BABEŞ BOLYAI UNIVERSITY, CLUJ NAPOCA, ROMÂNIA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

The "Quizzicat" Application

– Diploma thesis –

Coordinator
Lect. Dr. Suciu Dan Mircea

Author
Ciocea Christine-Gerlinde

2020

UNIVERSITATEA BABEŞ BOLYAI, CLUJ NAPOCA, ROMÂNIA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Aplicaţia "Quizzicat"

– Lucrare de licenţă –

Coordonator ştiinţific
Lect. Dr. Suciu Dan Mircea

Absolvent
Ciocea Christine-Gerlinde

2020

Abstract

The purpose of this thesis is to present and document the creation of a mobile application for playing trivia games, using modern mobile development tools and intelligent algorithms for user authentication, topic recommendations and data storage and retrieval.

The content of the thesis is structured into seven chapters.

The first one, "Introduction", contains an overview of the subject matter, presenting the motivation behind this body of work, as well as the way it was approached, as well as briefly going over some related applications.

The second chapter, titled "Related Work", goes into more details regarding the similarities and differences between the "Quizzicat" application and other existing mobile trivia games.

The third one, "Prerequisites of Android Mobile Development", explains the basic elements of modern Android mobile development, going over the basic structure of an Android application and presenting a short overview of the Kotlin programming language.

The fourth chapter, titled "Design Patterns", contains some theoretical aspects regarding design patterns, the pros and cons of using them and a brief explaination of some of the most well known ones.

The fifth chapter, "Application Design", contains the use case, database and sequence diagrams for the application, as well as details regarding how design patterns have been integrated into the application architecture and how external libraries are used to support different functionalities.

The sixth one, "Implementation" contains a walk-through of each main screen of the application, going over the program flow and the most relevant implementation details.

The last chapter, "Conclusions and Future Work" presents the overall impressions regarding the development process and notes some possible enhancements that could be added to the application in the future.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Motivation

It is not an exaggeration to state that mobile phones, particularly smartphones, have become an essential gadget in the day to day life of the modern human. This can easily be proven by analyzing the ownership of smartphones over the past few years - according to surveys conducted by the Pew Research Center between the years 2011 and 2019, the percentage of adult U.S. citizens that own a smartphone has increased from 35% in the starting year to 81% in 2019. [1]

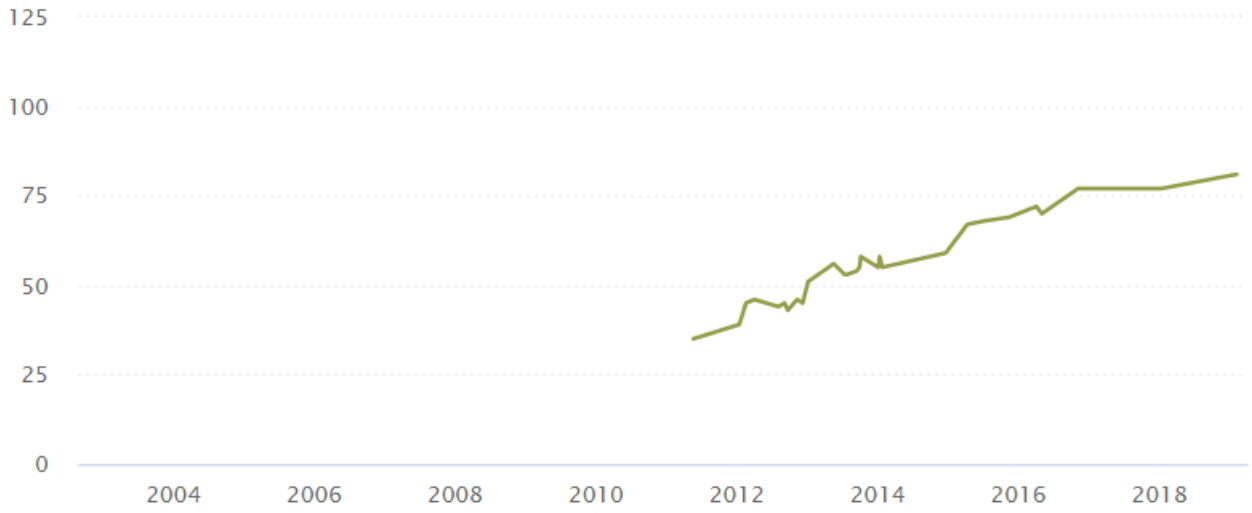

Figure 1.1: The evolution of the percentage of U.S. adults who use smartphone devices. [1]

As the number of smartphone users increases, so too does their demand to be entertained via different applications and mobile games, giving developers ample opportunities to create wildly successful applications. An example of one such mobile game is "Flappy Bird", a simplistic, yet highly addictive game which took the world by storm in 2014, and ended up generating millions of dollars in ad revenue

for its developer, Dong Nguyen, even after it was removed from Google Play and Apple's App Store. [43]

The modern human also seems to be drawn by interactivity, especially when it comes to proving their knowledge via trivia quizzes. Currently, the world's fastest-growing learning platform is Kahoot.com, an application which allows users to create, share and play trivia quizzes with other users in real-time. The application has reached a peak of 70 million unique users since its launch in 2013 and is growing at a rate of 75% yearly [20]

Considering the information presented above, it is clear that there exists a market for mobile trivia games. An interactive application with a user friendly interface that focuses not only on meeting the expectations of users already accustomed to other trivia game platforms, but also on adding unique features that set the application apart from its competitors can become extremely financially successful through the size of the user base and through the monetization of certain features.

1.2 Basic Approach

The goal of the Android mobile application Quizzicat is to be easy for the average player to access and use, and entertaining enough for the user to play trivia games on it multiple times per day.

The approach regarding the design of the interface was to keep the aspect minimalist, but still interesting to look at, using a bright, distinctive colour palette and various icons.

Similarly, the game play is straightforward and easy to understand for first time players, and the multitude of game modes (the user can play trivia games on their own or with a custom number other users), the tailored recommendations and the fact that users can submit their own questions are designed to keep the user involved in the game play for as long as possible.

1.3 Related Work

The application is similar to other trivia games that can be found on both Google Play and Apple's App Store, such as QuizUp, Kahoot! and Trivia Crack in its nature and content. Similarly to these applications, Quizzicat allows the user to create a new account using either their e-mail address or other social media accounts, to keep track of the number of correctly answered questions and contains various topics that the user can choose from when they want to play a quiz. Similarly to Kahoot!, Quizzicat allows contains a multiplayer functionality, allowing real-time trivia games between multiple users.

1.4 Original Contributions

The research and implementation details presented in this paper advances the design and implementation of several already existing trivia games.

Firstly, the Quizzicat application provides the user with a single player mode, in which they can configure the number and difficulty of the questions, and whether or not the quiz should be timed or not.

Secondly, the multiplayer mode also different in its execution - the user that creates does not have to set up the questions themselves, and thus can participate in the game, rather than only acting as a moderator.

Lastly, the application provides all users with a way to add their own questions to a topic from the game, and the acceptance of the custom prompt into the actual game is done automatically when the question is popular enough among the players, rather than manually by a team of moderators. Each player can rate pending questions, and questions with a low enough rating are automatically removed, thus making sure unpopular or wrong questions are not added to the roster.

1.5 Paper Structure

The present work contains 48 bibliographical references and is structured in six chapters as follows:

The first chapter represents an introduction into the subject matter of the thesis, briefly going over the reasons for choosing the topic of the application, how the design of the application was approached, the already existing trivia applications on the market and how Quizzicat differs from them.

The second chapter takes a deeper look into the would-be main competitors on the mobile trivia application market, presenting each application before giving a detailed comparison between the functionalities of each, in comparison to Quizzicat.

The third chapter goes over the most important aspects that are involved in developing a modern Android application, using Kotlin and Android Studio.

The fourth chapter gives a presentation of Design Patterns as a whole, going over what a design pattern is, why it is used in the industry and going over the categories of design patterns.

The fifth chapter lays out all the details regarding the design and architecture of the application, as well as the details regarding all the 3rd party libraries that were used to implement or enhance certain functionalities.

The sixth chapter takes each functionality of the application and explains how the corresponding behaviour has been implemented, starting from the Android Activity which launches the operation.

The seventh chapter contains a brief conclusion regarding the development process of the application, along with some additional changes that could be implemented in the future.

Chapter 2

Related work

The purpose of this chapter is to present the main functionalities some already existing trivia game applications and the way they compare with those provided by the "Quizzicat" application, highlighting the similarities and differences between them.

2.1 Existing Trivia Games Applications on the Market

2.1.1 QuizUp

Quizup is a mobile game, originally released for the iOS platform on the 6th of March 2013 and later released for Android on the 7th of November 2014. The game was originally developed by the Icelandic publisher Plain Vanilla Games, but has since been purchased by Glu Mobile. [41]

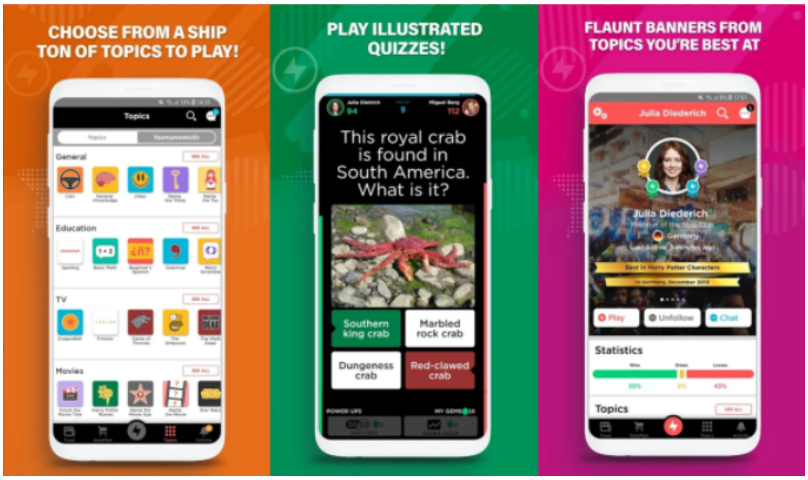

Figure 2.1: The different QuizUp interfaces [13]

At its core, it is a multiplayer game, in which the user competes against either a randomly chosen user of the same level, or against a user from their Friends list, during seven rounds of multiple-choice

questions of various topics, each round lasting ten seconds and being worth 20 points. The score earned decreases the longer it takes users to answer. Incorrect answers lead to no points being awarded. There also exists a bonus round after the first six questions, in which the fastest user to answer correctly can double their points, and thus possibly turn the tide of the entire match. The maximum number of points possible to be scored in a single game is 160. [48]

The total number of topics exceeds 1.200 and most of them are available in different languages, other than English. All the questions are submitted by different contributors, who are chosen by the moderating team. [46]

2.1.2 Kahoot!

Kahoot! is a game-based social learning mobile and web application, launched in 2013 by Johan Brand, Jamie Brooker and Morten Versvik in a joint project with the Norwegian University of Science and Technology. While its main purpose is to serve as an educational tool, users can create, share and play various trivia games with other online users as well. [17]

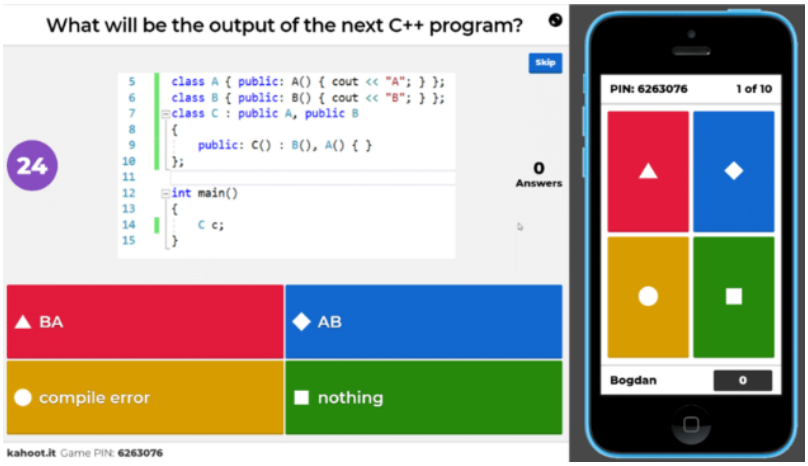

Figure 2.2: The Kahoot! web and mobile interfaces [17]

The game play is kept simple - a user creates and sets up a quiz, adding texts, images and the correct variants for each question. After the quiz is created, it is "hosted", and a generated game PIN is displayed. Players connect to the game using this number. The questions are timed, and the more time passes, the lower the awarded score is. At a question has been answered by every single user, the leader-board is displayed, containing the current number of points for each participant. Incorrect answers are not awarded any points.

Kahoot! can be played from a wide range of devices, including web browsers, mobile applications and tablets. [18]

2.1.3 TriviaCrack

Originally named Preguntados, Trivia Crack is a mobile game developed by the Argentinean company Etermax [44], initially launched for both iOS and Android on the 26th of October, 2013, in Spanish, since its core audience was meant to be Latin America, and was later translated into English. [15]

Figure 2.3: The Trivia Crack quiz interface [19]

The game's questions are based on six different general knowledge categories, namely Art, Entertainment, Geography, History, Science and Sports, and each category is represented by a symbolic cartoon character, having its own colour and name- there is Tina for Art, Pop for Entertainment, Tito for Geography, Hector for History, Al for Science and lastly, Bonzo for Sports. [47]

There are multiple game-play modes, namely Classic, Challenge and Trivia Rush. In the Classic version, the player is paired with a random user, and each player takes turns spinning a wheel containing all the 6 topics, after which each user has 30 seconds to pick the correct answer for the question. After three correct answers in a row, players get to either choose one character, or challenge their opponent and steal one of their characters. The winner of the game is the player who manages to collect every single one of the characters first. After a wrong answer, the user loses their chance to spin the wheel of topics. [19]

2.2 Similarities in approach

At their core, both Quizzicat and the applications presented above serve the same purpose - which is to facilitate an environment for the average user to play trivia quizzes against one or more players.

All of these applications require that the user create an account, either via an e-mail address or by linking an existing social media account to the application profile, such as a Facebook, Google or Twitter account.

Similarly to QuizUp and Trivia Crack, Quizzicat users can submit original questions, that must first go through an approval process before they are added to the topics' roster of questions. The purpose of this functionality is to ensure that passionate users do not get bored playing their favorite topics, since new, challenging questions that they have not encountered before are added.

Both Kahoot! and Quizzicat provide the user with a multiplayer mode. For both of the applications, a unique game PIN is generated, and all users who are aware of this number can join the game before it starts. A leader board is displayed in both applications, so that the players are aware of each other's score and ranking.

In the same fashion as QuizUp, on their profile page users can see statistics about the topics that they play, ranked from the most played to the least played and the amount of correct questions answered in comparison to the amount of incorrectly answered questions.

2.3 Differences and improvements

The Quizzicat application differs adds a couple of different functionalities or revises the approach of already existing ones present in the previously mentioned applications.

To start off, the application has a customizable single player mode, where the user can configure the number and the difficulty of the questions included in the quiz. The main advantage of the single player mode, as opposed to two player games with randomly selected users, is that it prevents the chance of playing against bots that are rigged in order to always choose the correct answer, which has become a problem for applications such as QuizUp. Another advantage would be that users are allowed to play at their own pace and knowledge level.

Secondly, the multiplayer mode also differs in execution - the user that creates one such game can choose the concrete topic that will be played, as well as their difficulty and the number of questions, and the content of the quiz will be randomized. This means that the player creating the game can also participate, without the possibility of having the advantage of already knowing the questions. In contrast, the Kahoot! multiplayer does not allow the user creating the game to participate, only to observe and moderate.

Furthermore, the addition of questions to the topics' roster is automated, as a question is added to the database of possible questions when it receives an average of at least 4 stars from at least 65% of

the user base. The system also removes extremely unpopular questions and heavily reported questions, which are the ones having over a rating equal or lower than 2 stars from at least 65% of user base, or that have been reported by at least 65% of existing users.

Chapter 3

Prerequisites of Android Mobile Development

In order to create mobile applications for the Android platform, it is required to have knowledge of the main development tools used in the industry and of the structure of an Android application project. All the fundamental tools and theoretical aspects regarding Android development are encompassed in this chapter.

