and iOS (Swift/Objective C).

The Flutter framework contains two sets of widgets that conform to specific design languages:

- **Material widgets**: they implement Google's design language of the same name.
- **Cupertino widgets**: They implement Apple's iOS Human interface guidelines.

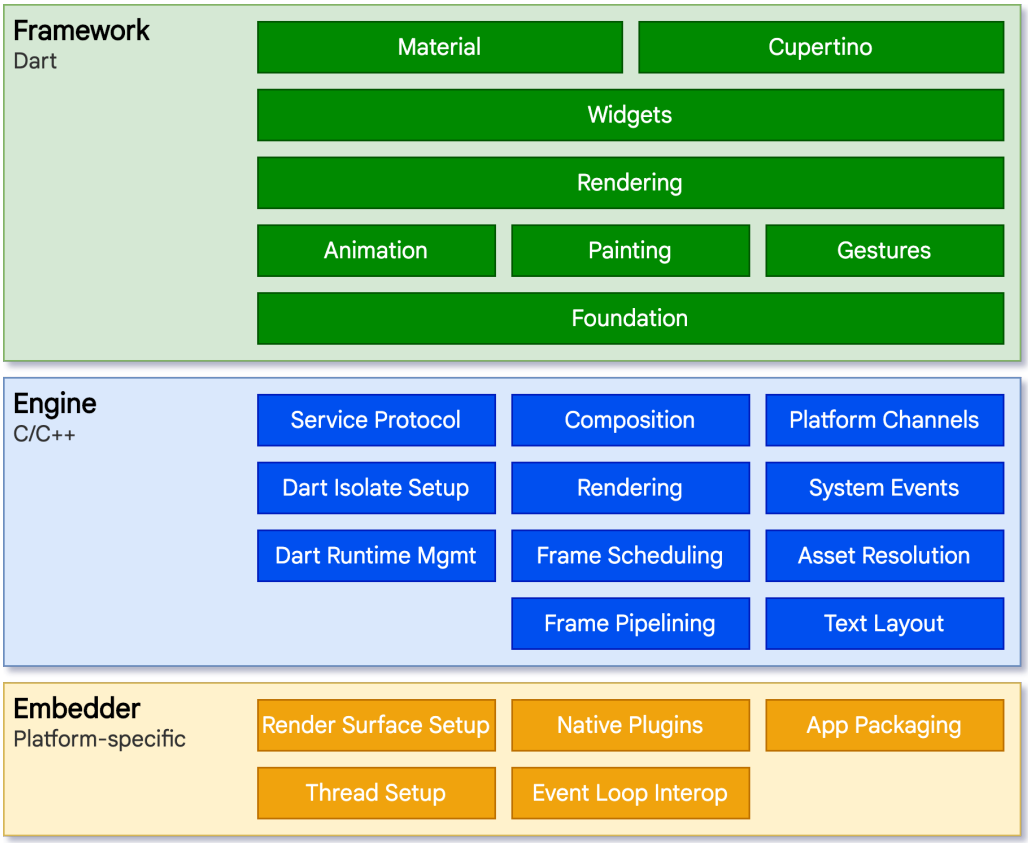

Figure 15: Flutter Architecture

5.3.3 Flutter project architecture

The architecture of the Flutter project is designed to achieve exceptional modularity, maintainability, and high separation of concerns of different components. *The primary goal is to facilitate straightforward and comprehensive testing for each component within the application.*

By adopting this architecture, the project embraces a clear separation of concerns, allowing developers to focus on individual components without being entangled in unnecessary complexities. This modularity ensures that changes or updates to one component do not adversely impact others, promoting a streamlined development process and facilitating bug-free code.

Furthermore, the well-defined boundaries between components make testing remarkably efficient. Each component can be isolated for testing, enabling a complete examination of its functionality in a controlled environment. This approach enhances the reliability and quality of the application, as potential issues can be identified and addressed promptly, leading to a robust final product.

