# System Design Document for KahIt

**Working Group:** Anas Alkoutli
Oussama Anadani
Mats Cedervall
Johan Ek
Jakob Ewerstrand

25 October 2019
Version 1.0

# Contents

# 1 Introduction

*KahIt* is an application developed for the android operative system and is meant to be run on either phones or tablets. It is a quiz application that is closer to a party game. It can be played with friends in a multiplayer mode, by one player hosting or locally on one device. When hosting a game friends can connect using *Nearby Connections* with up to 8 players. *KahIt* is designed to have items that can boost the player's score or reduce it. *KahIt* provides a verity of cosmetic items that do not affect players status but are shown for other players players.

## 1.1 Definitions, acronyms, and abbreviations

- Hotswap: A game mode were a multiplayer game is played on one device by rotating the active user and letting them preform one action per rotation, resulting in a multiplayer experience that only requires one device.
- MVVM : Model view view-model
- XML : Extensible markup language.
- STAN : A dependency analisys tool.
- JUnit : A java unit testing framwork.
- Bus : A library that enhances loose coupling since it uses a publisher/subscriber pattern. Bus can also be mentioned as event bus in this document.
- Map : A component in Java that acts similarly to a list since it holds multiple values. A map uses a key to get the wanted value instead of an index

# 2 System architecture

This section will present the system architecture of the application. It includes what components the application uses, how it uses them and how they work together

## 2.1 Android basics

Kahit is an andriod application which has been built and tested using Android Studio IDE. It uses SDK28 which is Android Pie but has support down to SDK24. SDK28 was chosen because it was easy to build the application with and test it on our own devices.

This application uses Gradle which is an open-source build automation tool. Gradle makes it easier to download and configure dependencies or other libraries.

## 2.2 Application components

Kahit is built with MVVM structure, which enhances separation of concern. This structure is similar too the MVC design pattern but a controller is replaced with a view model. The controller holds a dependency on the model in a standard MVC structure but in MVVM structure a view model depends on a repository which depends on the model. This section will present the components of the application and provide an overview on how they depend on each other.

### 2.2.1 Model

This component holds the data of the application. This is also called the domain layer since it represents the domain model and it should be invisible to the user and separated from other components such as the view-model and the view In other words this component holds the data needed for the application to work but not how it should behave

### 2.2.2 View-model

This component falls under the presentation layer of the application. As the name suggests it as a component between the view and the model therefore it contains all the logic needed for the view to work but also gets all the data needed from the model.

The View-model acts as a converter that gets data from the model and sends it to the view in way it can handle the data. It can also work the other way around where the model can be manipulated by it if an event is triggered in the view for example.

### 2.2.3 View

Similarly to the View-model falls the View component under the presentation layer. This component is the only component the user directly interacts with. The view's only purpose is to present data to the user but does not hold or handle data within itself. The View can manage the user inputs such as clicks and gestures which can trigger events that the View-model can act upon

### 2.2.4 Application events

Application events holds all the necessary events for the application to function. An event acts as a switch where the application takes action as soon as it is flipped. An example of this would be waiting for all players to be ready. When all players are ready an event will be sent on the event bus which notifies the components that are listening for this event that all players are ready and the game can start.

### 2.2.5 Repository

The repository package functions as a connection point between the network, the domain layer which is the model and the presentation layer which is the view and the model. This package has a singleton class the can be accessed from the view model to get information from the model send information to it. In this case the view model does not directly depend on the model and the repository makes easy to maintain the connection with the network manager.

### 2.2.6 Music services

This package handles all the audio related issues in the application. The user can choose to run the application with or without music. The application provides various tracks for the different categories.

### 2.2.7 Database services

The database services connect the program to a Firebase realtime database and is used to fetch the data required to instance the questions and items that are used by the rest of the program. This component is used by the different factory classes in the program as well as a number of the views.

### 2.2.8 Network manager

The Network manager component contains all of the details surrounding the game network communication. It cares for the setup and operation of the Nearby Connections API integration. It provides methods for the host to transmit model state changes to its clients. It also supplies several callback interfaces to the repository,

used to update the model and the general state of the app according to what is suitable to the state of the host.

When a host makes any action that results in changes within the model, then the host also broadcasts the change through the Nearby Connections API. Meanwhile clients are only allowed to request changes to be made by the host and always update their own model based on what the hosts broadcasts.

### 2.3 Application flow

A normal flow of the application can be described by explaining a simple task such as answering a question. When the game is started by the host an application event is sent to the network manager which sends it to all players. The application loads the question from the database and sends it as an event. The question view model loads the event and extracts the need information for the view.

After the question is displayed and the player has answered the time and answer are sent to the player's model through the repository to calculate and update the player's score. Since the player has answered a question, the repository sends an event to the host that the player is ready for the next question. Through the whole application flow will the music services be playing different tracks for different situations such as category related tracks or waiting tracks while waiting for the game to start.

## 3 System design

### 3.1 Model view view-model: MVVM

Instead of implementing a standard MVC pattern the application uses a Model view view-model (MVVM) which is one of the standard solutions to facilitate communication between the different parts of an Android application. In MVVM each of the views (xml files) of the application has a view class that handles the input of the user and appearance of the view, and a view-model class that holds all data for the view and performs operations on the data and updates the view. To facilitate the communication between the view and view-model, the view hold a reference to it's view-model and calls method in it. While the view-model uses live-data to update the view. Live-data is variables that are synchronised between classes, if the data is updated in the view-model a listener method is called in all classes the listen to the data in this case the view. The view-model fetches it's data by either using method calls or by getting it pushed by listeners or events through a event-bus.

