

DEVICES AND MOBILE APPLICATIONS - PROJECT

“Innovative Banking”

Mobile Banking Application

Coordinator: Conf. dr. Mihai DOINEA

Student: Eftene Robert-Petruț

Group: 1092

Team No: 11

Contents

[1. Introduction 3](#_Toc60916869)

[1.1. Defining the problem 3](#_Toc60916870)

[1.2. Defining the objectives 3](#_Toc60916871)

[1.3. Defining the marketplace 4](#_Toc60916872)

[2. Solution Description 4](#_Toc60916873)

[2.1. Usage scenarios 4](#_Toc60916874)

[2.2. Functionalities 4](#_Toc60916875)

[3. Solution Implementation 5](#_Toc60916876)

[3.1. Functionalities in depth 5](#_Toc60916877)

[3.2. Problems/Advantages in development 18](#_Toc60916878)

[4. Conclusions 19](#_Toc60916879)

[Table of figures 20](#_Toc60916880)

[References 21](#_Toc60916881)

# Introduction

## Defining the problem

The subject of this project is **mobile banking**, therefore the application **InnovativeBanking** aims to facilitate the main actions available when you own a bank account.

The main problem of a customer at a bank is the big amount of time that he must spend to complete a series of processes in order to perform simple actions which require quite a lot of time due to the context in which they are perform. Because the customer must come to the bank office, to wait in endless queues, to complete many forms, and so on, actions such as creating a new card, sending money to a partner, creating a savings vault and so on will take a lot of time. Another problem would be the fact that a customer, in most of the cases, must go to an ATM to query his account sold or if he wants a report with his transactions, he must go to a bank office.

A general problem within institutions is the high degree of congestion which becomes an even more serious problem in a pandemic, such as the one currently caused by Covid-19.

Therefore, with these issues taken into consideration, **InnovativeBanking** aims to achieve a list of objectives.

## Defining the objectives

The statistic say that smartphones users increase year after year. In 2020, the number of global smartphone users is 3.5 billion, marking a 9.3 percent increase from 2019 and a 45.5 percent from global population. In other words, more than four out of every ten people in the world are currently equipped with a smartphone [1] .

Therefore, the main objective of the **InnovativeBanking** application is to integrate most of the banking actions into a mobile application which can be use by every bank customer who is equipped with a smartphone. These activities that in the past were performed only into a bank office can be performed now by simply using any smartphone with the help of the banking application.

In several words, for solving the problems presented in the last chapter **InnovativeBanking** will meet the following specifications:

* Currently, 72.48% of the smartphone owners have an android operating system mobile [2]. Therefore, the application will be developed in the first instance for this operating system to be used by as many users as possible until further development.
* The possibility to create a personal account where all the actions of the user to be stored and centralize.
* The possibility to send and add money or to open an economy vault easily. This kind of actions, in the past, were taking a big amount of time due to the congestion level at the bank offices or because of the lack of staff.
* The application will let the user to query all his details about his cards or his transactions therefore the allocate time to do basic bank action will decrease because the bank client will not need any more to go to any bank office or ATM.

## Defining the marketplace

**InnovativeBanking** can be used by any user of an android operating system smartphone. The application helps the users to efficient certain banking actions, so it can be used at anytime from anywhere without the need of physical presence of the customer at the bank office.

# Solution Description

## Usage scenarios

A first scenario would be for the application to serve a certain bank. Therefore, the use of the application will decrease the time allocated by the clients for some of the bank actions and will make the processes more efficient.

Another scenario would be for the application to be scaled to represent a sustainable digital bank. In this scenario the process of sending money will be more efficient because all the user will belong to the same bank entity. Besides that, this scenario will let the application to act like a bridge between different banks entity which will add another point for efficiency.