3.1 The Kotlin Programming Language

Kotlin is defined as being a general-purpose, open source programming language, which aims to provide both object oriented concepts and functional programming ones. [16] Historically, it originated in 2011, was made open source in 2012, and is designed and developed by the software company JetBrains. [16] It is an Android native programming language and, since 2019, is the language supported by Google for mobile development. [21]

One of the main features of the language is its combination of object oriented programming aspects such as classes with methods, operator overloading and method overriding, with the functional programming style, supporting lambda expressions, anonymous and inline functions, lazy loading and higher-order functions. [12]

Some other key features of Kotlin are the fact that it is a statically typed programming language, meaning that, at compile time, the type of every variable and expression is known, the fact that its type system eliminates any chance of NullPointerExecptions through the safe call operator ?:, automatic smart casting of immutable values and many others. [12]

3.2 The Android Studio Development Environment

Android Studio is a free integrated development environment (IDE), developed and released by Google and JetBrains to serve as the official development platform for Google's Android operating system. [7]

Since Kotlin is the language officially supported by Google for Android mobile development, is is only natural that Android Studio completely support this programming language, providing tools for creating projects written in Kotlin, adding Kotlin files to existing projects, lint checking, setting breakpoints and debugging, running the code via emulators and converting Java code into Kotlin code. [8]

3.3 Android Studio Project Structure

The definition of a project in Android studio includes numerous types of files - files contaning source code, assets, images and vectors, built configurations, default libraries - some created from the start by the IDE, and others created by the programmer. All of files and modules that make up a project are displayed in the Project window located on the left side of the IDE. [9]

3.3.1 The AndroidManifest file [2]

The root of every Android Studio project is the AndroidManifest.xml files, which is automatically generated by Android Studio when the application is created. The file contains essential information about the project structure, among the most important ones being:

- The application's **package name**, matching the code's namespace
- **Activities** - all the created activities need to be declared in the manifest in order to be able to be run on the device. Each component needs to specify the name of the implementing class. The activity that is first launched when the application starts is also defined in the manifest file.
- **Permissions** - for security reasons, some permissions, such as the application's access to the internet connection of the device, are disabled by default. In the manifest file, both the permissions that the application needs to have in order to be granted access to these protected part and the permissions that other applications need to have when they try to communicate with the app are defined.
- **Software features** - the minimum API level of the device targeted needs to be specified in the manifest file.

3.3.2 Gradle build automation [3]

Another important aspect of Android development is represented by the Gradle build automation system, whose purpose is to compile application resources and source code into a single compressed APK file. The execution of the build happens each time the application is opened and each time it is installed on a device. It is also extensively used to import external libraries and frameworks, simplifying the inclusion of third party software in the project.

The most important gradle files are the following:

- **settings.gradle** - located in the root project directory, this file specifies which modules are included when the application is built using Gradle. By default, only the "app" module, which is the main module of the project, is included.
- **build.gradle** - the top-level build file, located in the root project directory, defines the repositories and dependencies that apply to all the modules of the project. No third party dependencies or plugins can included in this file.
- **build.gradle** - the module-level build file, located in each /project/module directory, allows you to customize the dependencies for each module of the application. This is the file through which dependencies for third party applications and plugins can be added.

3.3.3 Layout files [4]

The graphic design of the application is defined using XML Layout files, build using a hierarchy having either View or ViewGroup object nodes as the root, depending on the purpose of the Layout file. Views are usually utilized for creating widgets, subclassing other objects such as Buttons or ImageViews. ViewGroups are typically used for creating layout structures, such as the ConstraintLayout.

While the programmer can create different UI element programatically, in the source code of an Activity, using XML files awards a greater level of flexibility, resolution scalability for different screen sizes and orientations of devices and isolation, since the presentation and the functionality are kept separate. Android Studio provides preview and design tabs for the created XML files, meaning that you can perform minute adjustments of the interface and see the changes immediately, as opposed to having to run the application every time a change is made. The design tab allows you to create a design of a screen by dragging and dropping different objects into the screen space, making it easier to create and visualize the graphic design of the application.

3.3.4 Activity classes [5]

In the context of Android development, an activity represents the entry point for the interaction between the application and the user. The activity provides the window in which the screen UI is displayed, and where the behaviour of the functional elements of the screen screen is defined. Generally, each screen that the user interacts with has a corresponding activity that implements it.

Since most applications have more than a single screen, a main activity needs to specified, letting the project know which screen to display first when the application is opened. Each activity can launch another one, and information can be sent from the source activity from the destination. It is mandatory that every single created activity is declared in the ActivityManifest.xml file.

Each activity has its own lifecycle, meaning that it goes through different states, and the transition between these states is handled using a series of specific callbacks. The most important of these callbacks are:

- **onCreate()** - it is mandatory to be implemented and is added by default when creating a new activity class. This also represents the point in the lifecycle of the activity when the layout of the activity's interface is set, using the method **setContentView()**
- **onStop()** - it is called when the activity is no longer visible to the user, which happens, for example, when a new activity is launched
- **onRestart()** - it is called when an activity that was previously stopped is launched again. This happens, for example, when the user navigates to a screen, and then presses the back button, returning to the previous activity. This callback makes sure to store the state in which the activity was before onStop() was executed

3.3.5 Resource files [6]

Each Android project has a dedicated resource folder that contains items such as bitmaps, fonts, layouts, strings used in the interface (for example as the label of a button), custom shapes, vectors and images.

The layout XMLs discussed in subsection 3.3.3 can be found in the Resources folder, which can be identified in the project structure by the name "res".

A good practice of Android development is to separate resources such as images and strings from the code and the layouts. The resources should then be grouped together in appropriately named directories in the res folder. This separation facilitates easier modification of resources in the code.

An example would be the situation in which the developer needs to change the background color of a vector. When the resource is separated from the the code itself, the change only needs to take place in the resource file itself, rather than for every instance in which the file is used in the code. Additionally, separating the strings from the code and the layout files allows the application to be translated into different languages, allowing the app to be launched directly in the specific languages of the user running it.

Chapter 4

Design Patterns

This chapter is meant to provide an overview of design patterns, going over what a design pattern is, how and why it is used in software engineering and the classification of existing, well-known and used patterns.

4.1 Definition

In the world of software engineering, a **design pattern** represents a language-independent, customizable template used for maintaining the code clean and solving various design problems within the the source code. It acts as a blueprint for solving common, reoccurring issues in software design and architecture. [23]

4.2 Benefits of using Design Patterns

Integrating design patterns into the codebase has multiple benefits, the most obvious ones being the separation of concerns, respecting the principle of single responsibility for classes and low coupling between objects. Since design patterns are not bound to any specific programming language, they support a better understanding of the logic behind an application, since the behaviour can be explained using the classical names of the design patterns, allowing programmers to explain how a functionality is performed without being constrained by language specific implementation details. [24]

4.3 Criticism of Design Patterns

The main point of contention regarding design patterns is whether they are useful in the context of certain applications and programming languages. The efficiency of using a design pattern heavily

depends of the technical specifics of the chosen programming language. For example, in some programming languages, it makes little sense to create classes for the Strategy design pattern, when the same functionality can be achieved using a local lambda function. Another criticism of the usage of design patterns is that it can create more programming overhead that is necessary for implementing a functionality, which can go unnoticed by less experienced programmers. [25]

4.4 Classification

Design patterns are most commonly grouped together based on the type of design issue that they solve. Thus, three main categories have emerged, grouping patterns by the kind of design problem that they solve. [22]

4.4.1 Creational Patterns

The purpose of creational design patterns is to help with the creation, installation and instance management of objects, increasing the reusability and maintainability of the source code. [42]

4.4.1.1 Factory Method

The purpose of the Factory Method design pattern is to handle the problem of creating objects, usually ones that implement a common interface, without having the explicitly specify the type of object being created, replacing the need to call a constructor of a specific type. [26]

According to the "Gang of Four", the main purpose of the Factory Method design pattern is to "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses." [10]

The factory method is useful when delegating some responsibility to helper subclasses, without being able to anticipate at that point in the code the exact type of the helper object that will be used. In a case when there exists a multitude of subclasses inheriting from the same interface, usage of this design pattern replaces conditional blocks that switch the behaviour of the application according to the actual type of the object, giving the code a higher level of abstraction, isolation and maintainability. This way, new implementations of the superclass can be created and used at any time, without having to change the rest of the logic of the application. [10]

In order to use this pattern, a method needs to be defined for creating the object of a common interface, which implementing classes can then use as is or override and then specify the concrete type of the object being created. [10]

4.4.1.2 Abstract Factory

The Abstract Factory design pattern is similar in nature to the Factory Method, except that, instead of creating instances, it deal with creating groups of objects of related types. https://refactoring.guru/design-patterns/abstract-factory

According to the GoF, the Abstract Factory is meant to "Provide an interface for creating families of related or dependent objects without specifying their concrete classes". gof - 87

The Abstract Factory should be used when an application contains groups of classes and objects that are meant to be used together. [10]

This design pattern is more commonly used in complex frameworks, such as the Swing GUI toolkit, which supports different themes for the GUI objects. The user can select the custom appearance of the elements, which would implement the same base interface, customizing only the design properties. In one such case the usage of the design pattern is to act as a factory of factories, launching a different factory method based on the appearance configuration parameters, thus encapsulating the creation of the family of objects. [10]

4.4.1.3 Builder

The Builder design pattern is meant to handle the creation of complex objects, that have a variable number of attributes and for which writing a large number of constructors would be needed in order to handle the initialization of every field. Unlike a constructor, for which all values specified in the method definition need to be specified, a builder takes arguments by chaining method calls that set each field, allowing every possible combination of fields to be specified. [27]

As the GoF mentions, this design pattern is used in order to "Separate the construction of a complex object from its representation so that the same construction process can create different representations" and provides the code with a higher degree of reusability and flexibility, since for the creation of objects with different fields, the same construction methods would be called. [10]

The Builder design pattern is usually implemented by creating two classes - a Director class, which delegates the creation and assembly of the parts that make up a complex object, and the Builder object itself, which has separate methods for building each part of the object. [10]

4.4.1.4 Prototype

The Prototype design pattern is present for any class that implements a method through which the object can create a new instance by copying itself. [28]

An implementation of this design pattern requires that a class have a method *clone*, which returns an instance of itself with the current state of the fields, which can then be modified if needed. Thus, the responsibility of creating clones of given instances is delegated to the objects being copied, decoupling the cloning process from the code. [10]

This design pattern solves the issue of creating clone instances of complex objects, where each field would need to be copied individually into the new instance, not to mention the fact that access to some fields may be private and thus cloning from the outside would not work at all, or objects of classes that remain unknown until runtime. [28]

4.4.1.5 Singleton

According to the GoF, the Singleton design pattern's purpose is to "Ensure a class only has one instance, and provide a global point of access to it". [10]

This design pattern is applicable for objects that hold system data, such as the database reference or the registry. [10]

A class is considered to be a singleton if its constructor is private, access to the instance can only be obtained through a public static method, it is only initialized once and then is never changed and is thread safe. The properties can be achieved through static operations, lazy initialization and thread synchronization. [10]

4.4.2 Structural Patterns

The purpose of structural patterns is to manage the interaction between classes and objects that make up larger structures, while maintaining the composed hierarchy of objects flexible and maintainable. This design pattern is particularly useful for incorporating external libraries into the codebase of an application, while maintaining low coupling and isolation between the 3rd party classes and the ones of the user application. [10]

4.4.2.1 Adapter

Also commonly known as a Wrapper, the Adapter design pattern provides the middle ground for the interaction between two classes having incompatible interfaces. [10]

In order for a class to be considered an adapter, it needs to "wrap" one of the two incompatible interfaces, converting the conflicting components and returning them in a format that the second object can work with. Thus, the functionality of one of the objects can be used by the other, despite their incompatibly, by communicating with the adapter object. Another advantage of this design pattern is

that the two classes remain independent and decoupled, without one having to incorporate the other through composition in order to access some of its functionalities. [29]

4.4.2.2 Bridge

The Bridge design pattern is implemented when there is a need to decouple one or more closely related classes from one another, forming two independent hierarchies - the abstraction and the implementation, which can then be changed and used on their own for various purposes. [30]

This design pattern comes in handy when there exists more than one implementation for a given abstraction, and the decision of which one to use is made at runtime, instead of at compile time, or when functionalities belonging different implementations and abstractions of the same related classes need to be combined and extended independently. [10]

A traditional example of the usage of the Bridge pattern can be found in applications where the user interface is separate from the backend implementation. In this case, the UI represents the abstraction, while the underlying API that is called by the front end would be the implementation. Since both the UI and the API can be extended for different situations - the UI can be extended for mobile devices, tables or web browsers and the back end can have different implementations for running on either a Windows device or an Linux - the code related to specific front-back end combinations can be extracted into separate classes, representing the Bridges. [30]

4.4.2.3 Composite

The Composite design pattern allows the decomposition and reassembly of objects intro tree-like hierarchies, while also allowing the programmer to work with both the individual objects and the partial compositions as if they were independent objects. [10]

This implementation of the Composite pattern generally contains three main entities, namely *the Leaf* - that represents the smallest individual object that is handled - *the Component* - acting as the common interface for Leaf and Composite objects - and *the Composite* - whose purpose is to maintain a collection of objects of type Component, and to delegate operations to each of its sub-components of type Leaf. [45]

This tree-like organization of this design pattern should be applied when there is no difference between the way an individual object and a composition of objects are handled. [10] An example of such an application would be the Notepad application, which treats both characters and groups of characters - lines, pages, paragraphs - in the same way, and operations that can be performed in one way on an individual character are executed in the same way on groups of characters. In this scenario,

the Leaf element of the would be the individual character, the Composite would be an array of such characters and the Component would be a common interface for the two objects. [45]

4.4.2.4 Decorator

The purpose of the Decorator design pattern is to allow additional functionalities to be dynamically attached to an object, providing a more flexible alternative to extending by creating subclasses. [10]

This pattern can act as a wrapper, since sometimes it is desired to add a certain attribute or functionality to one object, instead of the entire class. An example of the usage of a decorator class is for reusable components of a GUI interface - buttons, text field, drop down menus. In order to change some property of a component, such as the colour of a button or the size of its border, at runtime, a decorator class could be used, which contains a reference to an object of class Component (a common interface for the object that is to be decorated and the decorator) and which performs the operation of changing the colour of the button and then delegates the request to its Component attribute. [45]

4.4.2.5 Facade

The Facade design pattern is meant to provide a simplified interface through which a complex system, usually belonging to an external library or framework, can easily be incorporated into an application, without revealing implementation details or coupling the code of the library too tightly to the code of the application. [10]

In order to use this design pattern, a single Facade class is required, which will handle the dependencies of the subsystem being incorporated into the application. Thus, the facade class represents the single entry point to the added external library, which can communicate with the application code or with other subsystems, each with their own facade interface. The most obvious benefit of using this design pattern is the fact that, if any changes were to happen within the 3rd party library or if the currently used subsystem needs to be replaced entirely, only the code from the Facade class would need to be replaced. Otherwise, the codebase would need to be analyzed for each instance of objects belonging to the 3rd party library. [10]

4.4.2.6 Flyweight

The Flyweight design pattern permits sharing common parts of the state of some fine-grained objects, instead of each individual object maintaining all the data, in order to support a larger number of objects, without filling up the memory. [31]

When using the flyweight design pattern, two types of states are defined, namely the *intrinsic* data, storing the constant information about an object, which can only be read and which is stored in the Flyweight objects themselves and is stored between them, and the *extrinsic* (or variable) data, which can not be shared and must be passed dynamically, at runtime, by the Client classes. [45].

This design pattern should be used when it is known that the application requires the creation of a large number of objects, some of which having shared data, and that the amount of objects created could overwhelm the physical capacities of the device running the program. One example where the Flyweight pattern in used is for real time video games. If each pixel is stored in a separate object, as more and more of the environment is loaded, the game may crash due to the computer not having enough RAM storage. In order to minimize the chances of this happening, the Flywieght pattern could be used to store pixels having the same colour and texture values, which would be shared between different objects. [31]

4.4.2.7 Proxy

The Proxy design pattern allows the creation of a placeholder that can be used instead of another object, through which the original object can be changed, either before or after the substitution request is made. [32]

The advantage that this pattern provides is that it allows control access to objects that could belong to 3rd party libraries, or could be expensive to create or clone. By using a Proxy object and controlling the access to an object, the full cost of its initialization is delaying until it is mandatory. The Proxy design pattern can be viewed as a more sophisticated version of a simple pointer. [10]

In order to apply the Proxy design pattern in order to create a substitute for a desired object, the Proxy class needs to maintain a reference to the wanted object, while also following the interface of the instance that it is meant to replace, in order for the Client to be able to receive requests from the placeholder object as if it were the concrete one. [32]

4.4.3 Behavioral Patterns

The purpose of behavioral design patterns is to manage the assignment and delegation of responsibilities between objects during the program execution, specifying not only the behaviour of the objects involved, but also the flow of the communication between them. [10]

4.4.3.1 Chain of Responsibility

The basis of the Chain of Responsibility design pattern is to send requests between objects along a chain of any number of candidate handlers, which can either process the received request, based on some run-time conditions, or pass the request to the next object in the chain or requests. [10]

The reason for passing a request along a chain of potential handlers is to avoid high coupling between sender objects and their corresponding receiver, allowing multiple objects the possibility of handling the request. [10]

In order to integrate the design pattern into an application, the chain of handlers needs to be created. A handler is simply an object which contains a method having as a parameter the request that is being passed around, and that checks whether the request will be processed by this particular object. It also contains a reference to the next handler object in the request chain, and the processing of the request has ended when either it is processed by one of the objects, or until the last handler in the chain is reached.