Our architecture has components that play various roles:

- **BLoC**: BLoC is a design pattern developed by Felix Angelov and recognized by Flutter as one possible implementation for state management. This pattern helps to separate presentation from business logic. The base element is the Bloc: an advanced class that relies on events to trigger state changes rather than functions. Blocs receive events and convert incoming events into outgoing states.

- **DTO**: The communication between the server and the user application uses DTOs so that if the server API changes, the client has to update only the DTOs and not the model.

- **Mapper**: A mapper is a component that performs the task of converting Data Transfer Objects (DTOs) into model objects and vice versa. It facilitates bidirectional data transformation, enabling seamless communication and data handling within the application.

- **Repository**: A repository serves as a single source of truth, communicating with services to fetch data. It uses mappers to convert DTOs into app-readable models and provides this data to upper layers like BLoCs for business logic and UI updates.

- **Provider/Service**: A self-contained, independent component that provides specific functionalities or interacts with external resources. These services are designed to perform well-defined tasks and provide capabilities to other components within the software system. A service abstracts the complexities of communication with external resources. By encapsulating the logic required to interact with external APIs or databases, they simplify the development process and improve the maintainability of the software.

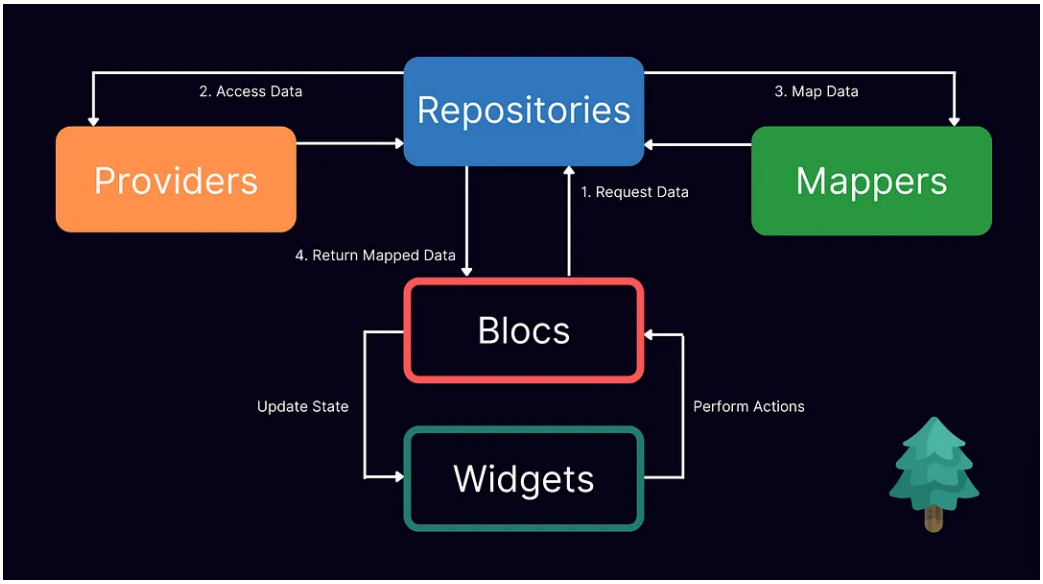

Figure 16: Typical interaction of the application

In general, the paradigm of the application works as follows:

1. When the user interacts with a **Widget**, the corresponding Bloc is notified.
2. The **Bloc** sends a data request to a Repository.
3. The **Repository** translates the request and forwards it to a Provider responsible for accessing the data source.
4. The **Provider** retrieves the requested data and returns it to the Repository.
5. The Repository then sends the data to a Mapper to convert it into the desired format.
6. The **Mapper** returns the mapped data to the Repository
7. The Repository returns the converted data to the BLoC.
8. Finally, the BLoC updates its state, triggering a notification to the Widget, which updates itself based on the new state.