### 3.2 Package structure

Figure 1 shows the structure of the packages in the application.

- View package provides classes that expose basic user interface classes that handle screen layout and interaction with the user.
- The ViewModel is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel allows data to survive configuration changes such as screen rotations. Architecture Components in Android provides ViewModel helper class for the UI controller that is responsible for preparing data for the UI(view). ViewModel objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance(View). For example, if you need to display a list of users in your app, make sure to assign responsibility to acquire and keep the list of users to a ViewModel, instead of an activity or fragment(View parts)
- Same figure shows that Repository-package is the main core of the whole application. Basically, repository package is the business Layer. it holds everything related to the application's Domain. It uses a lot of business objects and creates a lot of new objects, too. It knows that all these objects need to be persisted or retrieved from some storage. For instance, the repository has like these shelves where it puts all the new objects on. And everytime it needs to work with an object it just gets it from the shelf, does the work then puts it back. The repository doesn't know how these shelves work and how they're actually storing or retrieving the objects. It does only know that it just puts the business objects there and it gets them back exactly how they were.
- Model package holds all the data and the business that will be provided in the app. It depends on non of the packages because its the logical data that could be changed over the time and provided to different work environments.
- Database package provides all data to model-package which means it only depends on the model.
- View package depends on ViewModel to provide all the changes that happens when the user interacts with UI(screen). In additional, ViewModel is a path to the other packages and to the Repository package that in it's turn is a path to get the data form the model.
- NetworkModel package holds the all classes that provide network business that in turn has no dependency on other packages.
- BackgroundMusicService holds the music class that provides music as a service in the background of the application.
- ApplicationEvents contains plain Java Objects that are used as events on the GreenRobotEventBus.

