Report: Java Avanced Project - TriviaLim

Nguyen Viet Sang, Le Tan Nhat Linh
{viet-sang.nguyen, tan-nhat-linh.le}@etu.unilim.fr

Faculty of Science and Technique, University of Limoges

Abstract. This report presents the main phases to build an Android application named TriviaLim. Our application successfully comes out with a well-proportioned UI and provides almost of required functions. We also introduce the way to do the layouts and the implementation of the business logic running in this application as well as point out our highlights.

1 Introduction

In this project, we implemented an Android app which serves a game called TriviaLim. The following are the main functions we had already done:

- Register, login, logout with Firebase Authentication
- Play with classic mode
- Add new friends, defy an another player
- Play the defied game and then response to the sender
- Show the score table of the current player

Regarding each above function, we designed the data structure intending to the most convenient way to read and write objects on Firebase while executing the source code. We also built business logic carefully to ensure our application runs and meets the requirements correctly. We then implemented the UI as well as the corresponding use cases.

Besides meeting all requirements of the project in a short amount of time (we had finished this product before Oct. 23rd, 2019), we studied and applied some advanced techniques. Our special things are: (1) For all the screens in this application, we had made their own Custom View Components [4] which put each object in that screen to fixed size and position regardless the variants of screen's sizes or the landscape mode by extending *FrameLayout* class; (2) For the synchronization problem when retrieving data from Firebase, we used *Observable* and *Observer* [2] from RxJava (in extra time); (3) We defined our own simple protocol to send and receive messages between the two users involved in a defyed game. We Figure 1 shows the screenshots of our product we mention in this report.

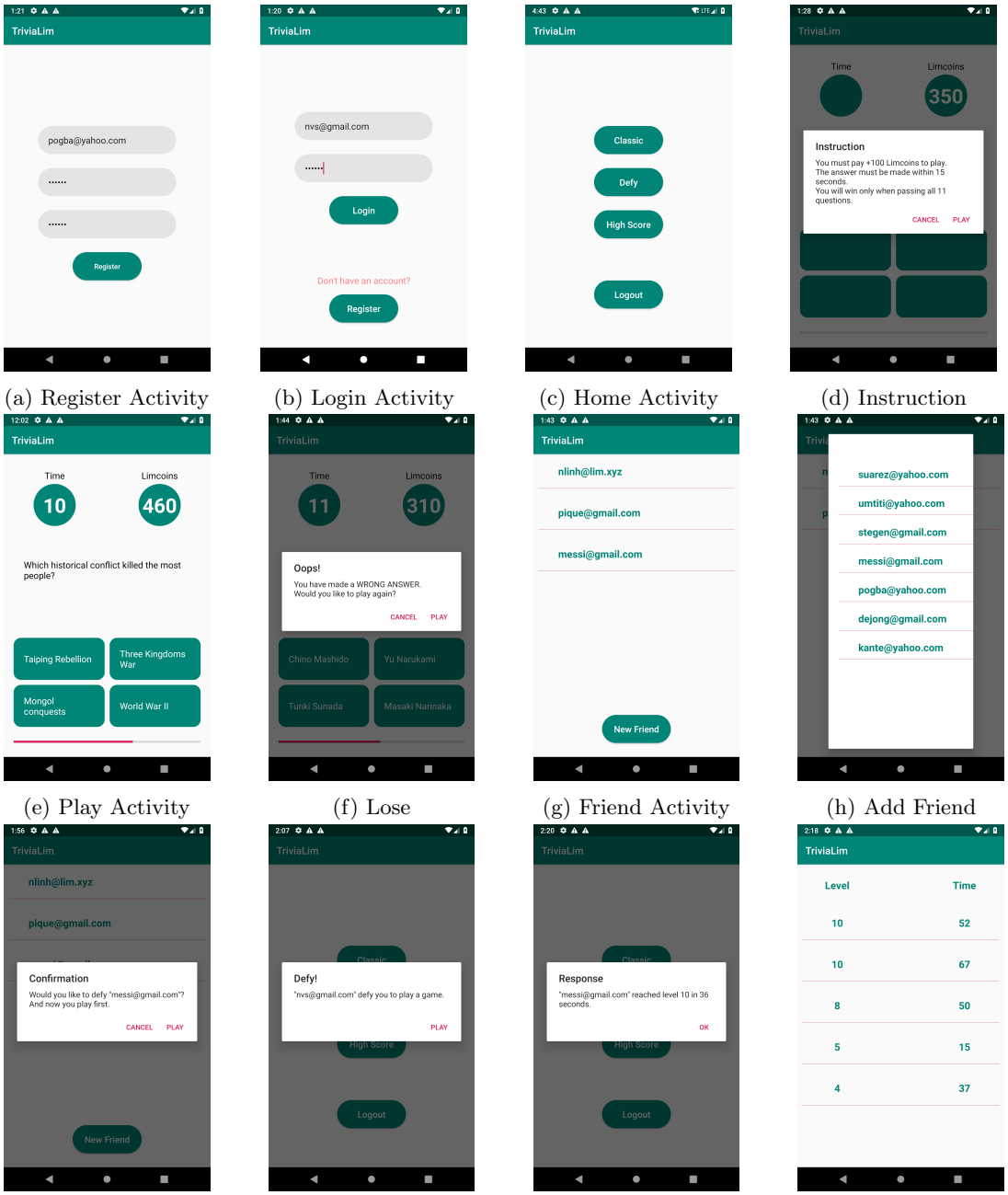
In the next sections, we will delve into the details of our implementation, including the structures of data we designed and practiced with Firebase and how to implement some main functions such as playing with classic mode or defying a friend.