For these scenarios, the use of the application will come with the following advantages:

* User-friendly interface
* Easier storage of a bank account data
* More efficient centralization of data
* Easier communication between application users
* Keeping a better track of the flow of the money

## Functionalities

With these scenarios in mind, the current version of the application comes with the following functionalities:

* **Account creation and login functionality** – a user can easily create a new account and after that he can already login to the application with the new account. All the users will have their data centralized using the account. The user can change his account detail and he can close his account.
* **Adding money to the account** – a user can easily add an amount of money using any credit card that he owns. Therefore, the user can add his cards details to the application in order to efficient even more the process.
* **Adding credit cards –** a user can easily add his credit cards to the application to decrease the amount of time spent on introducing the card details every time he wants to add money to his account. Besides that, the user can delete and edit any of his saved cards whenever he wants.
* **Sending money** – a user can easily send money to any receiver by completing a short form in which the user must introduce the receiver account details. Besides that, the user can add the receivers to a partners list which will allow him to send directly to these partners without completing again the form.
* **Getting the last stocks information** – for now a user can consult the last information about stock values of the most known companies. In a future version of the application the user will have the possibility to invest directly from the app in stocks.
* **Getting a report with transactions** – a user can consult whenever he wants his transaction which are sorted by the date of each transaction.
* **Getting a statistic regarding transactions –** a user receives a statistic regarding his expenses and his incomes. Therefore, he can make a comparison between expenses and incomes.
* **Create economy vault** – a user can easily open an economy vault completing a form. He can set a target value for the vault and keep track of the amount of money that are in the vault. Beside that he can add money whenever he wants to the vault and on the same note, he can withdraw money from the vault. The user can close and edit the vault too whenever he wants.
* **Setting a target for the account** **–** the user can set a target of money for his account.

# Solution Implementation

## Functionalities in depth

Most of the features presented in the previous chapter are developed in order to work with a database which store all the information regarding a user account. Therefore, the data is persisted in two ways:

* Using a local database with the help of **Room** which is a persistence library that provides an abstraction layer over SQLite [3]. The main benefits of using this library are the fact that it minimizes repetitive and boilerplate code, and it is easy to setup for using.
* Using a local file with help of the interface **SharedPreferences**.

For the local database, as I mentioned, the application use **Room** which is structured in three major categories:

* **The database class** which represents the main access point between the app and the persisted data. In order to establish a connection between the database and the application the database class must extend Room Database and must be annotated with “@Database” annotation (Figure 1).



Figure 1 Database class

* **Data entities** – java classes used for creating an object relational mapping pattern. These classes represent the tables of the database. **InnovativeBanking** database consist of 5 tables. Therefore, we have five java classes: CreditCardModel.java, PartnerModel.java, TransactionModel.java, UserModel.java and VaultModel.java. Each class contains a set of fields that represent the table fields from the database. In order to perform the mapping, the class must be annotated with the “@Entity” annotation. Every entity must have an id annotated with “@PrimaryKey”. In our case the id is set to be auto generated. The “@Ignore” specify the fact that the field which is annotated with this must be ignored by the database.

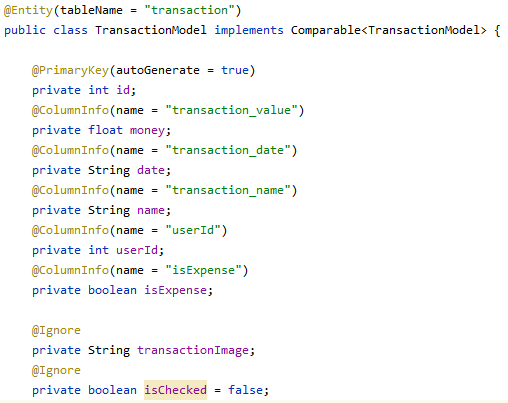


Figure Entity class example

* **Data access objects (DAOs) –** are interfaces that provide methods for accessing the database for querying, update, insert and delete data. The database class contains instances of every DAO. These instances are used in the application for accessing the database. Every class of this type must be annotated with “@Dao” and every method annotated with the method desired for accessing the database (Figure 3).