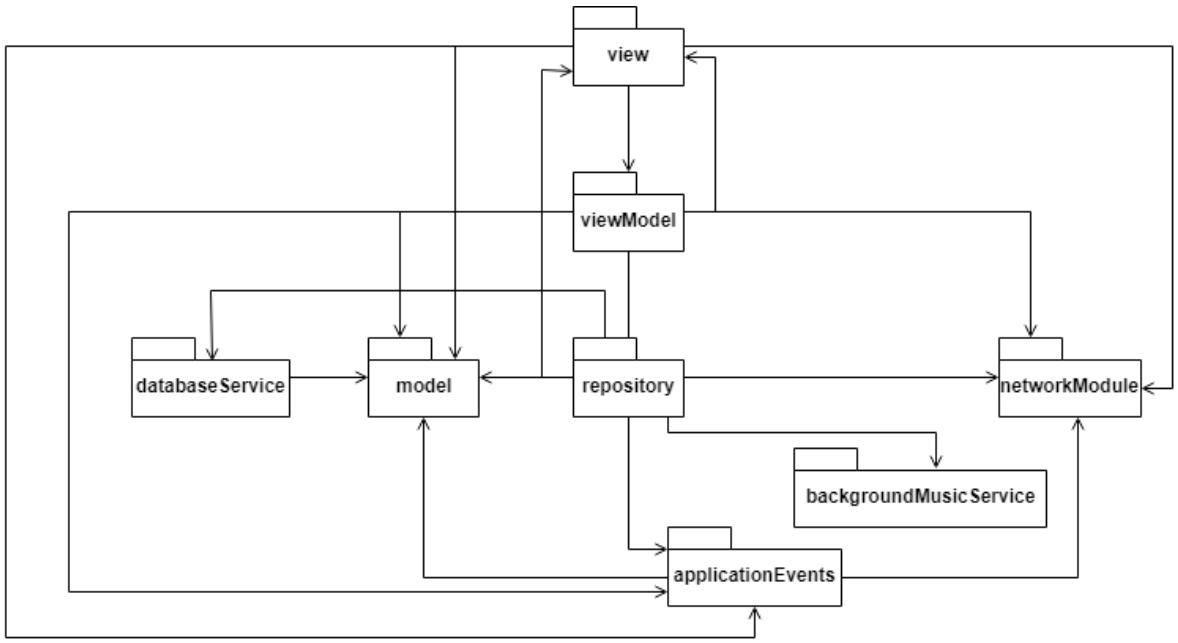


Figure 1: High-level package view

## 3.3 Model - Design model vs Domain model

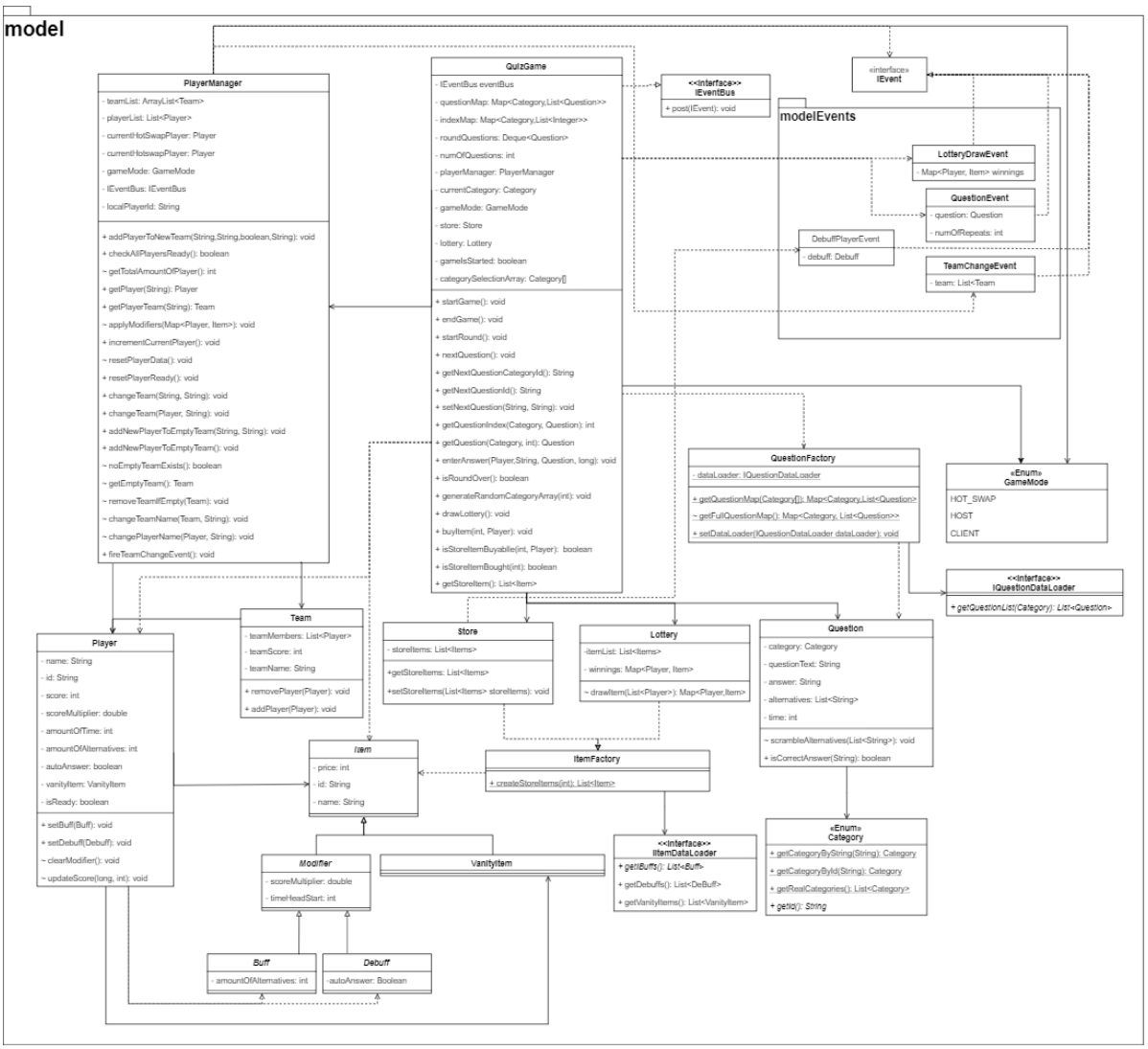


Figure 2: Design model

## 3.4 Database Service

The package DatabaseService is responsible for the connection to the Firebase Realtime database and the fetching of data from it. To accomplish this responsibility it has two data loader classes that connect to the database and fetch the question and item data and then creates their respective objects. The package also has data holder classes, these classes mimic the structure of the database and are used to ”cast” a part of the data structure (such as the data for a question) into a object. These classes also have methods that create questions or items out of the data that they hold. The reason

for why these ”middle man” objects are used is that the data in the database does not completely match the data in questions or items, for an example the database hold the name of the images associated with each of the items but the items do not hold their image names. So if the data from the database were to be cast directly into a item the image name would be lost.