5.3.4 Component diagram

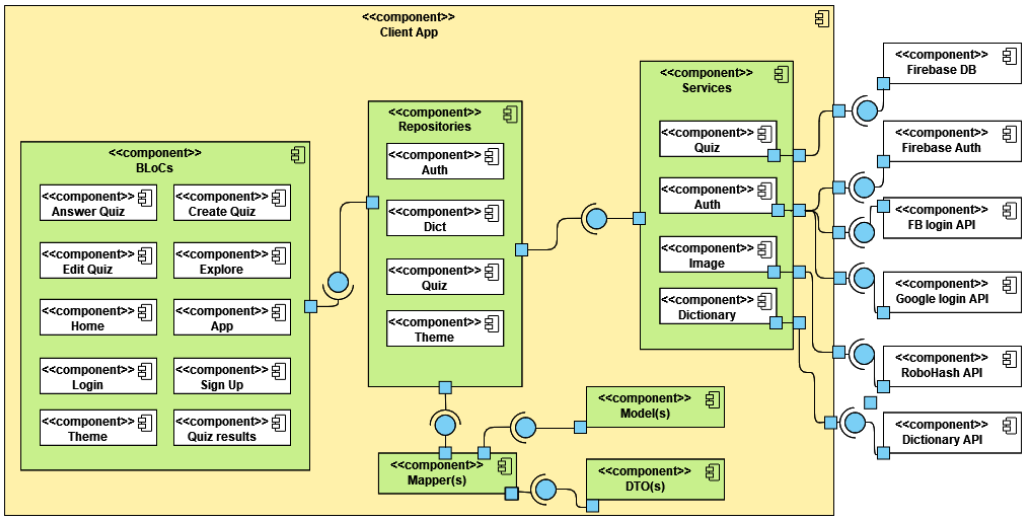

Figure 17: Component diagram of the software components of the application

5.3.5 Libraries

In our flutter artefact, we will use several libraries, including:

- `http`: Provides primitives to make HTTP requests.
- `animations`: Facilitates creating and managing animations in Flutter applications.
- `firebase_core`: The core package required for setting up Firebase in a Flutter project.
- `firebase_auth`: Enables Firebase Authentication.
- `google_sign_in`: Enables Google Sign-In functionality in Flutter applications.
- `flutter_facebook_auth`: Enables Facebook Authentication.
- `equatable`: Simplifies object comparison and hash code implementation, useful for immutable classes.

- `flutter_bloc`: Implements the BLoC (Business Logic Component) pattern for state management.
- `bloc`: Provides additional functionalities and utilities to handle app state using the BLoC pattern.
- `mocktail`: Offers utilities for creating and working with mock objects during testing.
- `cached_network_image`: Optimizes the loading and caching of network images.
- `build_runner`: Supports code generation in Flutter projects.
- `json_annotation`: Facilitates JSON serialization and deserialization in Dart classes using annotations.
- `cloud_firestore`: Provides access to Cloud Firestore, Firebase's cloud-based NoSQL database.
- `provider`: Allows easy and efficient data sharing between widgets using the Provider pattern.
- `very_good_analysis`: Provides static analysis tools to improve code quality and consistency in Flutter projects.
- `fake_cloud_firestore`: fakes to write unit and widget tests for Cloud Firestore.

5.3.6 Support for tablets

QuizTime will provide a slightly different user interface designed for **tablets** and **iPads**.
In fact, the organization of contents in the UI must be adapted to a wide screen with a 16:9 resolution (instead of the 9:16 of standard smartphones). This job will be done by using Flutter's MediaQuery, which allows to specify conditions based on screen size and device orientation (i.e. portrait vs. landscape).

5.4 External Services

5.4.1 APIs for images

RoboHash[1] is an API used to generate funny images of different types: robots, monsters, cats etc.
The RoboHash API generates avatar images based on text strings. A text input is sent to the API, and it uses algorithms to convert the text into a visually distinct avatar. The API returns the avatar image.
These images will be used as profile picture for those profiles that do not already have an image set and for quizzes.