2 Data structure

There are three principle objects we proposed as follows:

User contains necessary attributes of players who join this game. We consider the sketch of the data structure below. Each successfully registered user will be created a node on the Firebase whose parent is "user_id" (provided by the authentication of Firebase) and the first children are "username" (registered email), "limcoins" (initiated with 500), "defy" (store the message "none"). The subnode "list_friends" will be created when the user add a first new friend. In the next time adding new friends, the usernames of the friends will be appended to this node. Similarly, the "score_table" node will appear only when this user completes the first game. This node contains a list of *RankItem* objects.

```
user_id
├── username
├── limcoins
└── list_friends
```



(a) Register Activity (b) Login Activity (c) Home Activity (d) Instruction

(e) Play Activity (f) Lose (g) Friend Activity (h) Add Friend

(i) Defy (j) Notification (k) Response (l) Score Table

Fig. 1: Screenshots of application TriviaLim

- `defy`
- `list_qa_tuples`
- `score_table`

We now pay our attention to the "defy" node and the "list_qa_tuples" node. We use those nodes to define a simple protocol between two users when one of them want to defy the other. Regarding the first node, we use it to contain the message when having a user who wants to defy someone. For example, we suppose to have two users Messi and Ronaldo with ("abcMessi10", "messi@gmail.com") and ("xyzRonaldo7", "ronaldo@yahoo.com") considering to the tuple (user_id, username). At the beggining, we have two nodes of these two users on the Firebase with the strings "none" in the "defy" fields. Messi wants to defy Ronaldo with the same game. He firstly plays that game. When Messi finishes, the current list of eleven tuples of questions and correspoding answers is immediately written to the "list_qa_tuples" in the node "xyzRonaldo7". The "defy" field in the user's node of Ronaldo now contains the message "send@abcMessi10" denoted this defied game was sent from Messi. Ronaldo

accepts the challenge and when completing the game, his "defy" node is set to "none" again and the "list_qa_tuples" node is cleared. In the meantime, the message "response@xyzRonaldo7@5@35", for instance, is sent to the node "defy" of Messi. It announces that Ronaldo had completed the challenge of Messi and he had reached to level 5 in 35 seconds. The character "@" which is our convention plays a role as a separator. When Messi logins to the application in the next time, he will be received a dialog of an announcement about the result of Ronaldo, and after that, Messi's "defy" node will be reset to "none" too.