4.4.3.2 Command

The Command Design Pattern encapsulates a request into an object containing all the data that was originally sent via the request. According to the "Gang of Four", this design pattern allows the programmer to "parameterize clients with different requests, queue or log requests, and support undoable operations". [10]

The command design pattern is useful in the case when requests are made without any information regarding the operation being requested or its receiver. [10] Command objects usually act as a bridge between the front-end interface of an application and the back-end logic that is executed for the given user interaction. This maintains the principle of the separation of concerns, since the layout elements do not call the business logic methods directly, but rather delegating the method call to the Command. [33]

4.4.3.3 Iterator

The Iterator design pattern provides a way to parse the items of a collection of objects, without exposing the underlying implementation details of the collection. [10]

Regardless of the internal representation of a collection, an iterator over it must provide a means to access each element in a predefined order, without accessing the same element twice. The purpose of this design pattern is to decouple the parsing process associated to a collection from its code, and move it into a separate class, which will encapsulate all the details regarding the traversal of the collection,

including the current position, the list of elements, the next element to be traversed, different methods of parsing data, and others, depending on the internal structure of the collection. [34]

4.4.3.4 Mediator

The Mediator design pattern's purpose is to encapsulate the interaction between a some objects, reducing the inter-dependencies and the tight coupling between them by having the entities interact through the mediator object, restricting direct method calls and references between the objects. [10]

In order for a class to implement the Mediator design pattern, it needs to contain the necessary interfaces in order to communicate with each desired object, as well as to implement the way these classes communicate with each other, controlling the communication flow by receiving and delegating method calls between the objects through notifications received from them. [45]

4.4.3.5 Memento

The purpose of the Memento design pattern is to allow the state of an object to be captured, so that later during the program execution the saved state can be restored, but without exposing its underlying representation. A practical example for when saving the state of an object at a particular time is needed is for undo/redo mechanisms, where a certain object needs to be restored to its previous or changed states when an event is triggered. [10]

This design pattern solves the problem of storing the state of an object at a given time by delegating responsibility of creating a snapshot of the current to the object itself. A Memento class is also created, its purpose being to store the original object's internal state. An object of this type is created by the desired instance itself, which need to contain a *createMemento* method. The encapsulation principle is not violated by using this pattern, since usually the Memento is a local class of the original object, with all members private. Along with the method for creating a snapshot of the current state, the original class must also contain a method for restoring the attributes of the object to the ones stored in the Memento. [35]

4.4.3.6 Observer

The Observer design pattern behaves like a notification mechanism, in the sense that it allows a dependency to be defined between multiple objects, so that if the state of one particular instance is changed, the rest of the objects are notified. [10]

The way this design pattern is implemented is by creating a *Publisher* class, that maintains a list of Observers and provides methods for adding a new subscriber, removing an already existing one, and

notifying each observer object that a change of the internal state has been performed. The *Observer* interface contains a method for updating the internal state after a modification has been notified. [36]

This implementation allows for minimal coupling between the original object and its subscribers, since the Publisher class does not need to be aware of the implementation of each Observer, maintaining a list of objects of the interface type, while the Observers themselves do not need to contain direct references to the original object. [10]

4.4.3.7 State

The State design pattern allows an object to change its behaviour when there are some changes performed to its internal state, appearing as if the class of the object has changed. [10]

This design pattern is useful when it is known that the objects of a program can go through a defined number of states. A class is to be created for each possible state that the object can take, which will contain the behaviour specific to the current state. The object that can change its state holds a reference to its current state and delegates state-specific actions to the corresponding state object. [37]

4.4.3.8 Strategy

The Strategy design pattern defines a family of various algorithms, each encapsulated in a separate class, that can be used independently and interchangeably. [10]

This design pattern is useful when there exists a class that implements the same functionality in a multitude of ways, the behaviour depending on some request made by the client. In this context, each independent algorithm is extracted from the original class and placed in a new one, this class being the Strategy. The original class stores the strategy interface, and delegates the execution of the algorithm to the corresponding concrete class, not executing the behaviour itself. [38]

4.4.3.9 Template Method

The Template design pattern provides an outline of an algorithm encapsulated within an operation, allowing the redefinition of some steps in the procedure, without modifying its base structure, thus reducing the amount of duplicate code. [10]

To incorporate this design pattern, a desired algorithm needs to be broken down into a series of independent step, each of them being placed in a separate method, which may contain some default implementation or may be abstract. The newly created functions are then called by the template

method. Objects that make use of the algorithm need to subclass the superclass containing the template method and provide their own implementation for the abstract steps of the algorithm. [39]

4.4.3.10 Visitor

The Visitor design pattern is meant to separate algorithms from the objects used in the logic, allowing the definition of a new operation to be performed on an object within a structure, without altering the classes of the elements. [10]

This design pattern can be integrated by placing the new functionality into a separate *Visitor* class and passing the desired object as an argument to the method encapsulating the behaviour, instead of altering the code of the existing classes. [40]

Chapter 5

Application Design

The purpose of this chapter is the present some specific details regarding the organization and design of the application, such as its main use cases, the way the application data is structured and modeled into objects, the interaction between the main activities, as well as the incorporation of design patterns and external libraries into the code.

5.1 Diagrams

5.1.1 Use cases

The user is granted access to the application by creating and then accessing a validated account, either by linking the Quizzicat account with an existing social media account (Google, Facebook), or by creating a new account using a personal e-mail address, in which case the user should provide a display name, a correctly formatted e-mail address, a strong enough password and optionally, a profile picture. After the account is created, the user needs to validate it by accessing the link that will be sent to the inbox of the provided e-mail address. After that, the account can be accesses and the user can log into the application. If the user can not access their account due to a forgotten password, a password reset mail will be sent if requested by the user.

Once logged in, the user is presented with a list of topic categories on the first tab of the application. When selecting a category, all the topics belonging to said category are displayed. Once a topic is selected, the user can customize the individual trivia quiz, specifying the difficulty and the number of questions. Each question can be answered in at most 10 seconds, and the game stops if the user fails to select an answer within the time limit. At the end of the quiz, the number of correct and incorrect answers is displayed.

Users can create or join multiplayer trivia games. When creating the quiz, the user specifies the

topic of the questions, as well as their difficulty and number. Users can join the game by providing a unique 4-digit PIN associated with the game. Once all users have joined the game, the host can start the quiz. Each correct answer is worth 10 points, and the number of points that can be earned decreases by one for every passing second. An incorrect answer is awarded no points. After answering every question, users can see their own, as well as the other's accumulated number of points, in a scoreboard. The user can also see a list with all the multiplayer games that they have joined, seeing the username of the host, the date in which the game was played, as well as their number of accumulated points. If the user has managed to score the highest number of points for that particular trivia quiz, a crown is displayed next to their score, marking them as the winner.

New questions can be submitted by users, and rated by other players. A user can create a new pending question, by providing its category, topic, the text of the question and its answers, as well as the correct answer. After submitting the question, other users can see it in a leaderboard and vote for it once, giving it between 0 and 5 stars. If a player considers that a submitted question is inappropriate, incorrectly labelled, redundant or a duplicate, they can choose to report it, in which case the question will not longer be displayed to them. Once a certain approval rating has been met, questions are added to the game roster. Similarly, if a question is reported by a multitude of user or has a low enough rating, it will be removed from the leaderboard for all everyone. A user can see which questions they have submitted and are still awaiting approval or rejection, as well as their accepted and rejected questions, in separate lists.

A user should be able to see recommendations for new topics to play based on their playing history. The application displays the topics that the user has played at least one time, in a decreasing order of the number of times played, as well as topics most similar to the user's 3 most played ones.

Each user can see different statistics regarding their playing history, including the total number of individual games played, the total number of correct and incorrect answers, the ratio between correct and incorrect answers, as well as a breakdown of their played categories and topics.

Along with these main functionalities, the user is able to customize their profile information, changing their profile picture, username and e-mail address associated with the account, if it was created using a personal e-mail address and not a social media account. The user is also able to delete their account, reset their password and clear their playing history at any time.

The diagram below illustrates the use cases mentioned above, for the main actor which is the user that has downloaded and installed the application.

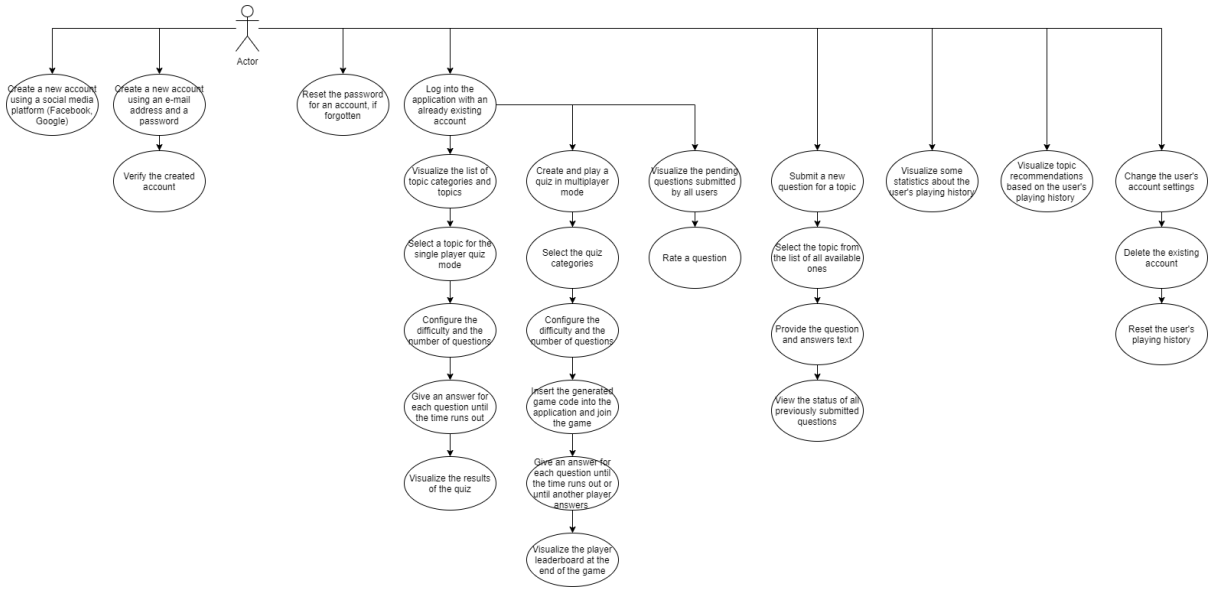

Figure 5.1: Use case diagram for the application

5.1.2 Sequence diagram

Each of the sequence diagrams below illustrates the way objects involved in the main functionalities of the application communicate with each other, launching new activities and navigating to and from different screens, depending on the action of the user.

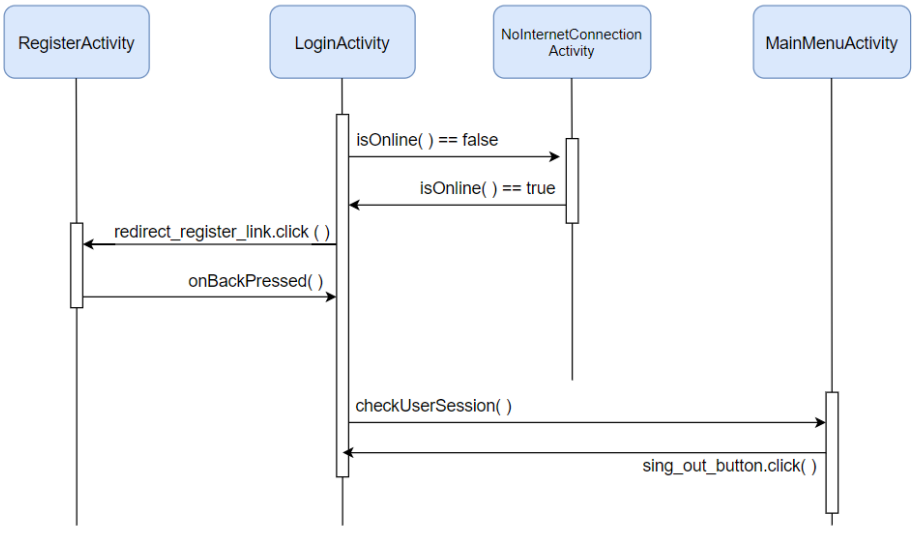

Figure 5.2: Sequence diagram for the login process

The first diagram, 5.2, shows the interaction between activities during the process of creating an account and/or logging into the application. The user can navigate between the Login screen and the Register screen using buttons. Additionally, when there is no internet connection on the current device, the flow is automatically transferred to the No Internet Connection screen.

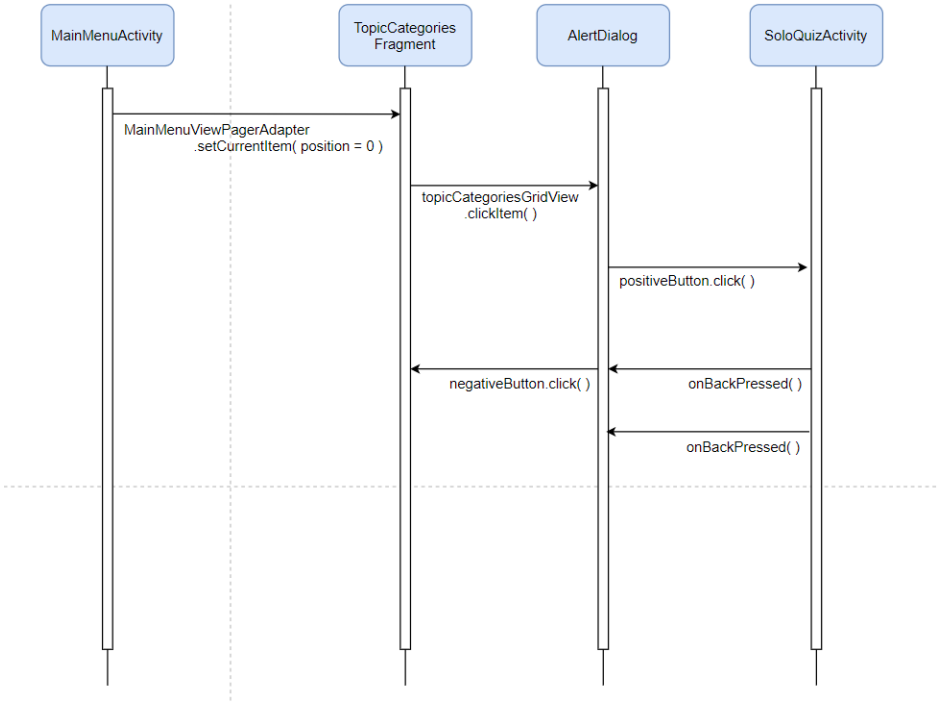

Figure 5.3: Sequence diagram for the solo gameplay functionality

Diagram 5.3 shows the interaction between the Activity classes involved in the flow of playing individual trivia quizzes. This starts by navigating to the first tab on the main screen of the application, and then clicking on an icon of a desired topic, which would transfer the flow of the program to an AlertDialog window, which in turn hands over control to the Solo Quiz Activity. Navigation between these objects is done using the back button from the screen or the negative button of the AlertDialog.

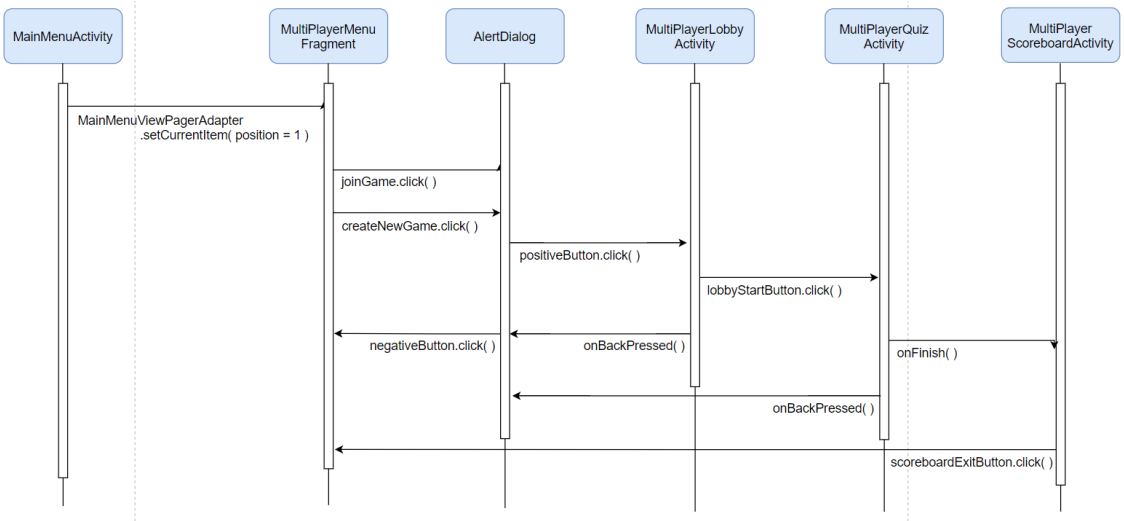

Figure 5.4: Sequence diagram for the multiplayer functionality

The third diagram, 5.4, illustrates the sequence in which activities are switched during the multiplayer gameplay. The flow of this functionality starts by navigating to the second tab of the main screen. From there, the user can interact with either the *Join Game* or **Create New Game** buttons, and in both cases control would then be transferred to an object of type AlertDialog. Once the user clicks its positive button, they are redirected to the Lobby screen. The application control is handed over to the Mutliplayer Quiz activity once the host of the trivia game presses the *Start Activity* button. After the user answers all questions in the trivia game, the activity is switch to the Scoreboard one. The only navigation path from this activity is to the main menu, using the back button of the device or the **Exit** button on the screen.