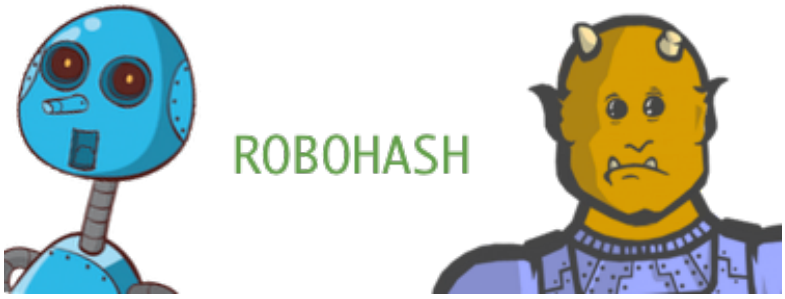

Figure 18: RoboHash logo

5.4.2 APIs for IdP

- **Email Auth:** The application utilizes the Google Flutter SDK to make an API request to the Firebase authentication provider. The authentication handler responds with a token, which is included as an Authorization header in each subsequent HTTP request. The Firebase SDK manages token expiration automatically, obtaining a new token when necessary.
- **Facebook sign-in :** For Facebook sign-in functionality, the application interacts with the Facebook API to retrieve user data. The retrieved data is then passed to Firebase for registering a new user.
- **Google sign in:** the Google API fetches user data from Google. The obtained data is then forwarded to Firebase to register a new user.

[1]https://robohash.org/

5.4.3 API for dictionary

Free Dictionary API[2] is a convenient service that provides access to an English dictionary's data through a simple request. It returns the requested information in JSON format, including phonetics, meanings, synonyms and antonyms.

5.5 Testing

The testing campaign will adhere to the Flutter guidelines (*link*). To ensure the reliability and quality of the application, we will conduct three types of testing, each serving specific purposes:

- **Unit Testing:** It ensures that the end-user can achieve the goals set in the application requirements, which determines whether the software is acceptable for delivery or not.
- **Widget Testing:** The goal of a widget test is to verify that the widget's UI looks and interacts as expected. Testing a widget involves multiple classes and requires a test environment that provides the appropriate widget life cycle context.
- **Integration Testing:** It ensures that an entire, integrated system meets a set of requirements. It is performed in an integrated hardware and software environment to ensure that the entire system functions properly.

5.5.1 Unit Testing

The primary goal of unit testing is to adequately cover the entire business logic section of the application. As the application involves a small model for proper deserialisation of data obtained from the back-end, unit tests will be essential to validate interactions and data handling within this model. We will provide mocked responses to each class of the model and perform multiple assertions to ensure that data is correctly processed and handled.

[2]https://dictionaryapi.dev/

5.5.2 Widget Testing

The focus of widget testing is to ensure that widgets in the application's user interface are properly loaded and populated with data. By feeding widgets with specific data and performing appropriate assertions, we will verify that the user interface appears and interacts as expected. During the widget testing phase, we will provide back-end mocks to generate fictitious data for rendering and validation. However, it is important to note that some components may belong to external packages that are already exhaustively tested, which may limit the scope of widget testing for those specific components.

5.5.3 Integration Testing

Integration testing expands on the widget and unit testing phases, focusing on how different widgets interact with each other as part of a coherent and integrated system. Unlike widget testing, where individual widgets are validated, integration testing examines the flow of interactions between various widgets and components. We will start integration testing from a given page and proceed through a series of taps, text insertions, and gestures to validate the execution flow and the interactions between widgets. This comprehensive testing approach will help us identify and resolve any potential issues that may arise when multiple widgets collaborate within the application.

Testing goal

The goal of the testing campaign is to detect as many bugs as possible during the development phase of the mobile application. This approach is also helpful since it allows us to promptly validate the code during its writing. This strategy will raise the quality of the final artefact.

Our primary goal is to achieve an extensive code coverage, of approximately 90%. This level of coverage will give us a high level of confidence that we have exhaustively tested the application, significantly reducing the likelihood of encountering bugs once it is released.