QATuple was used to define a tuple of question and its four multiple-choice answers. We also wrote a *Mapper* function to convert the string of JSON obtained from the API to a new list of *QATuple* objects. All manipulation relating to questions and answers were executed on this list.

```
qatuple_id
├── question
├── answer_1
├── answer_2
├── answer_3
├── answer_4_correction
└── level
```

RankItem includes the two compulsory attributes which are the number of questions accompanying with the time that the user had spent to answer them correctly. *RankItem* objects are used to record the results of each user and are saved in the "score_table" field in the node of corresponding users.

```
rank_item
├── number_of_questions
└── time
```

3 Implementation details

In this section, we firstly discuss our special approach to do the layout for UI. We then will dive deeper into the implementation of the Play Activity which plays the most important role in this project. For the other functions, if there are some similarities in the way of constructing, we will not analyze the things that have already been made clear.

3.1 Layout

To draw and arrange objects appearing in every activity, instead of simply configuring the XML files in the layout directory, we extended the $FrameLayout$ [5] class and overrode the functions $onMeasure$ and $onLayout$. By this method, we can measure the width and the height of each component exactly relying on the width and the height of our smartphone's screen. Moreover, we made the ratios between them and hence making our UI become proportional even when we rotate the smartphone to landscape.

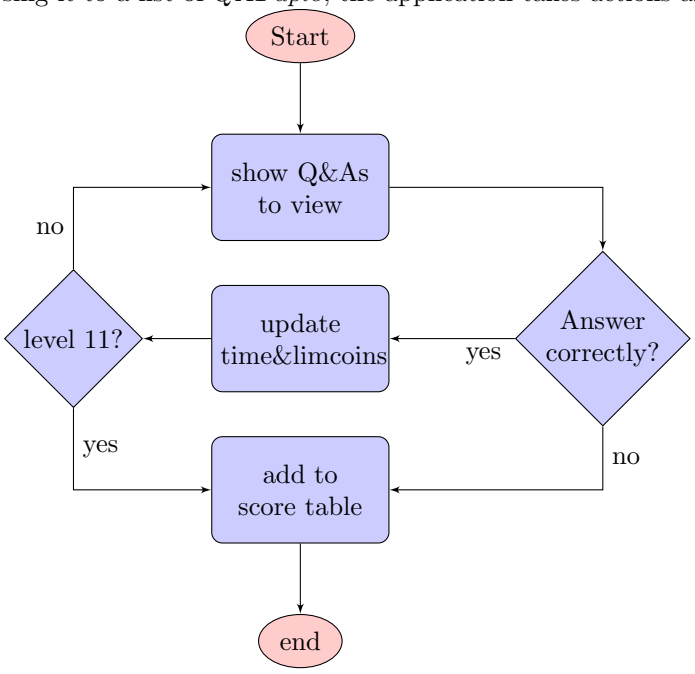
For example, we consider the screen of Login Activity in the figure 1b. At the beginning, we took the height of the navigation bar as the standard. Then, each component such as the two EditText objects where we enter the username and password, or the two Button objects of login or register, would be set the same height with the standard one. The widths of a EditText object and a Button object equivalent to 2/3 and 1/3 of the width of the screen respectively. Regarding to the terms of position, we set the login button in the center of the screen by calculating and passing the intended tuple of coordinates (left, top, right, bottom) of this object to the layout function. The positions of other objects were calculated based on the button at the center. Besides, we filled the color and smoothed four corners of the Button objects as well as the EditText objects in the way of setting the background with XML files in the drawable folder. As shown in the figures, we see that using this method made all objects in the screen become extremely proportional and professional.

For the task of viewing a list of items, for instance, Score Table in figure 1l, we divided the screen into permanent areas including one for the string "Level", one for the string "Time" and the rest for showing the list. We did the layout for a item which is a tuple of value (level, time) separately, it means defining the height of the item, the distance between the number and the left margin, etc. Then we showed the list of items by using RecyclerView [3].

3.2 Play Activity

As required, this activity combines two separated fragments. The first one involves the three TextViews of the question, the count-down timer and the Limcoins. The second fragment contains the four TextViews of the multiple-choice answers and the progress bar. For the layout, we firstly divided the screen into the upper half and the lower half, then did the layout for them separately.