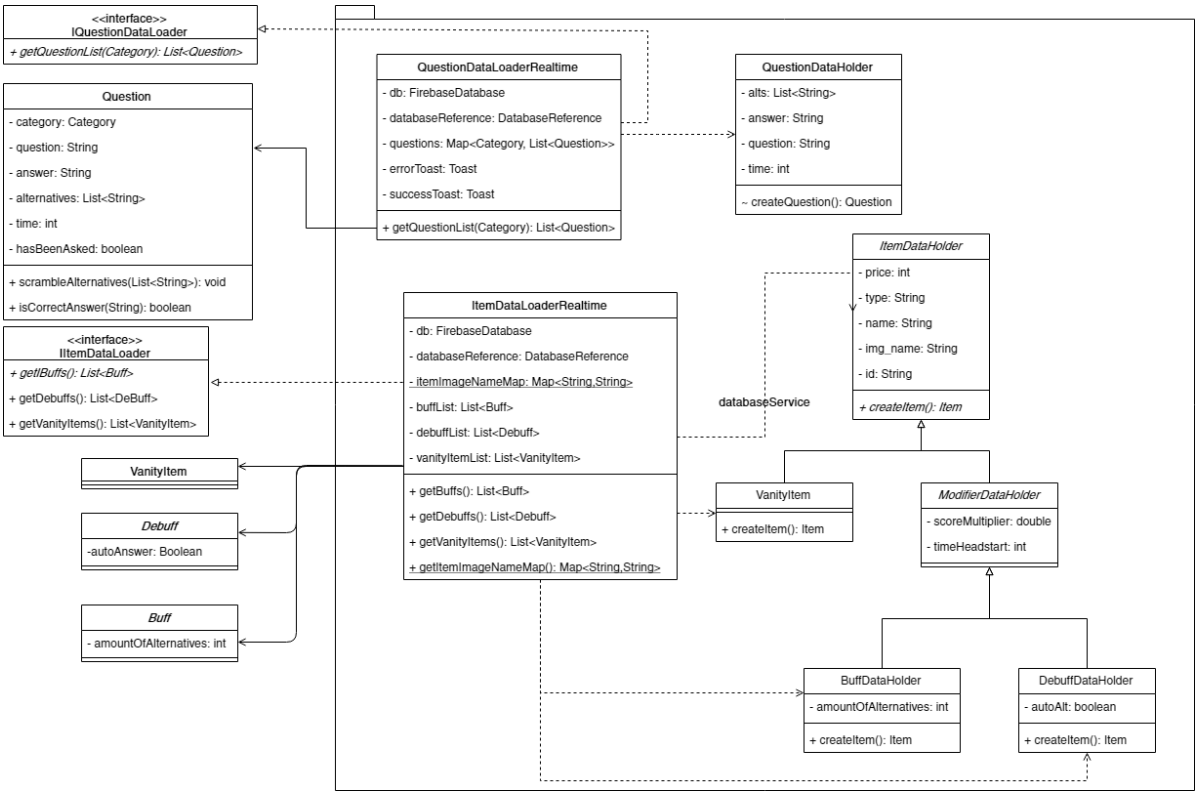


Figure 3: Database service package

## 3.5 Network manager

The NetworkManager package comprises of mainly two classes, the NetworkModule and the PacketHandler. The NetworkModule cares for the low level Nearby Connection API integration, handling the details relating to the setup, transmission and reception of byte-array payloads to and from other devices. The PacketHandler holds the high level package send and receive logic, responsible for the construction and deconstruction of network packets.

The network protocol consists of a single source of truth, the host, and several clients that may request changes to be made by the host. If the host complies with the request it updates its own model accordingly and broadcasts the resulting model change to all of the connected clients. Thus the host is always in control of the flow of the game,

strictly limiting the cheating capabilities of a malicious client.

The protocol relies on unique identifiers for each of the connection endpoints. The unique id is used by Nearby Connections to distinguish all of the different endpoints, to ensure that a packet is sent to the intended device. Furthermore, the id is also used by the client to determine if a received packet is affecting their local player. If that is the case, then the client updates the view to inform the user that their action has been confirmed by the server and is now part of the new game state.

Due to a limitation within the Nearby Connections API, the local device may never know their own unique identifier. To circumvent this issue, upon every established connection the peers exchanges each others id.

Efforts has been made to provide the repository with a clean interface to work with. Given the intricacy of the communication required to convey all the possible network and model changes, certain complexity still remains. This requires large interfaces for supplying sufficient callback methods to cover all the use cases, resulting in high complexity in the repository. That due to time restrictions could not be refactored.

Potential solutions to reduce the complexity of the repository includes splitting it into two parts, a modelRepository and a new NetRepository. The main improvement comes from extracting the network component to a separate repository. The refactor would improve the readability and separation of concern inside the repository package, but also increase the complexity due to a larger amount of classes.