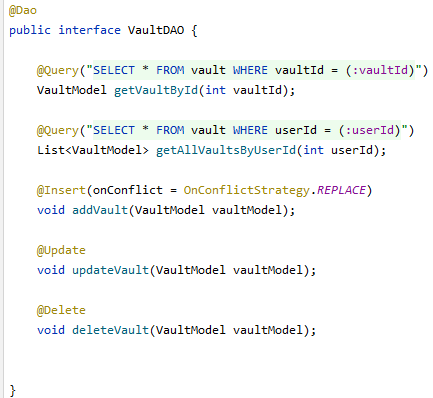
****

Figure 3 DAO example

In order to query some specific data of an entity with respect to another entity field the database entities form a relation. The table relations are:

* User entity and CreditCard entity (One to many)
* User entity and Transaction entity (One to many)

In order to create this kind of relation, I created a new class which contains an instance of the entity that represents the “one” part of the relation and a list with objects of the class that represent the “many” part of the relation (Figure 4).

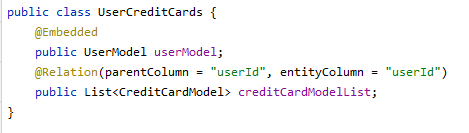


Figure 4 Relation class example

The application consists of several activities and fragments that are used for every feature described in the previous chapter (Figure 5).

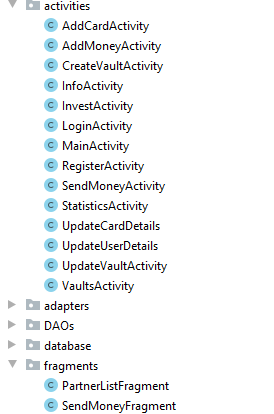


Figure 5 List of activities

The **login feature** is used in the LoginActivity (Figure 6). For this feature is used the UserDao to access the users from the database. All the users are queried and stored in a list. After that is made a loop over the list to check if the account entered in the activity exist in the database.

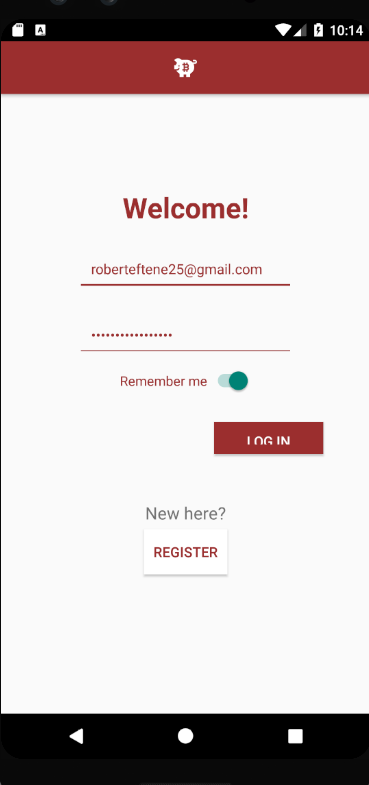


Figure 6 Login activity

If the login is successful, the id of the logged user is stored locally using SharedPreferences which create a file where, with the help of an editor, are saved items key-value based. In this case the value of the id is stored under the key “currentUserId” (Figure 7).

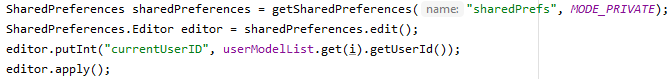


Figure 7 SharedPreferences

The id is used after that whenever information from the database about the logged user are needed in the application. For getting this id from the file created with SharedPreferences I created a method which return the id of the current logged user (Figure 8).

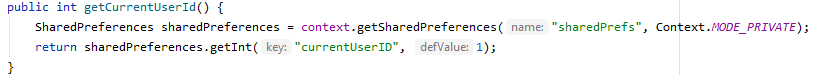


Figure 8 Get current user id method

An account can be **created** in the RegisterActivity. In this activity an object of the type UserModel is created with the data entered by the user in the register form and when the “register” button is pressed the insert method from UserDao is used for sending the object created in the database.