Flow. When a user clicks the Classic button in Home Activity (see figure 1c), the program will start the Play Activity with the dialog of instruction on the screen (see figure 1d). Once the user continues clicking "OK" button, the application will immediately make a request to obtain 11 questions accompanying with its answers from the API REST (https://opentdb.com/). We had established a service connection (TriviaService) and sent a message representing the request to this service. At the service side, we extended the class *Handler* to handle the incoming message. We then used library *Volley* [6] to implement the HTTP connection to get the JSON object containing the required questions and answers. In the next step, we returned this JSON object to the Play Activity by broadcast method. At the Play Activity, the broadcast receiver was still listening to catch the returned result. We should notice that connecting to HTTP task through *Volley* runs on a distinct thread. Therefore, we have to wait until the result is returned before manipulating on it. When receiving the result and after parsing it to a list of *QATuple*, the application takes actions as the following flowchart:



For the case of time out, the activity will also jump to the end state. We set the limited time to give the answer for each question to 15 seconds by using the *CountDownTimer* supported by Java. Whenever the player touches the finger to one of the four TextViews of the answers, the timer is suddenly stopped and the program adds the passing time for choosing the answer for the current question to the total passing time calculated from the first question. We then drive the application to check the player's answer whether it is true or false. When displaying the answers to the screen, to ensure the fair in this game, we randomly generate an integer number in range $[1, 4]$ denoted the position of the correct answer. For the rest three positions, we do not care too much about them since they contain the wrong answers. Because of keeping the postion of correction, we can easily check the coincidence with the answer given by player.

During playing time, we plug a real-time listener to the Limcoins node of the current user. Whenever a user earns amount of bonus coins, we add to the current number and update immediately and directly to its node on Firebase. We do the same things at the end of the game for updating a new record of the achieved level and the corresponding cost of time.

Fragment. Since the fragments contain their own components, we must have implemented the communication between them and their activity. For example, regarding to the upper fragment (called *QuestionFragment*) containing the three TextViews of the question, the timer and the Limcoins, we

proccessed the business logics in the activity and it thus was necessary to establish the communication with the fragment to display the text of question, the time as well as the Limcoins. Our solution is to create an interface in the *QuestionFragment* with three abstract functions *displayQuestion*, *setTextClock*, *setTextLimcoins*. In the Play Activity, these functions were overridden and called in where and when needed to show the texts to the screen. In a similar way, the Play Activity communicated with the lower fragment containing the four answers and the progress bar.

3.3 Defy a friend

In the Home Activity (see figure 1c), when a player chooses "Defy" button, the application jumps to Friend Activity listing the current player's friends as shown in the figure 1g. As mentioned in the section 3.1, we used RecyclerView by extending the class Adapter and then set an adapter for the list. When the user click one of the friends' username (email), a dialog of confirmation will appear like the figure 1i. If continuing choosing to play, the application will lead the user to Play Activity and the things will happen in the absolutely similar way when playing with classic mode. Besides, in the Friend Activity, we can also add a new friend by clicking the button at the bottom (see figure 1g). After clicking, a dialog of list containing emails of all friends of the current user which we had filtered to ensure they had not been in any relationship with the current user.

In the defying mode, we had designed the functions in the Play Activity in the way that they could be reused as many as possible. Therefore, the different thing between this mode and the classic mode is to save the data including the eleven tuples of questions and corresponding answers, and the defied message to the friend. We consider the $User$ tree in the section 2, the "defy" node is the place in which we save the message. When the defied friend logins to the application, this field will be checked to realize if there is someone sending an invitation to play or not (see figure 1j). The "list_qa_tuples" node contains the eleven $QATuple$ objects so that the defied friend could play with the same challenge as the sender. At the end state in the above flowchart, the defied friend finishes the game and the result (level and spending time) will be written back to the "defy" node of the sender of defied message. In the next time loginning to the application, we examine this response, parse the message and show the notification via dialog as shown in the figure 1k.