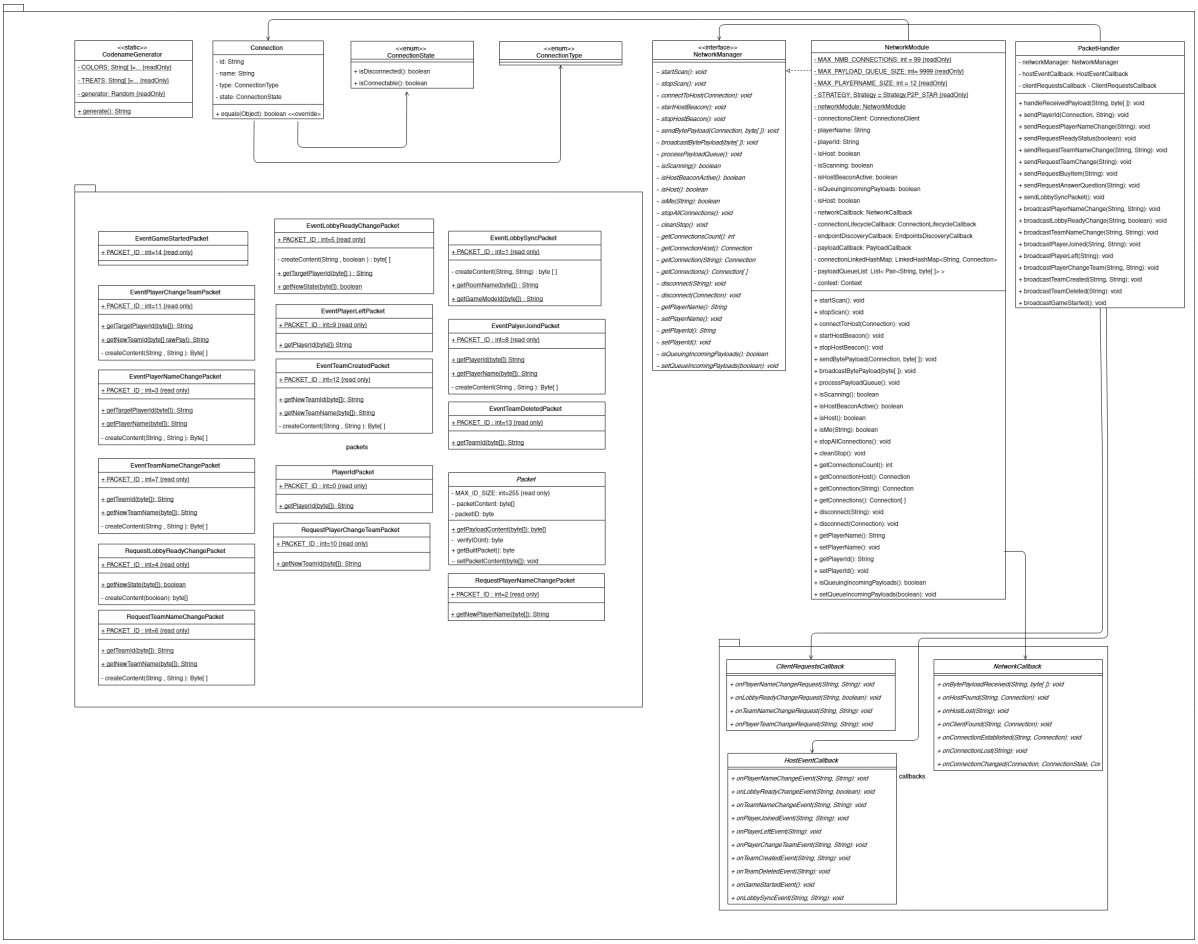


Figure 4: Network module package

## 3.6 View

The View in the used MVVM-pattern represents the UI of the application just like the more common MVC-pattern. Therefore it should be devoid of any logic. In the application each View consists of a given Activity or Fragment and the different views should only route the user's action to it's specific ViewModel. Each View only observes the given ViewModel for that specific View(within it's lifecycle) using LiveData. LiveData provided many benefits compared to normal observers such that it is lifecycle-aware, it can only be observed in the given context of that Activity's or Fragments components. LiveData therefore prevents memory leaks and it prevents crashes due to activities being destroyed.

The applications different Views inflates it's components from a specific xml-file. The different child-views specific Id's are then used to setup onClick- or onTouchListeners that are then used to notify the ViewModel using normal dependency. This naturally

makes the View dependent on the ViewModel but not the other way around. In some cases the components are dynamically generated instead of specifying them in an xml-file simply because it proved to be easier given that the View varies a lot depending on the amount of players. The LotteryView uses such a solution.

In some instances views do call on the repository directly. This is not optimal but it is mainly related to the networks current implementation and would have been addressed if time was given.

## 3.7 ViewModel

The ViewModel's responsibility is to manage and prepare data for the View while also handling the communication with the model. In the application the ViewModel is only created with an activity in it's scope and it will persist after that view. When a new View instance is created the viewModel will reconnect with it's owner(view). The use of viewModels in the application reduces the amount of logic in the model that is needed in order to prepare the data for the View the ViewModel can now performs that task.