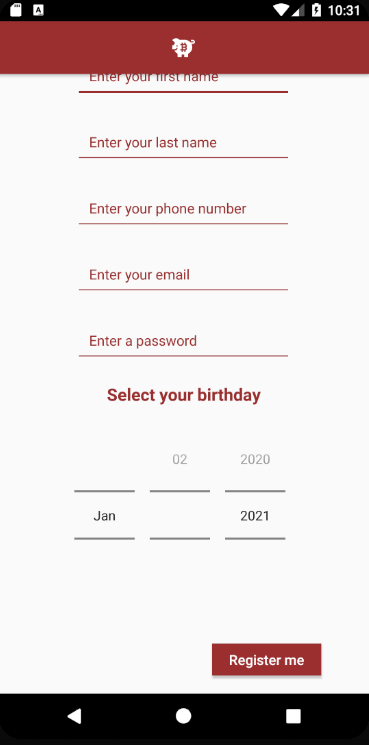


Figure 9 Register Activity

In the MainActivity the user has a report with his transactions ordered by the date of them (Figure 10).

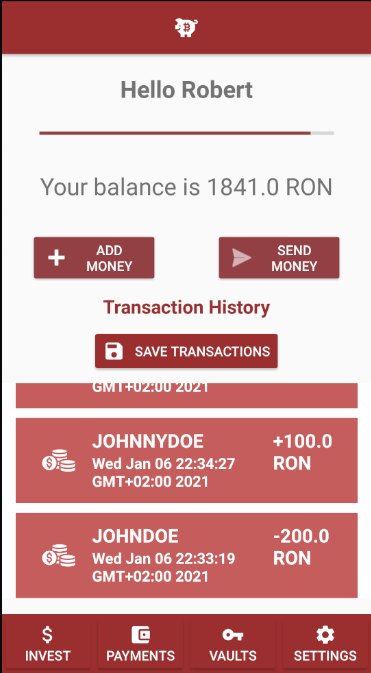


Figure 10 MainActivity

The transactions are returned as a list from the database using the class UserTransactions which was created for developing a relation between user and transactions (Figure 11).

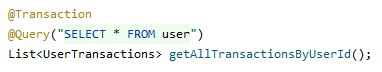


Figure 11 Get all transactions by user

The list returned from the data base consist of objects of type UserTransactions. Every instance of this class contains a User and his transactions. From this list the transactions of the current user logged in are extracted using his id (Figure 12).

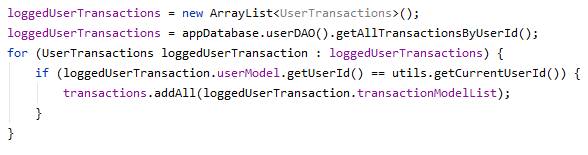


Figure 12 Extract user transactions by id

The transaction list is displayed using a custom adapter which offer the freedom to assign certain custom layout to each item of the list and to decide how every item of the list behave. The adapter acts as a bridge between a view and the data presented in the view.

The functionality of adding an amount of money is called in the AddMoneyActivity. In this activity the user has the possibility to view a list with his saved cards, to add a new card, to edit a chosen card by the user, to delete a chosen card and to add money from a selected card from the list (Figure 13).

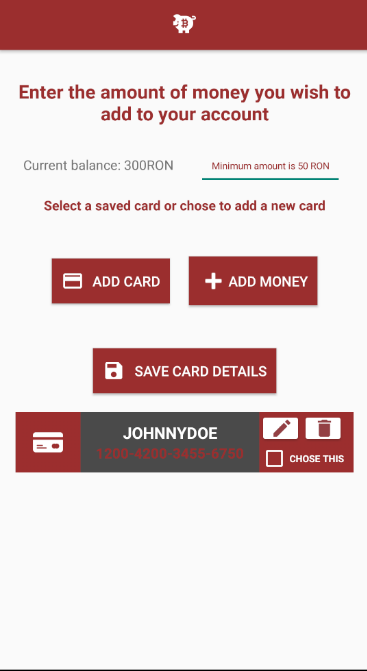


Figure 13 Add money activity

**Add money** – when the “add money” button is pressed an instance of the Database uses the UserDao. From the UserDao are used two methods, one is the method for getting a specific user by id (the id of the current user logged in). The user returned by this method is the user that will be updated using the second method, the one for updating a user. Before the update method is called, the user’s balance is updated accordingly to the amount of money added by the user using the accessor method from the UserModel class.

**Add card** – when the “add card” button is pressed an AddCardActivity starts in which the user complete a form with the details of the card that he wants to save. After that, an instance of the database uses the CreditCardDao for inserting the new card in the database. The method for inserting takes as parameter the object that will be inserted. This object is of type CreditCardModel and is created using the details given by the user in the form (Figure 14).