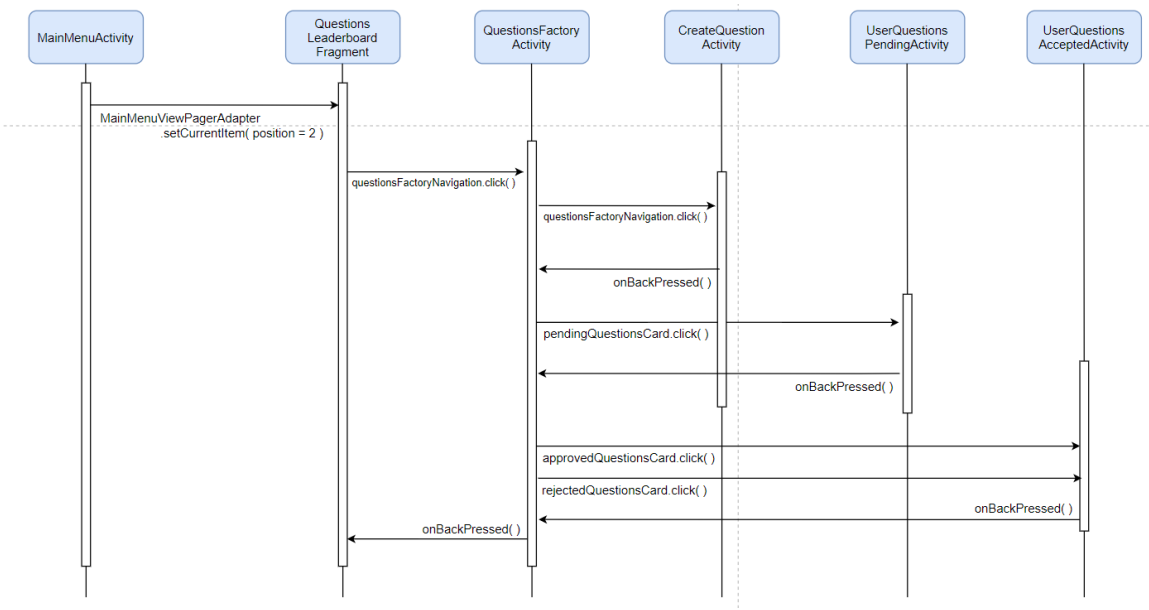

Figure 5.5: Sequence diagram for the question submission functionality

Diagram 5.5 shows all the possible interactions between objects involved in the questions submission functionality, starting from the third tab of the application. From there, the user can navigate to the Questions Factory activity, which acts as a menu for a user's submitted questions. By pressing on the available card views, the user can choose to either delegate to control to the Create Question Activity, the Pending Questions Activity or the Accepted Questions Activity, and navigate back to either the Question Factory or the main menu via the back button of the device.

In the fifth diagram, 5.6, the interaction between objects involved in the Topic Recommendations functionality is shown, starting when the user navigate to the fourth tab of the main menu. From there, if a Topic icon is selected, the program execution is transferred to the same entities involved in the Solo Quiz functionality.

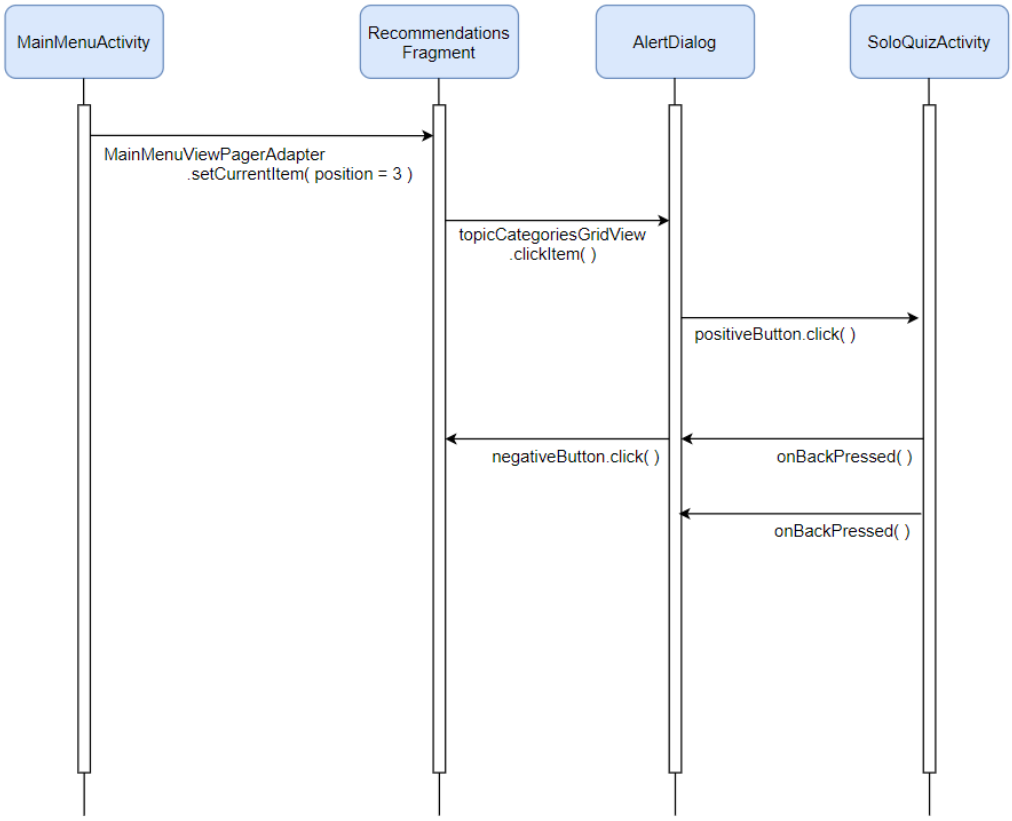

Figure 5.6: Sequence diagram for the recommendations functionality

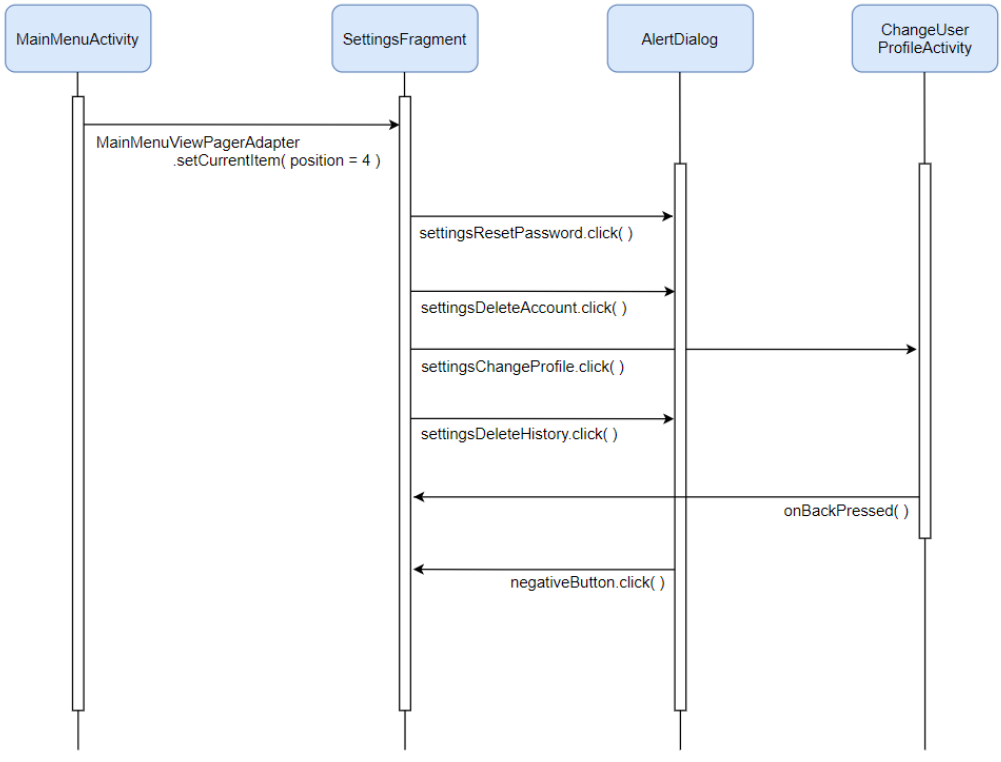

Figure 5.7: Sequence diagram for user profile settings functionalities

The last sequence diagram, 5.7, shows the interaction between elements involved in the activities that make up the User Settings functionality. From the Settings Fragment, the user mostly interacts with AlertDialog objects, each having a different purpose and underlying logic. The only exception is for changing the account data, in which case the control of the flow is transferred to the corresponding activity.

5.1.3 Database diagram

The diagram below presents the structure of the Firestore documents containing the data that the application works with, as well as the relations between the objects.

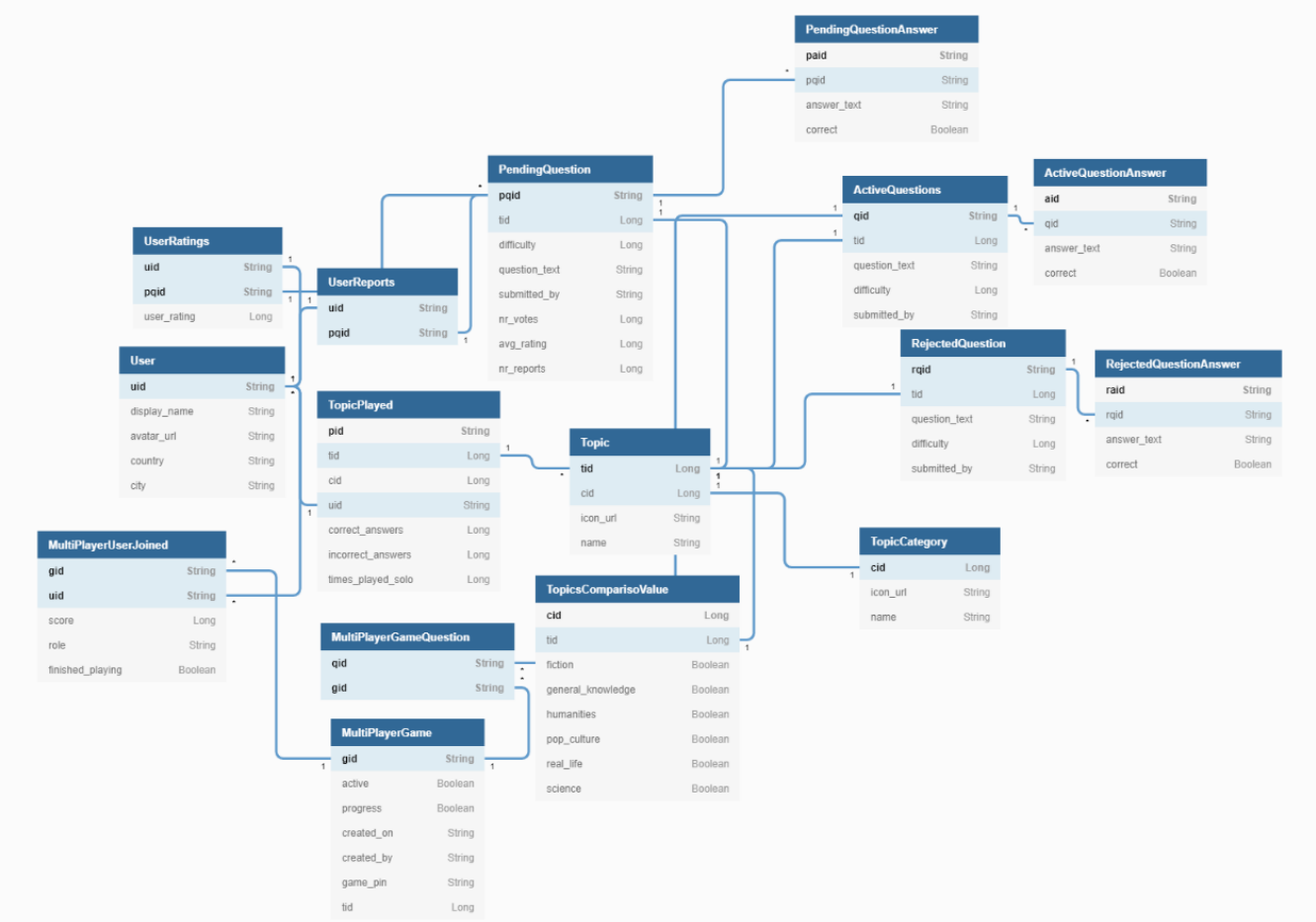

Figure 5.8: Database diagram

The data about the user's profile details and gameplay experience can be found in collections **Users TopicPlayed** and **MultiPlayerUserJoined**. The latter two contain information about the user's playing history, the first one of the two about the individual quizzes that the user plays, and the second one about the mutliplayer games that the user either joins or creates.

All the data regarding the multiplayer quizzes is stored in structures **MultiPlayerGame**, containing the general information about the game, **MultiPlayerUserJoined**, where entries are added for the users that create or join a game, and **MultiPlayerGameQuestion**, which stores the IDs of the questions that will be used for a particular game.

Information about the available topics is stored in tables **Topic**, **TopicCategory** and **TopicsComparisonValue**. Each topic is associated to a category, so that the user can filter through them in a more accessible manner. For each existing topic, in the TopicsComparisonValue structure, the values for the comparison fields are set, which are used in order to determine how similar two topics are. These content of these collection is only read by the application, no modifications are performed on the entries.

The application handles three kinds of questions, active ones, which are included in the game roster and stored in **ActiveQuestion**, pending questions, which are still being rated by the players, stored in **PendingQuestion** and rejected ones, which have recieved a poor rating or have been reported too many times by users, stored in **RejectedQuestion**. The corresponding answers for each type of question can be found in tables **ActiveQuestionAnswer**, **PendingQuestionAnswer** and **RejectedQuestionAnswer**. The user's interaction with the questions is stored in tables **UserReports**, where new entries are added every time a user flags a question and **UserRatings**, where the score given for each question by a user is recorded.

5.2 Application architecture

5.2.1 Integration of Design Patterns

In order to maintain a clean architecture and code-base, as well as for usability and maintainability purposes, the implementation has been structured in order to integrate design patterns.

5.2.1.1 Facade

The Facade design pattern has been used for two main purposes: to isolate third party library calls from the rest of the code-base, and to provide classes for repeated retrieval of the same data.

An example of the former is class **ImageLoadingFacade**, depicted in the class diagram below:

Since there are multiple implementation of Image Loading Libraries for Android projects, the purpose of this class is to decouple the process of image loading from the rest of the code-base, in case the currently used library needs to be replaced with another. Whenever an image needs to be loaded into either an ImageView or a CircleImageView element, the public methods **loadImage**, respectively

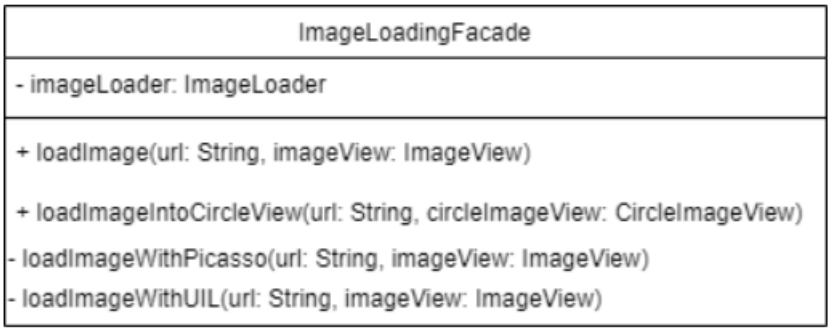

Figure 5.9: Class ImageLoadingFacade

loadImageWithCircleView, are called. Thus, when replacing the current image loader with another one, the only modification that would need to be done is to the code of the public methods, eliminating the need to comb through the code-base in order to replace every instance and usage of a particular image loading library.