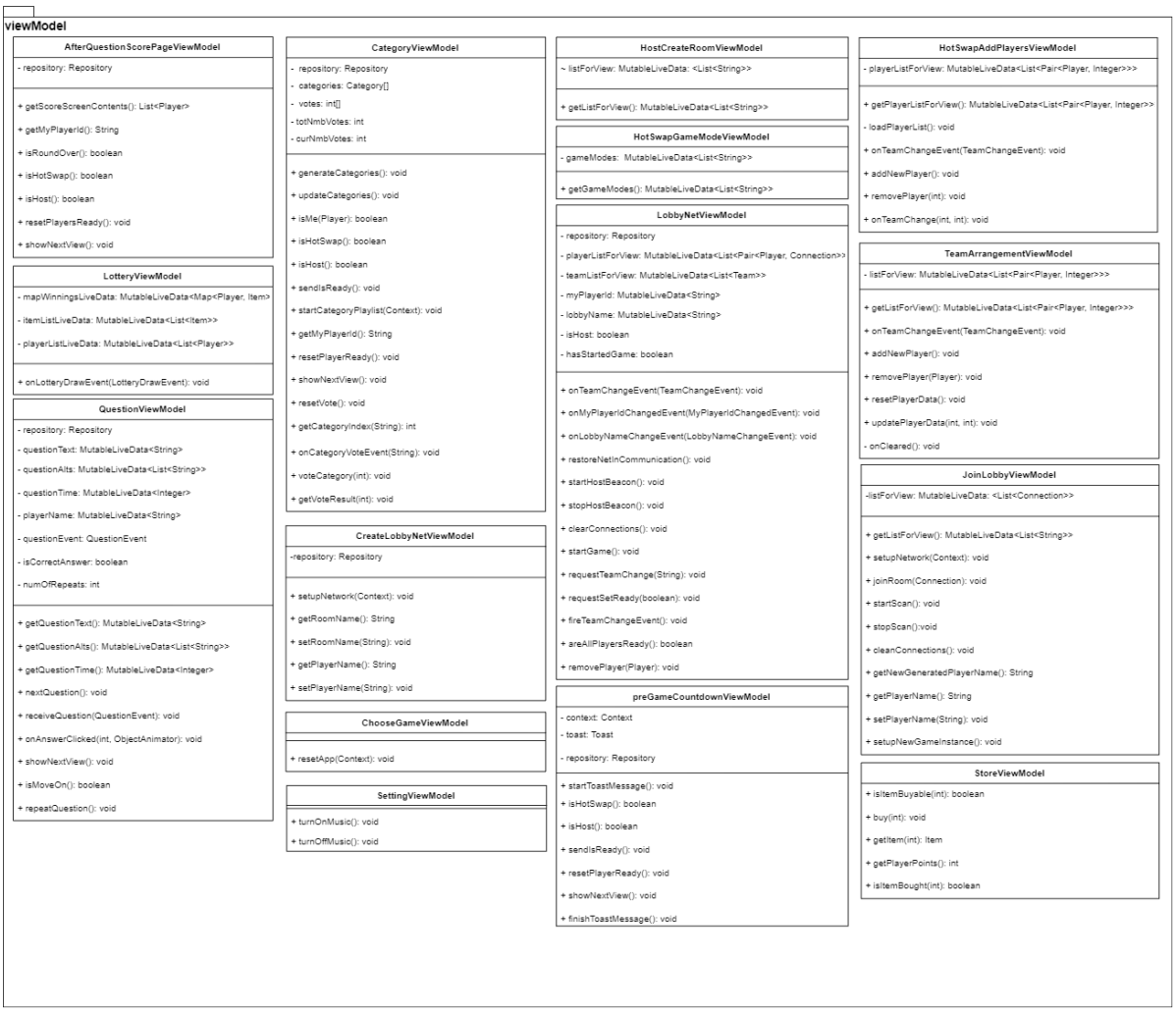


Figure 5: ViewModel package

# 4 Persistent data management

## 4.1 Question database - Firebase Realtime database

The questions and their related data is stored in a Firebase Realtime database. The database is structured in the following way: All categories are children of the object question and all questions are children of their respective category. Each question has the children question, answer, time, alts and id(see figure 6). Question, answer and id hold string values, the question, answer and id of the question respectively. Time holds a long value, the time that the question will be displayed to the user. And alts holds a list of string objects, the alternative answers to the question.

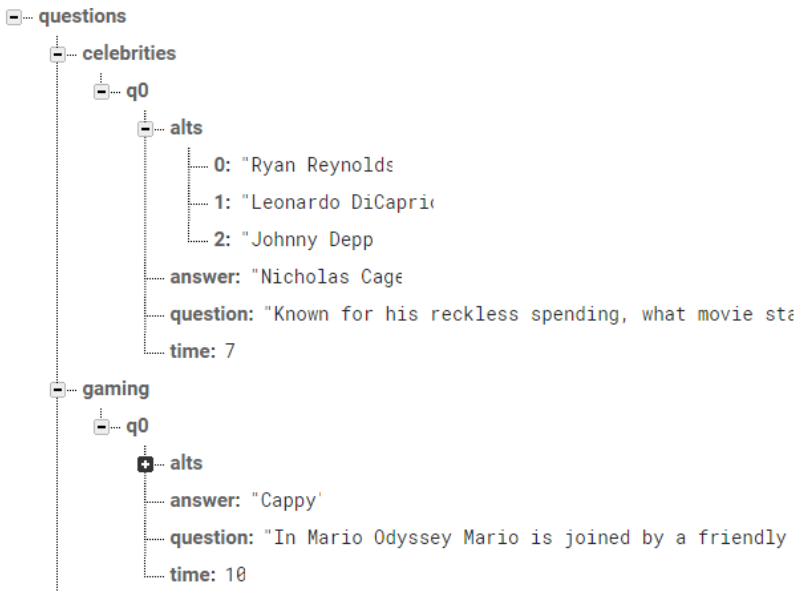


Figure 6: The structure of the question database

When the application launches a event listener is attached that fetches the questions from the database when the program launches and when the data in the database is updated. A backup of the fetched data is also stored on the device so that the application is still functional if it where to lose it's connection to the database.

## 4.2 Item database

Just like the questions the data needed to instance the different items in the game are also stored inside a Firebase Realtime database. The only real difference between this and the question database is the structure or each of the items. The top level node in the database is item which has the children buff, debuff and vanityItem. These children hold the data for their respective type of items. Each of these items have the children: price which is a int values and type, name and img_name that hold string values. The children of buff and debuff hold the int value time headstart and the double value scoreMultiplier. The children of buff also have the int values amountOfAlternatives (se figure 7 and the children of debuff hold the boolean value autoAlt. All of theses values except img_name is used in the creation of the different types of items, while img_name is stored in a map with the name of the item as the key and the image name as the value, this is used to connect items to their respective images in the different views.

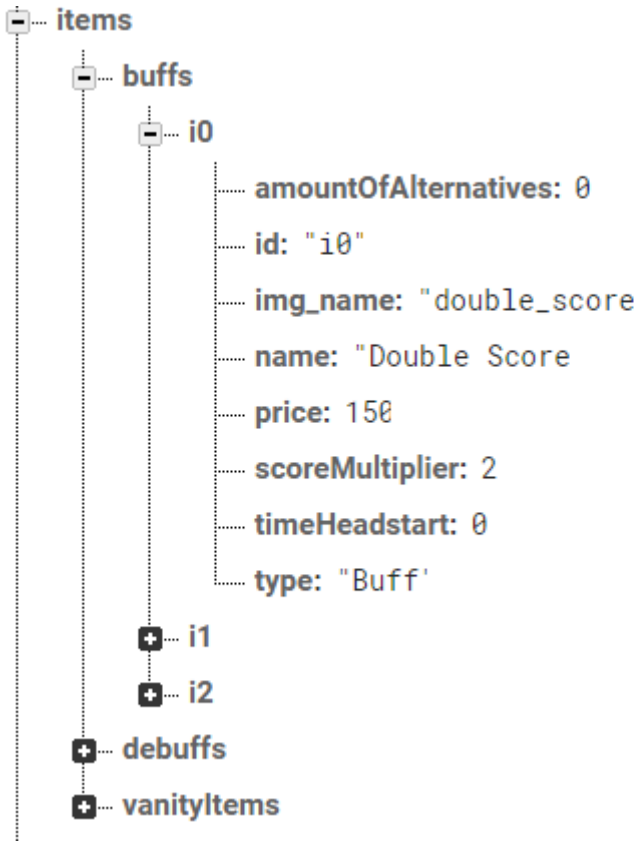


Figure 7: The structure of one buff in the item database

# 5 Quality

## 5.1 Testing and Continues integration

The application is tested using both JUnit tests and Android Instrumentality tests. Android Instrumentality tests where used for classes that required context to be able to be tested, such as the data loaders. For the view model classes Mockito was also used to mock certain components to simplify the testing process and remove any unwanted side-effects. The classes that did not require context or need any mocking where tested using JUnit.