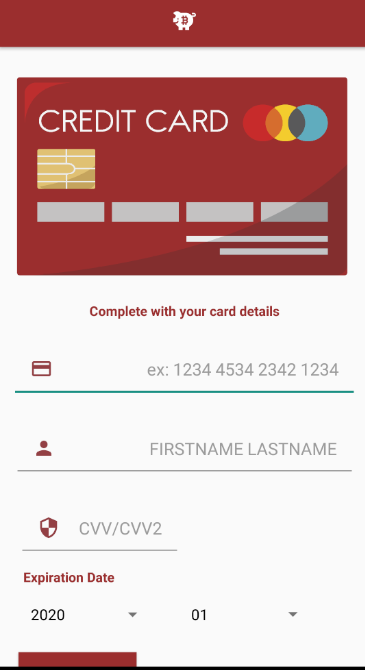


Figure 14 Add card activity

**Edit card** – the functionality of editing a specific card is made using an instance of the database class from which CreditCardDao is accessed and the update method is used. When the edit button is pressed the id of the card is passed between AddMoneyActivity to UpdateCardActivity using a method of the class Intent. With the help of this id the data of the selected card for editing are taken in the form for updating the card (Figure 15). After that the id is gotten by the UpdateCardDetails activity (Figure 16).

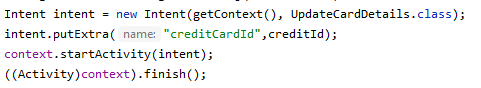


Figure 15 Pass data between activities



Figure 16 Get intent

**Delete card** – for deleting a specific card the delete method from CardCreditDao is used. This method takes as parameter the CreditCard instance which the user selects for deleting.

**Save card details** – this functionality writes a report into a txt file with a list of the cards saved and their details (Figure 16).



Figure 17 Write to txt

In the SendMoneyActivity are rendered two fragments. In the first phase it is visible only the PartnersListFragment which contains a list view with user’s partners (Figure 17). If the user wants to send to a person manually, he can press on the “send manually” button. When this button is pressed an instance of the FragmentTransaction starts a transaction which replaces the fragment of partners with a fragment that contain a form for sending money manually (Figure 19). Same action is happening when the user press on a partner from the list but this time the partner id is passed between fragments for getting in the form the data of the chosen partner

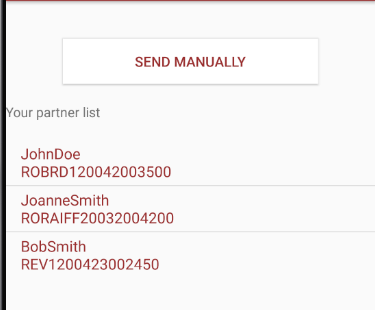


Figure 18 Partners list fragment

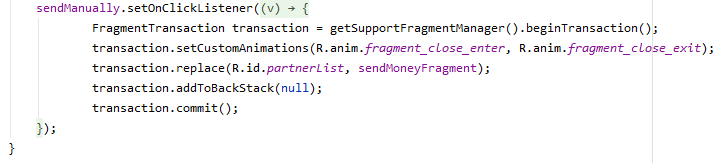


Figure 19 Fragment transaction

In the InvestActivity a list with companies and their stocks information is rendered. The data for this list is downloaded from the internet as a json object. The json is downloaded using AsyncTask class. This class allows an asynchronous task to be done in the background and after that it publishes the result on the UI thread. The task is defined by 3 generic types called “Params, Progress and Result” and it goes through 4 steps: “onPreExecute, doInBackground, onProgressUpdate, onPostExecute”. Before displaying the json downloaded from the internet it is parsed into an object of type CompanyModel. Every objects of this type got from the json is added to a list and displayed in the activity (Figure 20).