The Facade pattern is also used in creating classes for repeated retrieval of some information from the database. An example of this case is class **TopicsDataRetrievalFacade**, depicted in the class diagram below:

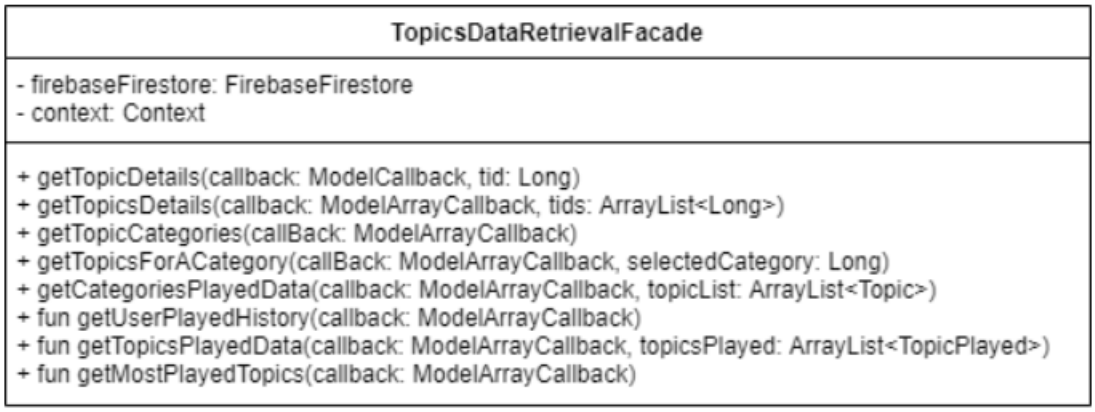

Figure 5.10: Class TopicsDataRetrievalFacade

Since some data regarding the quiz topics and topic categories, such as the Icon URL or the name of the topic/topic category, is needed in multiple places throughout the application flow, class TopicsDataRetrievalFacade also provides a way to reuse the calls to the database that fetch the required data. The data retrieval classes are also meant to isolate the calls to the Cloud Firestore API. The other data retrieval facades used in the application are **MultiPlayerDataRetrievalFacade**, which manages all the data used for the multiplayer games, including questions, users and the game objects themselves, **PendingDataRetrievalFacade**, used for retrieved the Pending Questions for the

Questions Leaderboard, **QuestionsDataRetrieval**, which fetches the questions and answers that are already added to the game and **UserDataRetrieval**, used for retrieval of additional details regarding the user currently logged in, **RecommendDataFacade**, which retrieves the comparison values used for computing the recommendations based on the user game history and computes the similarity index for a given topic, in comparison to all other topics.

5.2.1.2 Adapter

The application makes use of the following adapter classes: **ActiveQuestionsAdapter**, **LobbyUsersAdapter**, **MainMenuViewPageAdapter**, **MultiPlayerGamesAdapter**, **PendingQuestionsAdapter**, **RecommendedTopicsAdapter**, **RejectedQuestionsAdapter TopicCategoriesAdater**, **TopicSpinnerAdapter**.

ActiveQuestionsAdapter, **RejectedQuestionsAdapter**, **PendingQuestionsAdapter**, **LobbyUsersAdapter**,**MultiPlayerGamesAdapter**, **RecommendedTopicsAdapter** implement the existing RecyclerView adapter, binding a custom item view for the RecyclerView element that will be displayed and handling the *onClick* events for various items contained in the custom layout of the element. The first thee adapters are used for displaying either the pending, the active or the rejected questions. LobbyUsersAdapter is used for displaying a list with each user that joins a multiplayer game, while MultiPlayerGamesAdapter adapter displays some information about the user's played games.

Class **TopicCategoriesAdapter** implements the BaseAdapter class and provides an adapter for the grid that displays the Topics and Topic Categories in the Solo Quiz interface. The custom adapter is needed since the element of the grid view has a custom layout and thus the values need to be manually set for each XML element when the item is loaded into the grid layout.

Adapter **TopicSpinnerAdapter** has a similar purpose, but for a different layout element. In this case, the ArrayAdapter class is implemented, in order to create and load a custom element into a given Spinner object.

Class **MainMenuViewPagerAdapter** handles switching between the tabs of the main screen of the application, implementing class FragmentStateAdapter. The most important method of this class has the role of loading an appropriate fragment for each tab.

5.2.1.3 Singleton

The Singleton design pattern is used by the Firebase API, classes **FirebaseAuth**, **FirebaseFirestore** and **FirebaseStorage** being instantiated only once, at program execution. After that, from each activity where one of the previously mentioned classes is used, the same instance of the object is called

via method **getInstance()**. This ensures that only one connection to the storage and database is open during the execution of the program.

Class **HttpRequestUtils** is also an instance of the Singleton pattern, having only one instance that can be retrieved via a public method, and being thread safe. Statement *companion object*, that is wrapped around the INSTANCE attribute of the class is the Kotlin-equivalent of the *static* notation. The INSTANCE attribute also has the *@Volatile* annotation attached, signifying the fact that writes to this field are immediately made visible to other threads. Thread safety is further ensured by fact that, in method **getInstance()**, returning the INSTANCE attribute is done using method *synchronized()*, that sets a lock on the object.

5.2.1.4 Builder

The Builder design pattern has been integrated by using the *builder( )* method of the Android class AlertDialog. Objects of this type can be found in multiple places throughout the application flow, asking the user to confirm an action - deleting their playing history, changing their password, etc. - or having them provide the data for creating a Trivia Game. Because of the flexibility provided by the Builder design pattern, alert dialog screens have been customized to look differently, according to their purpose, by chaining different element creation methods to the main **builder( )** method call.

```
settings_delete_history.setOnClickListener { it: View!
    AlertDialog.Builder(context!!)
        .setTitle("Delete history")
        .setMessage("Are you sure you want to delete your game history?")
        .setPositiveButton( text: "Confirm") { _, _ ->
            run { this: SettingsFragment
                deletePlayingHistory()
            }
        }
        .setNegativeButton( text: "Cancel",  listener: null)
        .create()
        .show()
}
```

Figure 5.11: AlertDialog created with the default appearance

5.2.1.5 Observer

For the multiplayer functionality, it is mandatory that all users can see in real time when someone either joins a game that has not yet started, or finishes answering all the questions. Additionally, a multiplayer game needs to start at the same time for all joined users, at the moment when the host presses the **"Start"** button in the Lobby screen.

```
val inflater : LayoutInflater! = LayoutInflater.from(context)
val customizingQuizView : View! = inflater.inflate(R.layout.view_customize_solo_quiz, root: null)
val selectedDifficulty : Spinner = customizingQuizView.findViewById(R.id.customize_quiz_difficulty)
val selectedNumberOfQuestions : Spinner = customizingQuizView.findViewById(R.id.customize_quiz_number)

AlertDialog.Builder(context!!)
    .setView(customizingQuizView)
    .setPositiveButton( text: "Let's play") { _, _ ->
        val soloQuizIntent = Intent(activity, SoloQuizActivity::class.java)
        soloQuizIntent.putExtra( name: "questionsDifficulty", selectedDifficulty.selectedItem.toString())
        soloQuizIntent.putExtra( name: "questionsNumber", selectedNumberOfQuestions.selectedItem.toString())
        soloQuizIntent.putExtra( name: "questionsTopic", topicsList!![position].tid)
        soloQuizIntent.putExtra( name: "questionsCategory", topicsList!![position].cid)
        startActivity(soloQuizIntent)
    }
    .setNegativeButton( text: "Cancel", listener: null)
    .show()
```

Figure 5.12: AlertDialog where a custom view has been attached

For the implementation of this functionality, a Firestore Snapshot Listener is user, acting as an Observer and notifying all instances of the MultiplayerLobbyActivity class of any change to the state of the collection reference storing the multiplayer games data. Thus, when the host presses the **"Start"** button on their Lobby screen, field *progress* of the current game entry is updated in the database table to be *true*, and the change is notified to all the instances of the Multiplayer Lobby, which in turn starts the MultiplayerQuizActivity.

```
private fun listenForGameStart() {
    val gamesCollection : CollectionReference = mFirestoreDatabase!!.collection( collectionPath: "Multi_Player_Games")
    gamesCollection.addSnapshotListener { snapshot, e ->
        if (e != null) {
            Toast.makeText( context: this, text: "Users could not be fetched! Please try again!", Toast.LENGTH_LONG).show()
            return@addSnapshotListener
        }

        for (changes : DocumentChange! in snapshot!!.documentChanges) {
            val gameID : String = changes.document.data.get("gid") as String
            val active : Boolean = changes.document.data.get("active") as Boolean
            val progress : Boolean = changes.document.data.get("progress") as Boolean
            if (gameID == gid) {
                if (changes.type == DocumentChange.Type.MODIFIED) {
                    if (active && progress) {
                        val gameIntent = Intent( packageContext: this, MultiPlayerQuizActivity::class.java)
                        gameIntent.putExtra( name: "gid", gid)
                        gameIntent.putExtra( name: "gamePIN", gamePIN)
                        startActivity(gameIntent)
                    }
                }
            }
        }
    }
}
```

Figure 5.13: The Observer pattern being used through the Snapshot Listener

5.2.1.6 Template

Since the data retrieval from the Firestore database is an asynchronous process, it is necessary to have a method which notifies when the query execution has finished and returns the complete result of the query. A naive solution for this problem would be to create a callback method for each kind of object fetched from the database. However, considering the large number of entities that are used by

application, this would lead to many duplicate methods, where the only difference would be the type of the parameter assigned to the callback method.

In order to handle the callbacks of every Model object of the application, interfaces **ModelCallback** and **ModelArrayCallback** have been created, which contain the Template Method *onCallback( )*, which takes as parameter either an object or an array of objects of type **ModelEntity**, which is a superclass for all the models used in the application.

Depending on whether one or a multitude of objects are fetched from the database, the method from the corresponding data retrieval Facade that handles the query has as parameter an instance of either ModelCallback or ModelArrayCallback.

The template method provides an abstract step for returning the result of a database query. Since it makes use of the ModelEntity superclass, when getting the result of the data retrieval callback method, any Model instance can downcast the concrete type over the result and further use it in the logic of the application.

```
interface ModelCallback {
    fun onCallback(value: ModelEntity)
}
```

Figure 5.14: Interface ModelCallback

```
interface ModelArrayCallback {
    fun onCallback(value: List<ModelEntity>)
}
```

Figure 5.15: Interface ModelArrayCallback

5.3 Integration of external libraries

5.3.1 FirebaseAuth - user authentication methods

Most mobile applications out on the market today require that the user first create an account before allowing them to access the main functionalities of the app. User accounts are handy things, since they allow the data that is accumulated during the usage of the application by a certain user to be safely stored, allowing offline access and online-offline synchronization and backup of user data in case of unexpected server troubles.

Thus, the authentication process is an integral part of mobile applications, and needs to be handled in a serious manner, in order to avoid the apparition of any security vulnerabilities. [14]

For the Quizzicat application, the user can create a new account in the following ways:

- **using an e-mail address and a password** - a classical method of authentication, the user is asked to provide their personal e-mail address and a password, along with a number of other details such as the display name and the profile picture. The user account is created in the Firebase

Authentication database via the method *createUserWithEmailAndPassword* of the framework. [11] Special measures have been implemented to ensure that fraudulent accounts can not be created using someone else's address: when a new account is created using this authentication method, a confirmation e-mail is sent to the given e-mail address, letting the person behind the address know that an account was created using this e-mail. The user is then asked to validate their account using a unique, one-time verification link. Unless this step has been taken, the newly created account can not be accessed - no matter how many times an attempt to log on with non-verified credentials is performed, the user's access is blocked. The verification e-mail is sent using the Firebase method *sendEmailVerification*.

- **using a Facebook or Google account** - for these two authentication methods, the Facebook and Google APIs have been used, which are widely used and proven to be secure. Users need to log into their social media accounts and confirm the fact that they want to link the current account with the Quizzicat application. Access to the application can be removed at any time from either the Facebook profile or the Google account, and once this access has been removed, the user can no longer sign in using this social media account.

All user data is stored in the Firebase Authentication database, ensuring that only developers associated with the project can view the existing accounts. Still, developers do not have access to all the information regarding a user profile, and can not see in any way the password associated with an account. The information is also secured in the event of a data leak, since the value of the password is hashed.

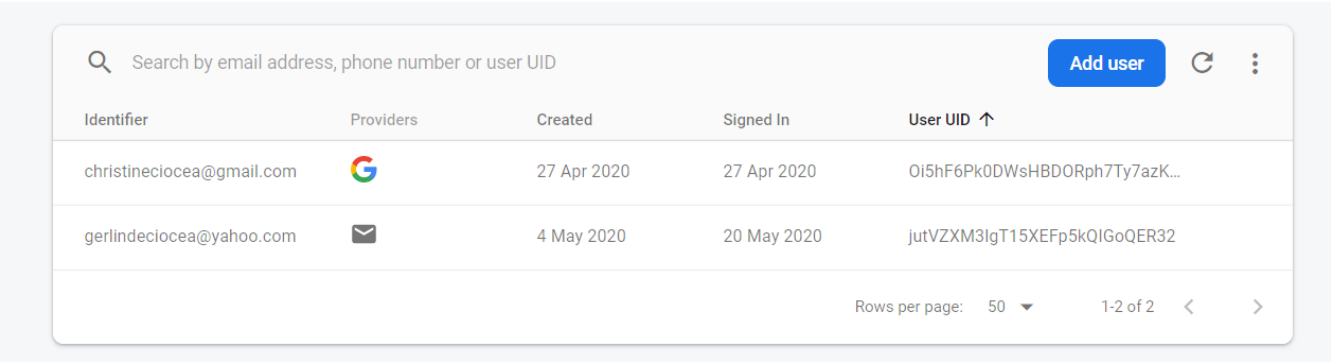

Figure 5.16: The Firebase Authentication interface

5.3.2 Cloud Firestore - server-less data management

The application data is maintained using the Firebase Cloud Firestore framework and is stored in the NoSQL cloud database provided by Firebase.The infomration is organized in collections and documents,

as opposed to the traditional tables. A collection would correspond to a table, containing multiple documents, that act as table entries. The Cloud Firestore framework also provides flexible query creation using indexes, realtime listeners for data changes and offline support.

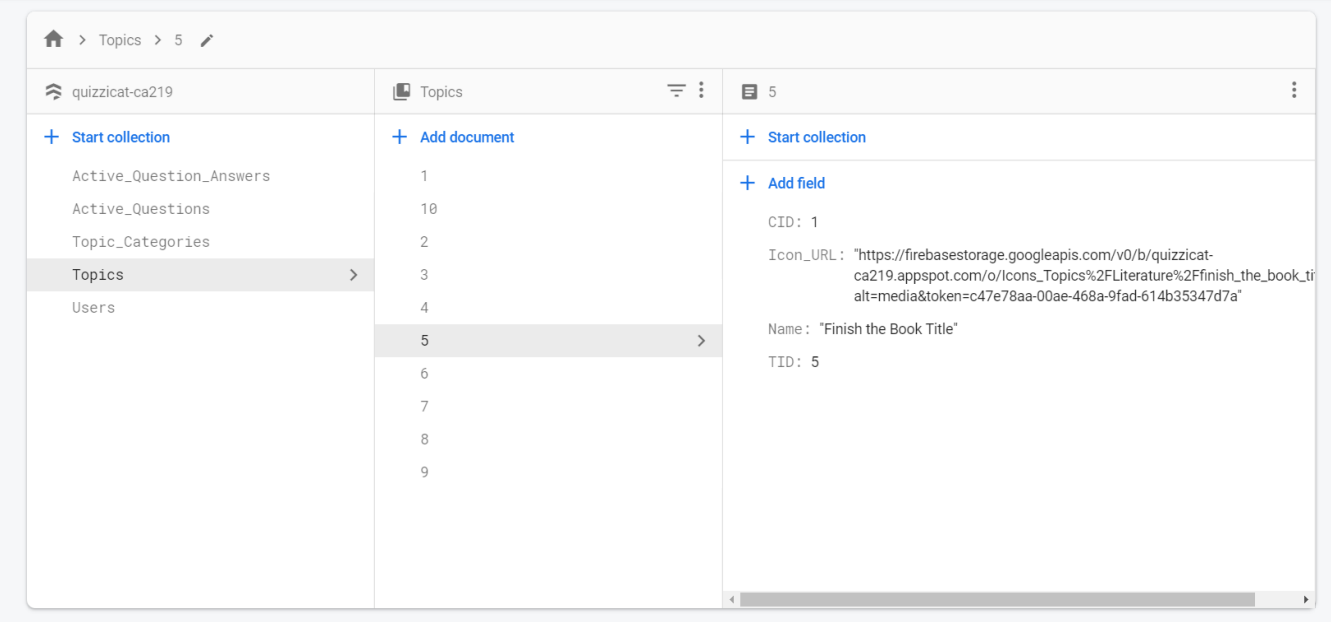

Figure 5.17: The Cloud Firestore interface, showing how the Topics data is stored

5.3.3 Firebase Storage - storing application images

The application contains multiple images and avatars, including the user profile picture and various icons for quiz topic and topic categories. These are stored remotely, using the Firestore framework, since storing all of them in the application package itself would lead to a fairly large APK file size. The framework generates a unique link for each uploaded image, which is saved to a corresponding database field. During the application execution, the URL fetched from the database will be given to an external image loading library.