3.4 Score Table

In the Home Activity (see figure 1c), clicking the "High Score" button leads us to the Rank Activity (see figure 1l). Whenever coming to the end state of the game, we create a new *RankItem* object with the two parameters of the current level and the consuming time. We then append this object directly to the "score_table" node of the current user on the Firebase. To show the score table as the figure 1l, we read all the values contained in this node, cast them to *RankItem* objects, sort these objects by using *Collections* library and finally use RecyclerView to display. For the layout, we have mentioned in the section 3.1.

3.5 Firebase Usages

For writing an object to a certain node on Firebase, we reference to that node and set value to it. If we would like to have a random key for a new node of the object, we use *push* method.

When reading an object from Firebase, we add a listener to the target node. There are three types of listeners: $addChildEventListener$ for handling different events on a node, $addValueEventListener$ for real-time listening, $addListenerForSingleValueEvent$ for reading data once. Depending on the purposes, we used them for suitable cases. Because each of these retrieving tasks is executed on a distinct thread, it is necessary to manage the moments of returning results to avoid referencing to null objects. We had studied and applied *Observable* and *Observer* from ReactiveX to deal with this problem. By this approach, we could also separate our code to **data tier** and **view tier**, hence making the project cleaner and brighter.

For example, we consider the task of reading eleven objects of $QATuple$ to play the defied game. We declared a function whose the returning type was $Observable < List < QATuple >>$. In this function, we used the method $emitter.onNext()$ to return the result. In the receiver's side, we defined an *Observer* object to receive the result. The observable function was called and assign to subscribe this object. Once receiving the result, we would do our next steps.

3.6 Authentication

Login. Login Activity is the first activity when opening the application. The interface consists of two input fields for email and password, a login button, a register button for new user, and a hidden text view used for notification (see figure 1b). In order to login, an user has to input the email on the first field, and the password on the second field, then press "Login" button. If the login is successful, the user's screen will display Home Activity (see figure 1c). If the user did not input anything on one of the two input fields or if the authentication with the server does not success, there will be a text of notification to inform the user. For new user to register, there is a "Register" button at the bottom of the screen which will lead to Register Activity. The authentication uses `FirebaseAuth.signInWithEmailAndPassword` [1]. We simply pass the email and user ID within the intent to the other activities and there, we use them to reference the current user's node of data on the Firebase. Afterward, the user's information can be access through by getting `FirebaseUser` from `FirebaseAuth` in other activities.

Register. In the Register Activity, there is three input fields for email, password and password confirm (figure 1a. The password is checked to be at least six characters to be able to register a new account. Using `FirebaseAuth.createUserWithEmailAndPassword` method [1], if the task is successful, the user will be immediately signed-in and a new *User* object will be created on the Firebase. Then, the application switches to Login Activity where it catches the current logged in user and switches to Home Activity. In case of register failure, there is a hidden Textviews object used to display messages about the failures.

Logout. In order for the user to logout of the current session, the user need to press "Logout" button on Home Activity (see figure 1c). The user will be signed-out from the `FirebaseAuth` and the application will switch back to Login Activity.

4 Conclusion

In this project, we have fulfilled the standard requirements set out from the beginning in a short amount of time. We have also presented our approach to do the layouts and the details of the implementation as well as our own special things.

For the limitations, we did not concentrate to construct a wonderful user experience. For the further work, we need to display the avatars of users, replace emails by usernames, design some delicate dialogs, etc. Besides, we have not specified the rules to secure the database. The method used to access database from the mobile is also not restricted. Therefore, there might be a loophole in the security aspect.

References

1. Authenticate with firebase using password-based accounts on android, https://firebase.google.com/docs/auth/android/password-auth
2. Class observable¡t¿, http://reactivex.io/RxJava/javadoc/io/reactivex/Observable.html
3. Create a list with recyclerview, https://developer.android.com/guide/topics/ui/layout/recyclerview
4. Custom view components, https://developer.android.com/guide/topics/ui/custom-components
5. Framelayout, https://developer.android.com/reference/android/widget/FrameLayout
6. Make a standard request, https://developer.android.com/training/volley/request