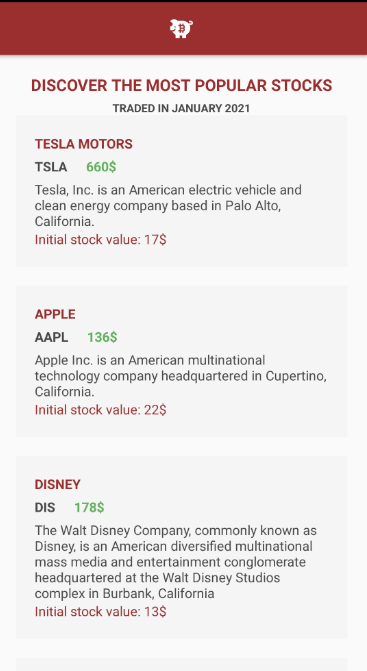


Figure 20 Invest activity

In the StatisticsActivity the user can get a statistic regarding his expenses and his incomes. These values are calculated using the list of transactions. Every TransactionModel instance has a field “isExpense” which is a Boolean that determine if the transaction is an expense or an income. Therefore, a loop is made over the list with all the transaction an if the boolean is true then the transaction value is added as an expense and if it is false it is added as an income. After that, a pie chart is created with these values using graphic elements like canvas, paint and methods for drawing (Figure 21).

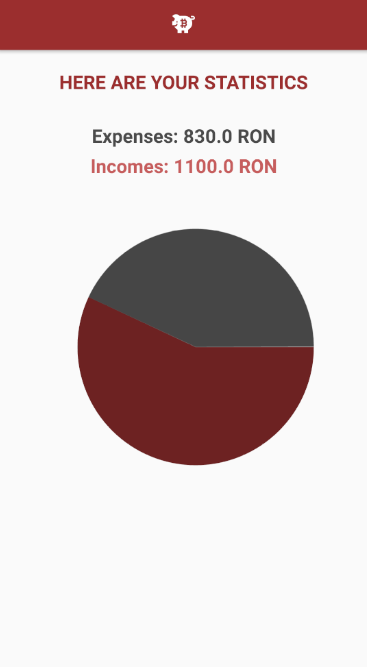


Figure 21 Statistics activity

In the VaultsActivity the application has the functionality of creating a vault for economies. These vaults are created using VaultDao’s method to insert. A user can add or withdraw money from a selected vault. In case of adding an amount of money that a user does not have in his account the application will display a toast to the user to alert him. Same in the case of withdrawing an amount of money bigger than the vault’s balance. These validations are made using the UserDAO’s method to get the data of the current user logged in and before adding money to the vault an if checks if the user’s balance is bigger than the amount of money that the user wishes to add to the vault. In the second case, to validate if the user has the amount of money in the vault an if checks if the vault balance, which is taken using accessor method of the VaultModel class, is smaller than the amount of money that the user wants to withdraw. Beside that the user can edit and close a selected vault. In case of closing a vault, which has money in it, the money will be added to the user balance. Therefore, when a user decides to close a vault an update on the current user logged will be made using an instance of the database class for accessing the update method from UserDAO interface. Before updating the user’s balance, the balance of the current user logged in is set to be the previous balance plus the balance of the vault that will be closed.

The vaults are displayed using a custom adapter like as the case of transactions. (Figure 22).

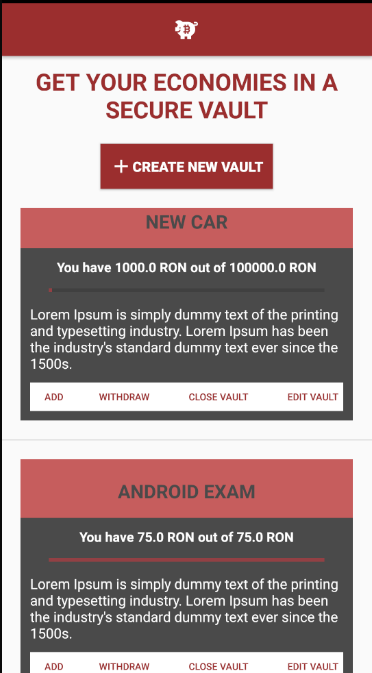


Figure 22 Vaults activity

In the InfoActivity the application offers multiple functionalities. Here, a user can edit his account details using UpdateUserDetailsActivity in which the current logged user details are written in the inputs of the form for editing. These details are taken using the id of the user which is gotten from the SharedPreferences file. Another functionality that a user can use in this activity is the functionality of setting a target of his account balance. For this the UserDao’ s method to update an object is used for updating his balance target. The user can also delete his account in this activity (Figure 23).