5.3.4 Picasso - image loading

Since the images are stored remotely, they are accessed in the application via the URL address from the corresponding database entry. In order to load the content of the image itself into the appropriate ImageView elements of the screen structure, the Picasso library is used.

Picasso is an open source image loading, downloading and caching library for Android application, with methods dedicated to handling ImageView recycling, automatic memory caching and complex image transformation.

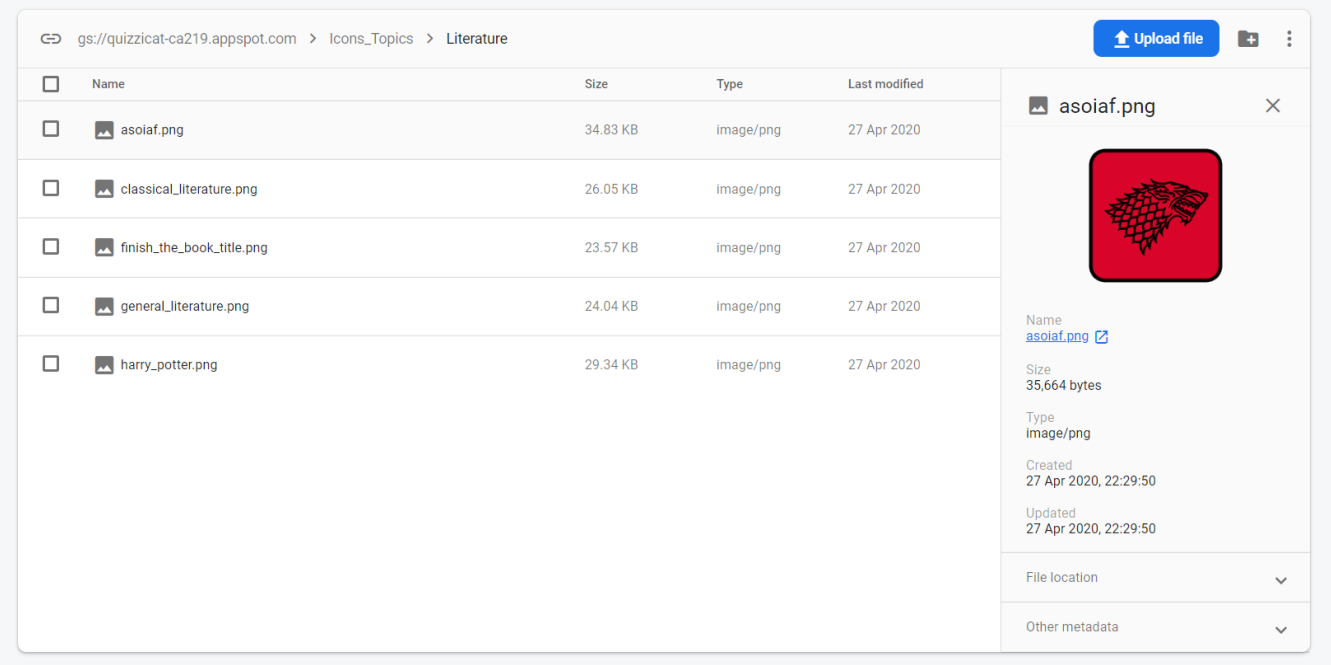

Figure 5.18: The Firestore interface, showing how some icons for History-related topics are saved

The library is imported into the application by adding the appropriate dependency in the Gradle module file.

To load the content of an image into an ImageView element, class ImageLoadingFacade, with method loadImageWithPicasso is used.

Algorithm 1 Image loading using Picasso

loadImageWithPicasso(url: String, imageView: ImageView)
@ Load the content of the image with the given URL into the ImageView element reference
Picasso.with(context).load(url).into(imageView)

5.3.5 MP Android Chart - displaying the user statistics using Pie Charts

One of the functionalities of the Quizzicat application is to display the user's most playing solo trivia games. In order for the visualization of this data to be aesthetically pleasing and interesting for the user, it is shown in two different pie charts, one for displaying the Categories played, and the other for the Topics. While the Topics pie chart initially shows the data for Topics belonging to all Categories, by selecting a slice from the Categories pie chart, the one displaying the Topic data will be updated to only show those topics from the selected category.

The objects used for displaying these Pie Charts belong to the **MPAndroidChart** library, which has been imported into the application through the module Gradle file. Two XML elements of type **PieChart** have been placed in the layout file corresponding to the statistics activity, and the objects

themselves are created in method *setupPieChart* of class *UserStatisticsActivity*. In this method, properties such as the colours of the slices, the size and colour of the text displayed on each slice and the animation of the pie chart are set.

In order to display only the Topics belonging to a certain Category when a pie chart is selected, listener *setOnChartValueSelectedListener* has been attached to the *categoriesPieChart* object. In order to use this listener, default methods *onNothingSelected* and *onValueSelected* need to be implemented. In this case, when nothing is selected, the Topics pie chart is created normally, with all the data for the played Topics. When a pie chart is selected, in the corresponding method, the entries of the pie chart data are filtered, and only those that have the same Category ID as the one selected are kept, and the Topics pie chart is refreshed.

Chapter 6

Implementation

The purpose of this chapter is to explain the logic behind the implementation of each main functionality, with its corresponding activities.

6.1 LoginActivity

If the user has a stable internet connection, the first interface that is displayed when launching the application would be the Login interface. Here, the user can log into an already created account with a validated e-mail address and password, or access an account created using a social media account - Facebook or Google.

The password associated with an account can also be reset, in case the user has forgotten it.

If the user wishes to create an account using a personal e-mail address, they can navigate to the RegisterActivity through the **"Sign-up now!"** link.

When the Activity is launched, it is checked whether the attribute *currentUser* is set for the current instance of the Firebase Authentication object, which would mean that an account was previously accessed on the current device. If so, the flow of the application is redirected to the MainMenuActivity.

For logging into an account, dedicated Firebase Authentication methods are used - **firebaseAuthWithGoogle** for signing in with a Google account, textbfhandleFacebookAccessToken and **signInWithCredential** for signing in with a Facebook account and **firebaseAuthWithEmailAndPassword** for for logging in using a validated e-mail address and its corresponding password.

For changing the password associated with an account, Firebase Auth method **sendPasswordResetEmail** is called, and an e-mail is sent to the given address, containing a link through which the user can reset their password.

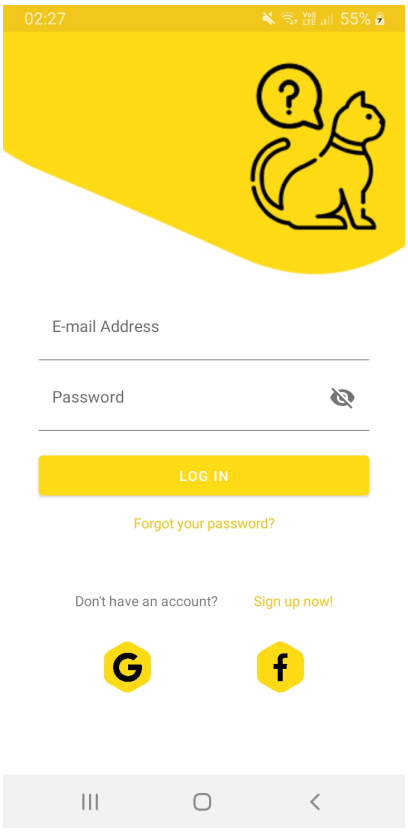

Figure 6.1: The interface of the LoginActivity

6.2 RegisterActivity

Through this activity, the user can create a new account using their personal e-mail address and a password. The user needs to provide the following information for the account creation - display name, e-mail address and password. Optionally, they can upload a profile picture from the mobile device storage. If no profile picture is uploaded, a default picture will be set for the account.

By default, the "Create Account" button is disabled, and remains so until the user checks that they have read and agree with the terms of service. The terms of service mention, among others, which data will be collected and how the security of the information is handled.

The image is selected from the device by starting an Activity with Intent of type **ACTION PICK**, and awaiting a return code of 0, that will be set when an image is selected from the storage. The selected picture is loaded into a CircleImageViewer element from the layout. The CircleImageViewer is an open source third party library for displaying images into a circular element, that can be installed through adding the corresponding dependency into the Gradle file.

The account is created by calling Firebase Auth method **createUserWithEmailAndPassword**. After the account is created, a verification e-mail is sent to the given address, containing a link through

which the user can validate their Quizzicat account. Validation management is done by Firebase Auth. Unless the user verifies their e-mail address, they can not access the newly created account.

If a profile picture was selected for the account, after the account was created it is uploaded to Firebase Storage using the method **putFile**.

Along the previously mentioned details regarding the user, the information about their location is collected, for recommendation purposes, using the method **getUserCountryDetails**. In this method, a GET request is made to the URL **http://ip-api.com/json**, which returns a JSON from which the city and country name are collected. The URL returns this information based on the public information exposed by the device to the internet network that it is connected to.

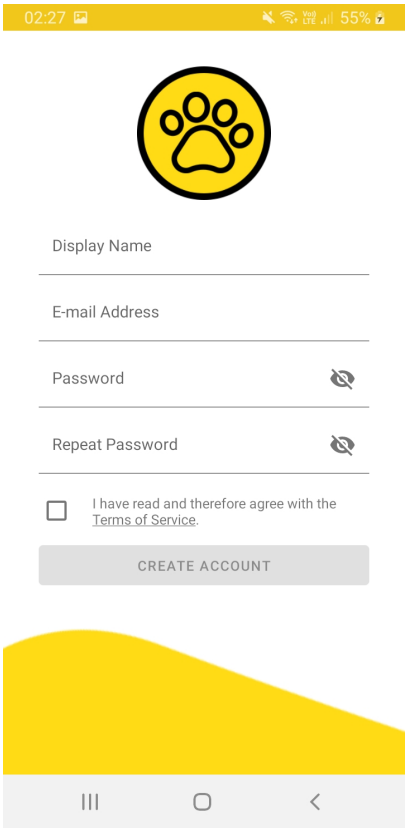

Figure 6.2: The interface of the RegisterActivity

6.3 NoInternetConnectionActivity

If no internet connection is detected when launching the application, the user will be redirected to NoInternetConnectionActivity, which displays a simple interface with some design elements (an ImageView, three TextView elements and a Button), letting the user know that a stable internet connection is required in order to access the application functionalities.

When pressing the **"Try Again"** button, method **isOnline** is called, which in turn calls the static method **getConnectivityStatus** of class **NetworkUtils**. In this method, the Connectivity Manager of the application is retrieved, and its method **getNetworkCapabilities** is used to return the properties of the network that the mobile device is currently connected to. A corresponding value based on whether the device is connected to a Wi-Fi, a mobile connection or to no network is returned.

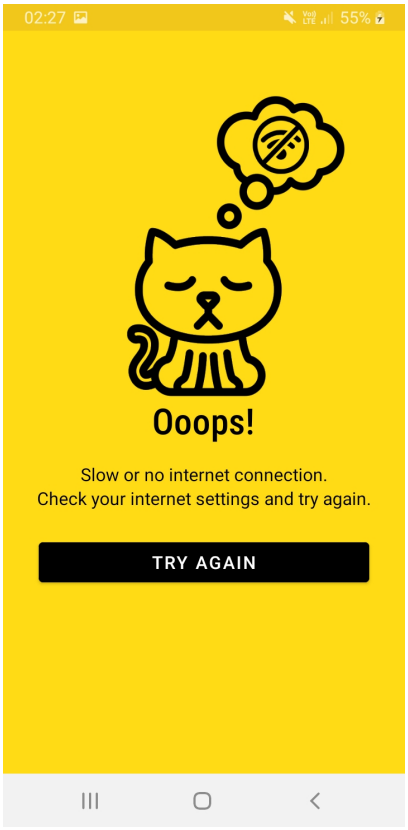

Figure 6.3: The interface of the NoInternetConnectionActivity

6.4 MainMenuActivity

The Main Menu activity represents the skeleton on which the main functionalities rest on. Its main purpose is to load the corresponding Fragment based on which tab the user is currently one. This is done using a ViewPager XML element and a custom FragmentStateAdapter, that returns the appropriate Fragment based on how the user swipes through the tabs.

6.4.1 TopicCategoriesFragment

The fragment that is selected by default at the start of the MainMenuActivity, class TopicCategories-Fragment allows the user to browse through the existing topic categories and their corresponding topics,

as well as to start playing a Solo Quiz after selecting a certain topic.

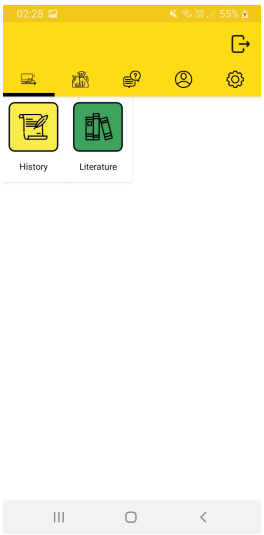

Figure 6.4: Topic Categories

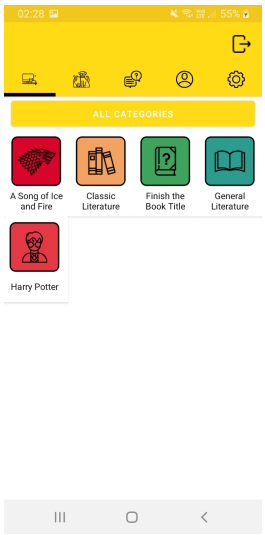

Figure 6.5: Topics corresponding to the Literature category

The topics and topic categories are displayed in a GridView, with a custom CardView for each item. The items are loaded into the container through the **TopicCategoriesAdapter**. The activity has an attribute *topicsLevel*, which is a Boolean variable indicating whether the items currently displayed are topics or topic categories. It is initially set to false, as by default the categories are displayed, and is then set to true when the user clicks on a category. At this event, the topics corresponding to this category are loaded in the same container, and the adapter is notified of the dataset change. This is possible since both **Topic** and **TopicCategory** inherit from **AbstractTopic** and have common fields, allowing the reuse of both the GridView and the adapter.

When the user clicks on a topic, an AlertDialog with a custom layout is shown, letting the user configure the Solo Quiz. From here, the user can choose the difficulty of the questions - Random, Easy, Medium, Hard - and the number of questions in the quiz. After pressing the confirmation button of the AlertDialog, the activity SoloQuizActvitiy is launched.

6.4.1.1 SoloQuizActivity

The customization options for the Solo Quiz are passed from the TopicCategoriesFragment to the SoloQuizActivity via the *Intent*, which is the router object that coordonates the navigation. The quiz difficulty, the number of questions, as well as the topic and category are set to the Intent by using method **putExtra()**.

In the SoloQuizActivity, all questions and answers matching the given category, topic and difficulty

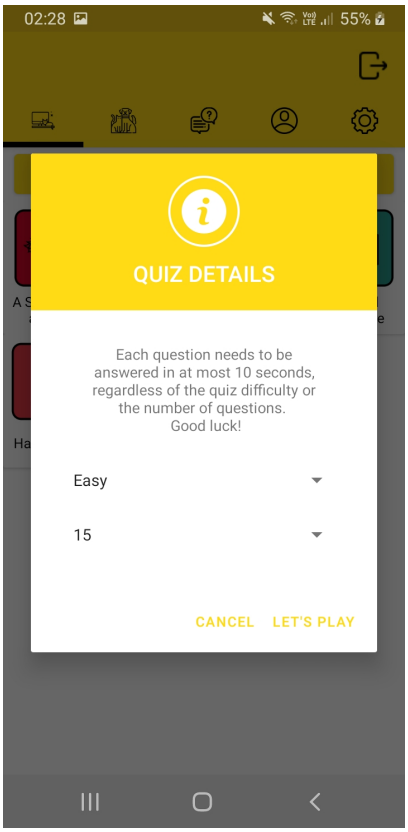

Figure 6.6: Configuring the Solo Quiz details

are fetched, after which questions are picked randomly from the retrieved list until the selected number of questions given by the user is reached.