All classes in the model except QuizGame were tested with 100% line coverage while QuizGame was tested with 90% line coverage (see figure 8), while only most of the classes in database service and some of the methods in networkManager were tested. The rest of the application remains untested at the present. This is because of a lack of time and the fact that only a small amount of the test were written as their class counterparts were developed. In the case of any future development this should be one of the first things that should be rectified.

| modelEvents | 100% (4/4) | 100% (9/9) | 100% (18/18) |
|---|---|---|---|
| Buff | 100% (1/1) | 100% (2/2) | 100% (4/4) |
| Category | 100% (14/14) | 100% (44/44) | 100% (84/84) |
| Debuff | 100% (1/1) | 100% (2/2) | 100% (4/4) |
| GameMode | 100% (1/1) | 100% (2/2) | 100% (4/4) |
| Item | 100% (1/1) | 100% (4/4) | 100% (8/8) |
| ItemFactory | 100% (1/1) | 100% (8/8) | 100% (50/50) |
| Lottery | 100% (1/1) | 100% (4/4) | 100% (12/12) |
| Modifier | 100% (1/1) | 100% (3/3) | 100% (6/6) |
| Player | 100% (1/1) | 100% (23/23) | 100% (52/52) |
| PlayerManager | 100% (1/1) | 100% (34/34) | 100% (168/... |
| Question | 100% (1/1) | 100% (8/8) | 100% (16/16) |
| QuestionFactory | 100% (1/1) | 100% (5/5) | 100% (17/17) |
| QuizGame | 100% (1/1) | 92% (35/38) | 89% (127/1... |
| Store | 100% (1/1) | 100% (8/8) | 100% (28/28) |
| Team | 100% (1/1) | 100% (7/7) | 100% (14/14) |
| VanityItem | 100% (1/1) | 100% (1/1) | 100% (2/2) |

Figure 8: The code coverage of the model

Continues integration was also used in the form of Travis. The application was built towards android api 24 and 28. Travis was also used to generate JavaDoc and a git inspector report that was pushed to the projects github page. However there seemed to be some inconsistencies with the git inspector report so it should not be taken as gospel.

## 5.2 Known issues

Currently the application has the following known issues:

- The player can currently not choose their profile picture.
- Teams can not be renamed.
- There is currently no end to the game.
- Network gameplay does not support lottery functionality.
- Network gameplay does not support store functionality.
- Hotswap lobby allows the add button to be swiped but not completely removed. It should not move at all.
- A game can't be setup into different game modes, the selection is simply a dummy implementation.

- Clients are currently not visibly being notified about them waiting for the server in various situations during gameplay.
- Strings are hard coded instead of using android string resources. Makes future translation work much more tedious.
- In hotswap lobby, the add-player button does not disappear when maximum players are reached.
- Firebase connection is not shut off when the app is sent to background.
- In network game players can not see each others category vote.
- In hotswap mode only one player may vote for a category.
- Weird dependencies between repository and several views caused by how NewView-Event is populated. Not fixed due to time constraints.
- The question number is not updating.

## 5.3 STAN

The dependency diagram that was generated using STAN only displays the classes that did not use any android classes(9). This is because we could not get android studio to build a jar file with these present. Because of this the diagram may be a bit misleading. Never the less it does display the dependencies between the classes in model and the classes in the network module(10).