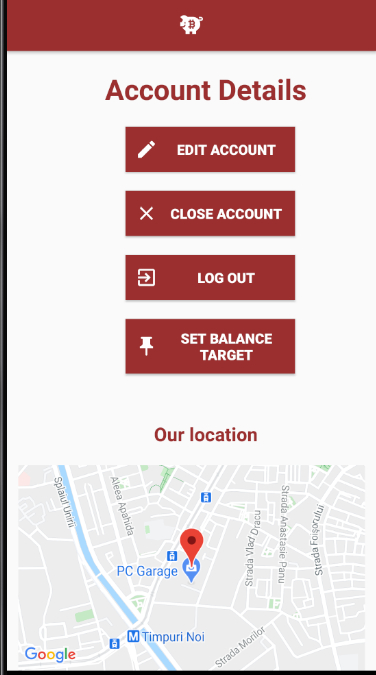


Figure 23 Info Activity

## Problems/Advantages in development

**Problems:**

From a business point of view a problem would be that not all the smartphone owners have an android operating system on their mobile. Another problem from this point of view will be the legibility of the application.

From a development point of view the main problem would represent the requirements of the program Android Studio which are quite high. Due to this fact the development process will be slowed down.

From a developer point of view the main problem was the lack of experience with all the classes, methods, tools and so on provided by android library. Therefore, for certain functionalities was quite difficult to find and understand an efficient solution.

**Advantages:**

From a developer point of view the main advantage was the friendly interface of the Android Studio. The materials provided on the platform and presented were a great advantage. Another advantage was the fact that the code was written in Java, a language which was presented before in the previous year. The fact that the language is widely used helped finding solutions faster.

# Conclusions

Considering that this application does not solve all the problems presented in the beginning of this documentation I would take into consideration scaling the application and migrating the local database to an external one and adding improvements from a security point of view.

Although the java language is widely used in the development of applications, I consider that a newer technology specific to the development of mobile applications, such as Kotlin, Swift, Flutter and so on can efficient the development process and, more important, increase the time assigned for the business logic of the application.

# Table of figures

[Figure 1 Database class 6](#_Toc60876749)

[Figure 2 Entity class example 6](#_Toc60876750)

[Figure 3 DAO example 7](#_Toc60876751)

[Figure 4 Relation class example 8](#_Toc60876752)

[Figure 5 List of activities 8](#_Toc60876753)

[Figure 6 Login activity 9](#_Toc60876754)

[Figure 7 SharedPreferences 9](#_Toc60876755)

[Figure 8 Get current user id method 10](#_Toc60876756)

[Figure 9 Register Activity 10](#_Toc60876757)

[Figure 10 MainActivity 11](#_Toc60876758)

[Figure 11 Get all transactions by user 11](#_Toc60876759)

[Figure 12 Extract user transactions by id 11](#_Toc60876760)

[Figure 13 Add money activity 12](#_Toc60876761)

[Figure 14 Add card activity 13](#_Toc60876762)

[Figure 15 Pass data between activities 13](#_Toc60876763)

[Figure 16 Get intent 13](#_Toc60876764)

[Figure 17 Write to txt 14](#_Toc60876765)

[Figure 18 Partners list fragment 14](#_Toc60876766)

[Figure 19 Fragment transaction 14](#_Toc60876767)

[Figure 20 Invest activity 15](#_Toc60876768)

[Figure 21 Statistics activity 16](#_Toc60876769)

[Figure 22 Vaults activity 17](#_Toc60876770)

[Figure 23 Info Activity 17](#_Toc60876771)

# References

|  |  |
| --- | --- |
| [1] | Oberlo, "HOW MANY PEOPLE HAVE SMARTPHONES IN 2020?," 2020. |
| [2] | StatCounter, "Mobile Operating System Market Share Worldwide," 2020. |
| [3] | A. D. Documentation, "Save data in a local database using Room". |