Once the questions are fetched, the user has 10 seconds to answer each question, and failure to do so will result in the game being lost, in which case an appropriate message is displayed. It is important to note that the number of questions answered, either correctly or incorrectly, is still added to the user statistics, even if the player has ran out of time.

The timer used for each question is an object of type **CountDownTimer**, for which abstract methods **onTick** - handles what happens when a unit of time psses, in this case that is updating the TextView which displays the number of seconds left - and **onFinish** - which handles the processes after the timer has exhausted the given number of milliseconds - have been redefined.

When the user selects an answer and presses the "Next Question" button, their choice is highlighted either in green, for a correct answer, or with red, signifying an incorrect answer. After each answer, the counters for the number of correctly answered questions, respectively incorrectly answered questions, are incremented. To give the user enough time to look at the highlighted answer (and eventually re-read the question and figure out their mistake), the execution on the main thread of the activity is paused with four seconds after each answer. This is done by accessing the Runnable object of the

current execution thread via the constructor of class **Handler**, and then setting the actions to be performed after the four second delay by appending the method call **postDelayed**.

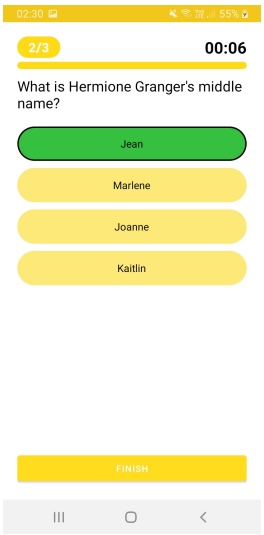

Figure 6.7: User has answered correctly

Figure 6.8: User has answered incorrectly

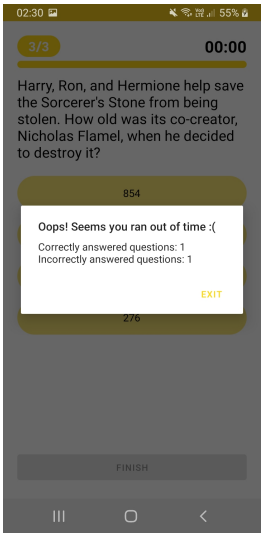

Figure 6.9: User has run out of time

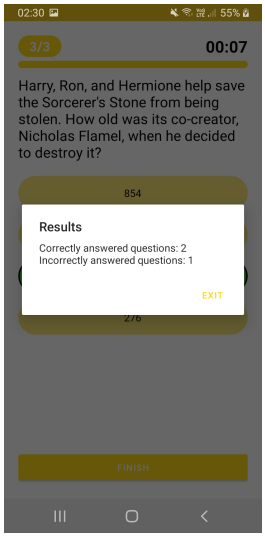

Figure 6.10: User has finished the quiz in time

6.4.2 MultiPlayerMenuFragment

When navigating to the second tab of the application, the MultiPlayer Menu is displayed, which consists of two buttons aligned in a LinearLayout, and a RecyclerView for displaying some data about the user's previously played games - the icon of the topic that the quiz was about, the username of the host, as well as the date in which the game was played and the number of points the current user has managed to obtain. If the user has managed to score the highest amount of points for that game,

a crown icon is displayed next to the number of points.

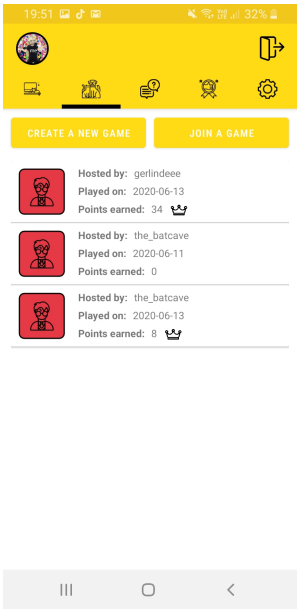

Figure 6.11: Multiplayer Trivia Games menu interface

Using the two buttons at the top of the menu interface, the user can choose to either create a new game, or to join an already existing one using a game PIN.

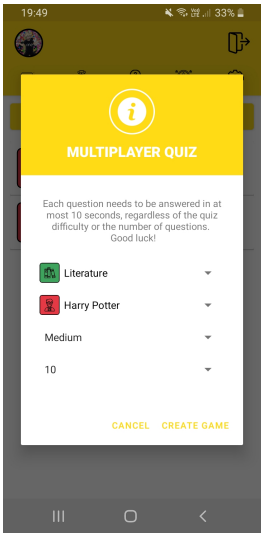

Figure 6.12: Creating a new multiplayer game

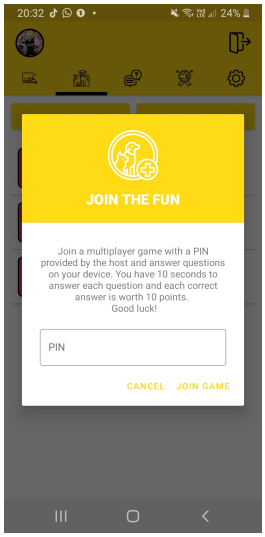

Figure 6.13: Joining an existing multiplayer game

When pressing the **Create a New Game** button, an AlertDialog with a custom view is displayed, allowing the user to customize the details of the game, in a similar manner with the solo trivia games, except that in this case, the user chooses the category and the topic from two Spinner objects, rather than a GridView. When nothing is selected for the Category Spinner, the *Visibility* property of the Topics Spinner is set to *"gone"*. Once the *onSelectedItem* listener of the Category spinner is notified

of the user selecting a category, the corresponding category ID is used to fetch all the corresponding topics and load the items into the Topics Spinner, and then changing its *Visibility* property to *"visible"*. By pressing the confirmation button of the AlertDialog, the Multiplayer Game entry is created and inserted into the corresponding database table by calling method **createMultiPlayerGame**, from the *MultiPlayerDataRetrievalFacade* class. This method also handles the creation of the unique PIN corresponding to the newly created game. To ensure the uniqueness of the randomly generated 4 digit PIN, all the multiplayer games having the *active* attribute set to *"true"* are fetched from the database, and a 4 digit number of randomly generated until a number that is not already assigned to an active game is found. The random generation of 4 digit numbers, along with the size of the number domain is large enough for this operation not to be too time-costly.

When the pressing the **Join a Game** button, another custom AlertDialog is displayed, containing a numerical EditText object, prompting the user to input the unique game PIN associated with a multiplayer trivia quiz, which is given out by the host. If the user inputs a 4 digit number that is not associated with any currently active multiplayer trivia quiz, a Toast is displayed letting them know that the value they have given is incorrect.

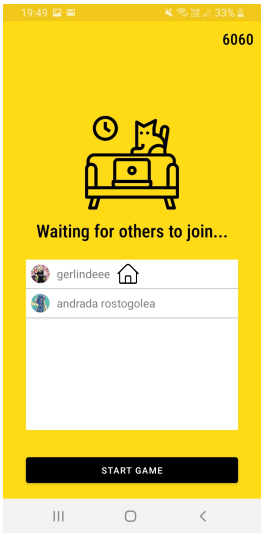

Figure 6.14: The Lobby, as seen by the host

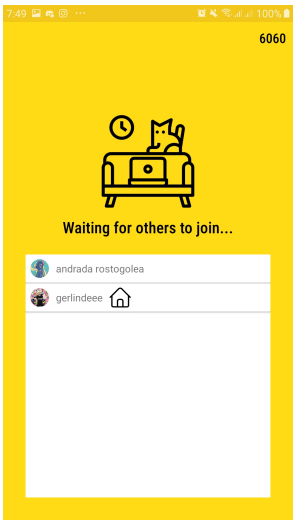

Figure 6.15: The Lobby, as seen by a player

6.4.2.1 MultiPlayerLobbyActivity

After the previous action, for both the host and the joining user, the **MultiPlayerLobbyActivity** is launched. The interface of this screen differs between the two users, depending on their role. For the host, a button is shown underneath the list of joined users, allowing the host to start the game when all players have joined the lobby. The user role is sent from the previous activity by attaching it to the

intent that launches Lobby activity. Since players and hosts launch the MultiPlayerLobbyActivity via different program flows - players join by giving the game pin while hosts are automatically redirected to the lobby after creating a game, the MultiPlayerMenuFragment is aware of the role that the user will have when the intent is created.

The name and profile picture of users who join is added to the RecyclerView in real time, and removed from the list if the user presses the back button. This is done by attaching a Snapshot listener to a reference pointing to the *Multi_Player_Users_Joined* database table, sending a notification each time a row is added, altered or removed. If the modification concerns the current game, the list of users is updated and a notification is sent to the RecyclerView adapter that the dataset has been changed.

Once all users have joined the game, the host can start the trivia game by pressing the **Start Game** button, which will launch the **MultiPlayerQuizActivity** for all users waiting in the Lobby. The simultaneous launch of this activity is also performed using a Snapshot Listener, this time coupled with a reference to the *Multi_Player_Games* table, which checks, once a modification to a row in the table is notified, that the *progress* field of the current game has been set to *"true"*.

6.4.2.2 MultiPlayerQuizActivity

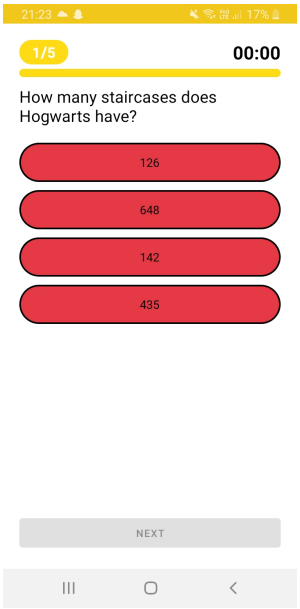

Figure 6.16: Users answers all questions incorrectly during a multiplayer game

The gameplay of the multiplayer trivia quizzes is almost identical to the one for the individual trivia games, except for the fact that when the time for answering a question has run out, the quiz does not stop, instead highlighting all the answers with red, and moving on to the next question. Additionally, while the individual trivia games are scored in the number of correctly answered questions,

the multiplayer version works on a point based system - the maximum number of points that can be awarded for a correctly answered question is 10, and the number decreases with each passing second. If the user does not answer the question correctly, the score for the current question is 0. After the current user answers all the questions in the quiz, their *score* is updated in the database table, along with the field *finished_playing*, which is then set to *"true"*, and then the **MultiPlayerScoreboardActivity** is launched.

6.4.2.3 MultiPlayerScoreboardActivity

The interface and behaviour of the MultiPlayerScoreboardActivity is similar to the one of the MultiPlayerLobbyActivity, but instead of waiting for users to join, the users are displayed as they finish answering all the questions, along with the total number of points that they have managed to accumulate. Similarly to the implementation of the Lobby, a Snapshot Listener is attached to the reference of the *Multi_Player_Users_Joined* collection from the database, but instead of handling additions or removals from the database, the observer is notified when an entry is changed, and then checks if the user row that was modified belongs to the current game and also if the field *finished_playing* has been set to true, in which case it adds the retrieved user to the list and notifies the Recyclver view of the dataset change. The user can leave the activity and return to the main menu by pressing the **Exit** button.

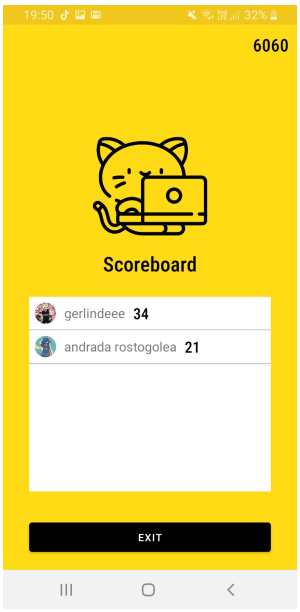

Figure 6.17: The multiplayer scoreboard interface

6.4.3 QuestionsLeaderboardFragment

In the Question Leadeboard interface, the currently logged in user can see questions submitted by other users, that are still waiting for approval. The user can only see the pending questions submitted by others, evaluate them using a 5-star rating system or report them if they believe the question or the corresponding answers to be incorrect, redundant or inappropriate. Both rating and report a question ca only be performed once - the Rating Bar becomes inactive once the user vote has been logged, and the message *Rated!* is displayed on the bottom on the CardView, and if the user chooses to Report the question, the question is removed from the RecyclerView.

If the number of votes that a question receives is equal or greater than 65% of the number of registered users and the average score of the question is equal or greater than 4, the question is removed from the *Pending_ Questions* table and instead added to the *Active_ Questions* collection, which contains the questions roster for the games. On the other hand, if the number of reports that a question receives becomes greater than 65% of the total number of users or the rating falls below 2 stars with the same number of votes, the question is removed from the *Pending_ Questions* table and added to the *Rejected_ Questions* collection.

The layout of the fragment contains two elements, a button labelled "Questions Factory" and a Recycler Viewer, which contains custom layouts for displaying the pending questions. When the user clicks on the icon corresponding to the topic of the question, the answer choices are displayed inside an AlertDialog, the correct one highlighted in green.

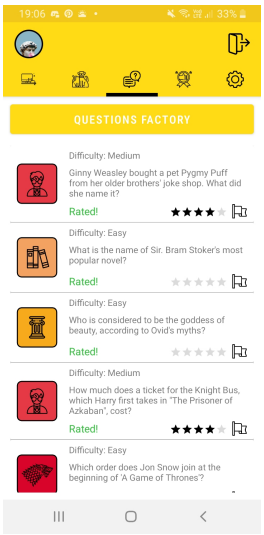

Figure 6.18: Question Leaderboard interface

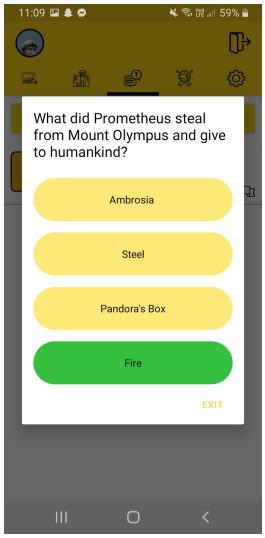

Figure 6.19: The answers corresponding to a pending question

As different users create new questions, or pending questions are either accepted or rejected, the

contents of the RecyclerView is updated with every addition and removal in real time. This is done by attaching a Snapshot Listener to the database collection that stores the pending questions, and for each notification of an addition or a removal, the list is updated accordingly and the RecyclerView adapter is notified of the dataset changes.

Clicking on the **Questions Factory** button will launch the QuestionsFactory activity, which contains four CardViews that act as buttons for different functionalities. From here, the user can choose to create a new question, display the submitted questions that have been approved by the community, those that are still pending, and those that have been rejected by fellow players.

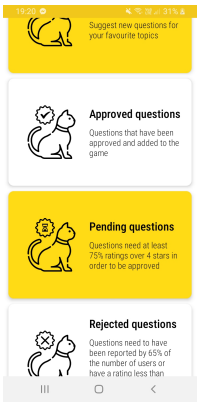

Figure 6.20: Questions Factory interface

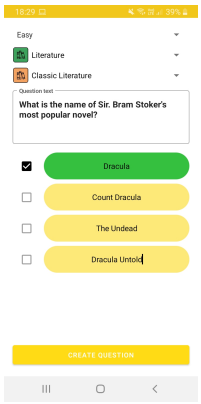

Figure 6.21: The interface for submitting questions

When creating a question, the user must specify its difficulty, the topic and its corresponding category, the text of the question, and the four answer variants, out of which one must be checked as being the correct one. Unless all these fields are specified, a custom exception of type EmptyFieldsException is throw and the submission is not accepted.

For a user's accepted, pending and rejected questions, separate Activities have been created, namely **UserQuestionsAcceptedActivity**, **UserQuestionsPendingActivity** and **UserQuestionsRejectedActivity**, each reusing the same XML layout that contains a RecyclerView, only with different adapters.

Accepted, pending and rejected questions are displayed similarly to the list of questions in the Questions Leaderboard, except without the Rating Bar and Report button. The answers to each question can also be visualized by pressing on the Topic icon ImageView, which opens an AlertDialog with a custom view, showing the text of the selected question and each of the four answers, the correct one being highlighted in green.

From here, a user can choose to remove any question that they have submitted, after pressing on the confirmation button of the created AlertDialog, which will trigger the removal of both the question from the Pending_Questions table, and the corresponding answer choices from the Pending_Question_Answers one.