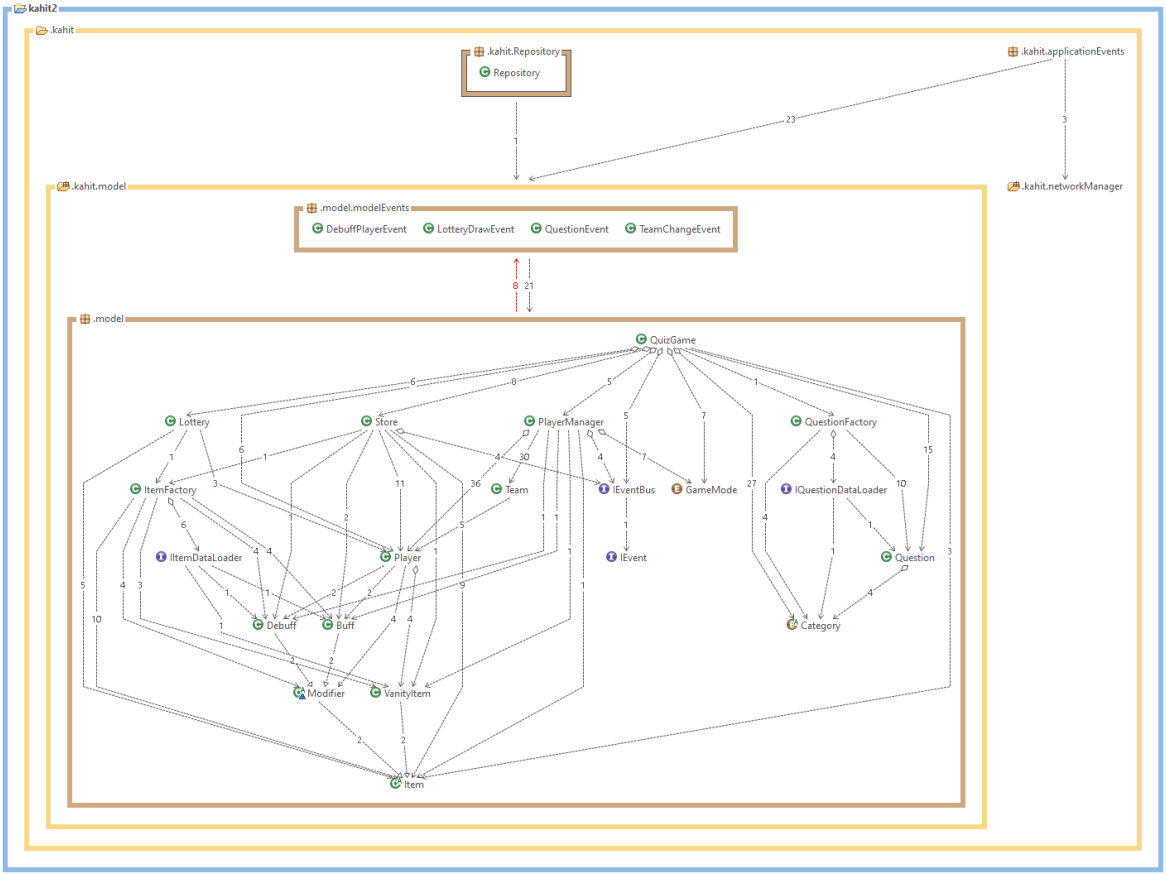


Figure 9: The STAN report of the model.

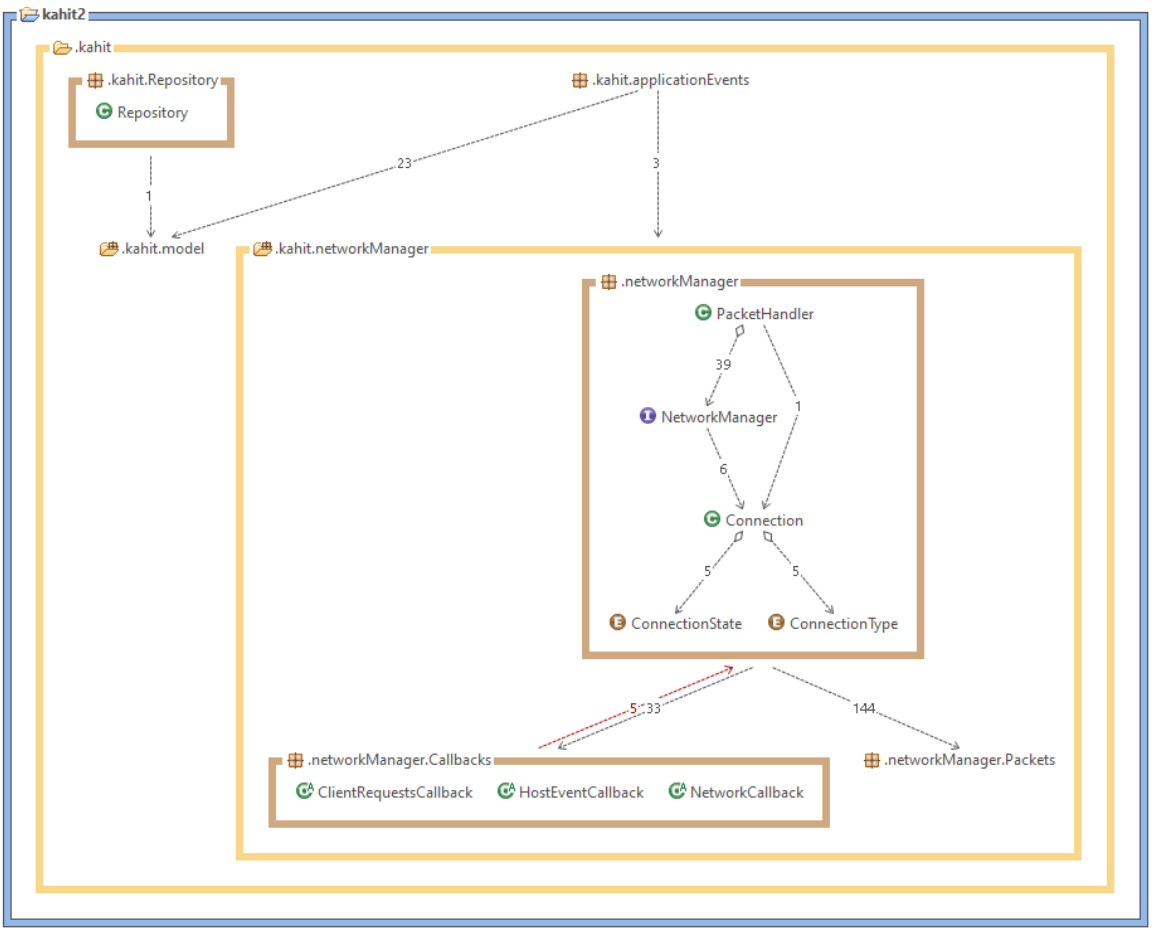


Figure 10: The STAN report of parts of the network module

### 5.4 Access control and security

NA

## 6 References

Firebase Documentation: `https://firebase.google.com/docs/database`

Nearby Connection: `https://developers.google.com/nearby/connections/overview`

Green Robot EventBus: `http://greenrobot.org/eventbus/`

Gradle automation tool: `https://docs.gradle.org/current/userguide/what_is_gradle.html`

MVVM structure: https://www.toptal.com/android/android-apps-mvvm-with-clean-architecture