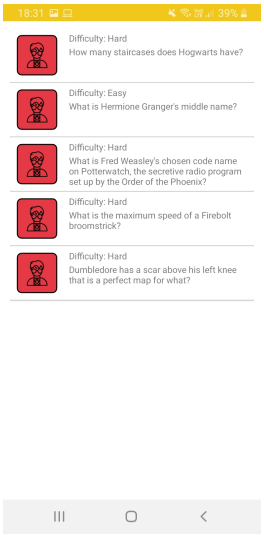

Figure 6.22: Accepted questions for a user

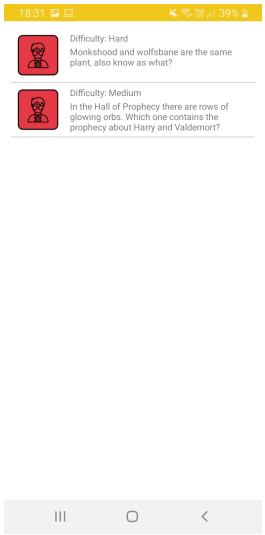

Figure 6.23: Rejected questions for a user

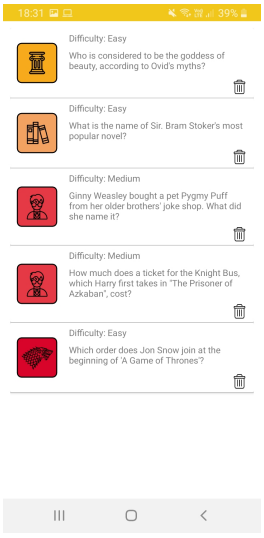

Figure 6.24: Pending questions for a user

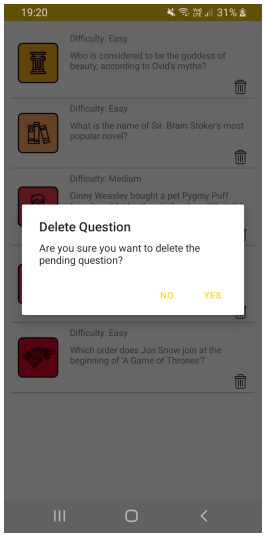

Figure 6.25: Question removal confirmation

6.4.4 RecommendationsFragment

In the second to last Fragment of the navigation bar, the user can see recommendations for Topics to play in individual quizzes based on their playing history. The layout of the Fragment contains four horizontal RecyclerViewer elements, which loads and maintains the elements using the **RecommendedTopicsAdapter** as the adapter.

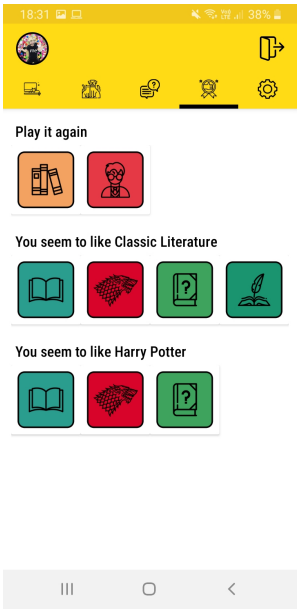

Figure 6.26: The Recommendations interface

The first RecyclerView displays all the Topics that the user has played at least once, sorted descending by the number of times the topic has been played in an individual trivia game. Fetching the data is performed using method *getTopicsPlayedData* of the *TopicsDataRetrievalFacade* class.

The next three RecyclerView elements are dependent on the result of the the first one. Each one is meant to display topics related to the top three most played topics of the current user. If the user has less than three played topics, then a corresponding number of recommendation RecyclerViewers are displayed, which is done by changing the *visibility* property between *"visible"* and *"gone"*.

Similar topics are recommended based on criteria-based filtering, which compares the topics according to some predefined traits, which are stored in the **Topics_Comparison_Values** collection. For each existing topic, the following fields receive a Boolean value, to signify if the criteria is met or not: **fiction**, **general_knowledge**, **humanities**, **pop_culture**, **real_life** and **science**. After retrieving these comparison values from the database, each criteria of the current topic is compared against the rest, and for each field where there is a value match, the value of a similarity index is increased. In order for a topic to be considered related enough to another one, the similarity index

need to be at least 3. In the recommendations screen, all the topics similar to each topic from the top three most played ones is retrieved and loaded into the corresponding RecyclerView.

Clicking on any one of the topic icons from the lists displayed in the recommendations interface launches the solo quiz behaviour, opening an AlertDialog interface from which the user can customize the quiz details, and afterward play an individual trivia game by clicking on the positive button of the dialog window.

6.4.5 SettingsFragment

From the rightmost tab of the main menu, the currently logged in user can change their account information and password, delete their playing history or delete their Quizzicat account.

When choosing to change their account information, activity **ChangeUserProfileActivity** is launched, which contains a gallery image browser in which the current profile picture is displayed, stored in the Firebase Storage cloud, and two EditText fields, one for the display name and the other for the e-mail address. By default, the input for these fields is disabled. Once the input is enabled, the visibility for another EditText and a Button for saving the changes is set from *GONE* to *VISIBLE*. Their purpose is for the user to confirm the changes done to their account information by providing the password associated with the currently logged in account. This check is done by fetching the AuthCredential object associated with the current FirebaseAuth user, and attempting to call method **reauthenticate**, passing the credentials as the parameter. If the reauthentication is successful, then the password provided in the EditText is correct. Otherwise, a Toast is created and displayed, letting the user know that the provided password is incorrect. If the user disables editing for the input fields without modifying the content, the visibility of the previously mentioned fields is set back to *GONE*.

Besides the display name and profile picture, the user can change the e-mail address associated with the account, for accounts created with a personal address. The user is not allowed to change the address associated for accounts created using social media profiles, as the Google/Facebook security credential association for the application would no longer work. When the personal address is changed, the user needs to validate their account once more, via a link sent to the new e-mail address, the same way it was done for the creation of the account.

The password reset is performed in the same way as for the LoginActivity, via the FirebaseAuth method **sendPasswordResetEmail**. Before changing the password, the user is asked to confirm the e-mail address associated with the account, for security purposes.

When the user chooses to delete their account, an AlertDialog with a custom layout is displayed, letting the user know that their authentication information, as well as their pending and rejected

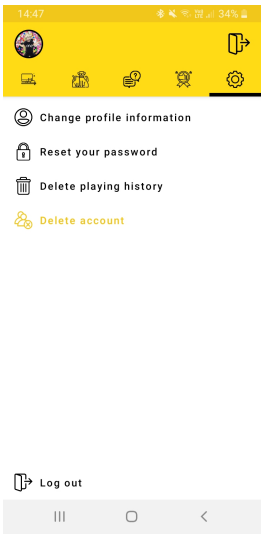

Figure 6.27: Account Settings interface

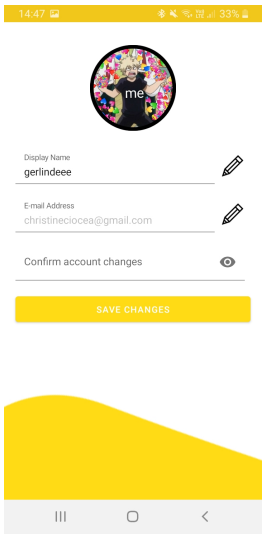

Figure 6.28: Changing the account information

submitted questions and playing history will be deleted. Once the user confirms the deletion, the data removal process is performed in the following order: first, the pending and rejected questions are deleted from the respective tables in the database, then the profile picture is removed from the Firebase Storage space (unless the user has no profile picture selected, and thus their avatar is a default image), then the information is deleted from the Users database table and finally, the Firebase Auth credential is destroyed, which needs to be the last step performed. If all the deletion steps are performed without errors, the user is logged out and redirected to the LoginActivity.

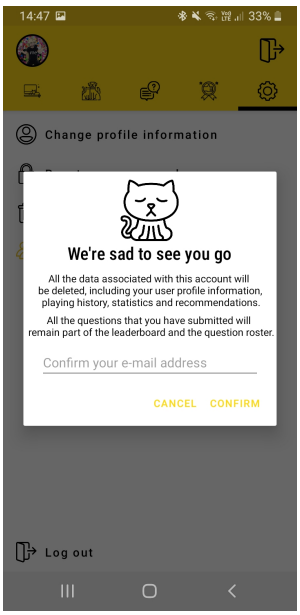

Figure 6.29: Confirmation of account deletion

6.4.6 UserStatisticsActivity

By pressing on the profile picture displayed in the upper left corner of the main menu screen, the user can navigate to the statistics screen, launched by navigated to the UserStatisticsActivity, which contains some statistics regarding the playing history of the currently logged in user, focusing on solo games.

If the user currently has no playing history, either due to the account being newly created or having just deleted their game history, a corresponding message is displayed.

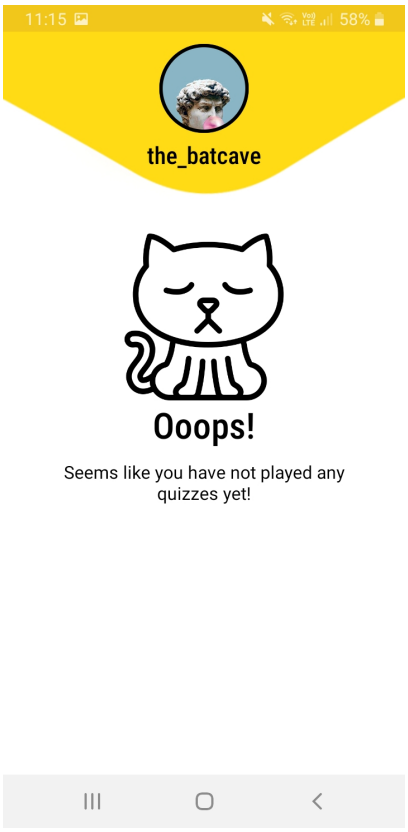

Figure 6.30: No user statistics to display

Otherwise, the following information will be displayed: the total amount of games played, the number of correctly and incorrectly answered questions, a ratio bar for visualizing the amount of correct answers against the number of wrong answers, as well as two Pie Charts representing the number of games played per category and topic. When selecting a slice from the Categories Pie Chart, the Topics Pie chart is updated to only display the topics belonging to the selected category.

The bar displaying the ratio between correctly and incorrectly answered questions is implemented using a Progress Bar with a custom progress drawable, that is red at one end and green at the other. The ratio is set in the method **createCorrectIncorrectBar** of the UserStatisticsActivity

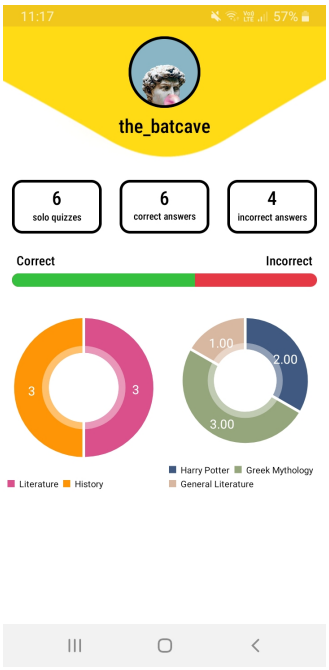

Figure 6.31: Category Pie Chart slice not selected

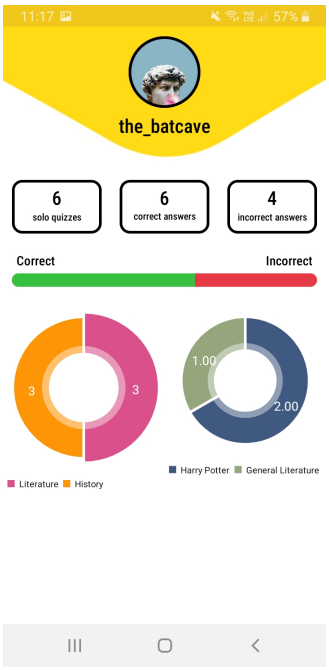

Figure 6.32: Category Pie Chart selected

class, computing the total amount of answered questions, which is then assigned to the ProgressBar *max* attribute. Finally, the total number of correctly answered questions is assigned to the ProgressBar *progress* attribute.

The Topics and Topic Categories Pie Charts are objects imported through the open source library MP Android Charts, which provides various types of charts for Android Applications. The entries for the charts need to be of the form (Label, Number), where Label would be the name of the Topic or Topic Category, and Number the amount of times a game from the given topic has been played. The input arrays for the Pie Charts are created in methods **createTopicsPieChart** and **createCategoriesPieChart**. Both methods end up calling **setupPieChart**, which customizes the appearance of the charts, setting properties such as slices colours, animation, value font size and colours and enabling the legend for the chart.

Chapter 7

Conclusions and Future Work

In conclusion, the "Quizzicat" application manages to provide an easy to use and entertaining application for trivia enthusiasts, allowing users to play quizzes about their favourite topics by themselves or with any number of other players, to share their knowledge of the subject matter by submitting their own questions for the game questions roster, viewing statistics about their playing history, and getting recommendations based on their most played topics.

Comparing the application to similar already existing ones, "Quizzicat" brings together the base functionalities of each - it has individual trivia quizzes like QuizUp, a way for users to submit their own questions like Trivia Crack, and a multiplayer functionality like Kahoot! - while implementing them in a customized manner, for example, when creating a multiplayer trivia quiz, the host does not need to create the questions themselves and is actually not at all aware of which particular questions are going to be selected for the quiz, allowing the host user a chance to play, unlike with a Kahoot! game, where the host is only a moderator and an observer, and adding new functionalities, such as receiving recommendations tailor for each user based on their playing history, unlike for QuizUp!, which recommends topics based on their recent general popularity among the user base.

Still, the application could be extended in the future, possibly by adding a way for users to submit ideas for topics and categories, instead of just questions, the recommendation system can always be refined by adding more criteria for filtering or it can even be replaced by a collaborative filtering engine, which would recommend to the user topics that other players, having similar favourite topics, are playing.

Bibliography

[1] Pew Research Center. Mobile Ownership Over Time, 2019.

[2] Google Developers. App Manifest Overview, 2020.

[3] Google Developers. Configure your build, 2020.

[4] Google Developers. Layouts, 2020.

[5] Google Developers. Introduction to Activities, 2020.

[6] Google Developers. App resources overview, 2020.

[7] Google Developers. Meet Android Studio, 2020.

[8] Google Developers. Get Started with Kotlin on Android, 2020.

[9] Google Developers. Projects overview, 2020.

[10] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. Design patterns: Elements of reusable object-oriented software, 1994.

[11] Firebase. Authenticate with Firebase using Password-Based Accounts on Android, 2020.

[12] GeeksforGeeks. Introduction to Kotlin, 2020.

[13] Glu. The QuizUp Application on Google Playstore, 2020.

[14] Mobile Security Testing Guide. Mobile App Authentication Architectures, 2020.

[15] Anna Heim. How Latin American mobile game Trivia Crack conquered the US market, 2014.

[16] Martin Heller. What is Kotlin? The Java alternative explained, 2020.

[17] Kahoot! About, 2013.

[18] Kahoot! What browsers work with Kahoot!?, 2020.

[19] Maya Kosoff. I Played The Super-Addictive Game Trivia Crack, And Now I Know Why It's Crushing It In The App Store, 2014.

[20] The K!rew. New year, new heights: Kahoot! grows by 75% to reach 70 million unique users, 2018.

[21] Frederic Lardinois. Kotlin is now Google's preferred language for Android app development, 2020.

[22] Classification of patterns. Classification of patterns, 2020.

[23] RefactoringGuru. What's a design pattern?, 2020.

[24] RefactoringGuru. Why should I learn patterns?, 2020.

[25] RefactoringGuru. Criticism of patterns, 2020.

[26] RefactoringGuru. Factory Method, 2020.

[27] RefactoringGuru. Builder, 2020.

[28] RefactoringGuru. Prototype, 2020.

[29] RefactoringGuru. Adapter, also known as Wrapper, 2020.

[30] RefactoringGuru. Bridge, 2020.

[31] RefactoringGuru. Flyweight, 2020.

[32] RefactoringGuru. Proxy, 2020.

[33] RefactoringGuru. Command, 2020.

[34] RefactoringGuru. Iterator, 2020.

[35] RefactoringGuru. Memento, 2020.

[36] RefactoringGuru. Oberver, 2020.

[37] RefactoringGuru. State, 2020.

[38] RefactoringGuru. Strategy, 2020.

[39] RefactoringGuru. Template, 2020.

[40] RefactoringGuru. Visitor, 2020.

[41] Katie Roof. QuizUp Debuts On Android; How An Icelanding Game Became An Overnight Success, 2014.

[42] Alexey Soshin. Hands-on design patterns with kotlin: Gof, reactive patterns, concurrent patterns and more, 2018.

[43] Daniel Terdiman. No, Flappy Bird developer didn't give up on $50,000 a day.

[44] TriviaCrack. Who We Are, 2020.

[45] w3sDesign. GoF Design Patterns Reference, 2014-2018.

[46] Rachel Weber. QuizUp's Thor Fridriksson: The man with all the answers, 2014.

[47] Kyrus Keenan Westcott. Breadking Down Those Weird Trivia Crack Characters, 2014.

[48] Leah Yamshon. You Should Play: QuizUp, 2014.