Budgetize:

Money Tracking Android Application

Bachelor thesis

A close up of a logo

Description automatically generated

Robert Marian Roland CIOTOIU

Supervisor:

Prof. dr. eng. Marius MARCU

**2020**

Contents

[1 Introduction 4](#_Toc43473601)

[1.1 Motivation 4](#_Toc43473602)

[1.2 The problem 4](#_Toc43473603)

[1.3 The solution 5](#_Toc43473604)

[2 State-of-the-art 6](#_Toc43473605)

[2.1 Overview 6](#_Toc43473606)

[2.2 Main competitors 6](#_Toc43473607)

[2.3 Conclusions 8](#_Toc43473608)

[3 Theoretical Foundation 9](#_Toc43473609)

[3.1 Concepts 9](#_Toc43473610)

[3.1.1 Object-oriented programming 9](#_Toc43473611)

[3.1.2 Android Activity Lifecycle 9](#_Toc43473612)

[3.1.3 Model-View-ViewModel (MVVM) 10](#_Toc43473613)

[3.1.4 Android LiveData 12](#_Toc43473614)

[3.1.5 SQL 12](#_Toc43473615)

[3.1.6 RESTful Web Service 12](#_Toc43473616)

[3.1.7 RESTful API 13](#_Toc43473617)

[3.2 Used Technologies 13](#_Toc43473618)

[3.2.1 JAVA Programming Language 13](#_Toc43473619)

[3.2.2 XML (eXtensible Markup Language) 13](#_Toc43473620)

[3.2.3 Gradle 14](#_Toc43473621)

[3.2.4 Apache Maven 14](#_Toc43473622)

[3.2.5 Android Operating System 14](#_Toc43473623)

[3.2.6 Spring Boot 16](#_Toc43473624)

[3.2.7 IntelliJ IDEA 16](#_Toc43473625)

[3.2.8 Android Studio 16](#_Toc43473626)

[3.2.9 MSSQL 17](#_Toc43473627)

[3.2.10 Room Database (SQLite) 17](#_Toc43473628)

[3.2.11 Room Explorer 18](#_Toc43473629)

[3.2.12 Leak Canary 18](#_Toc43473630)

[3.2.13 JSON 18](#_Toc43473631)

[3.2.14 GSON 20](#_Toc43473632)

[3.2.15 OAuth 1.0 Revision A 20](#_Toc43473633)

[3.2.16 OkHttp 21](#_Toc43473634)

[3.2.17 Open Bank Project API 22](#_Toc43473635)

[3.2.18 Google Sign-In 23](#_Toc43473636)

[3.2.19 Google Biometrics 23](#_Toc43473637)

[4 Solution Development 25](#_Toc43473638)

[4.1 Requirement specifications 25](#_Toc43473639)

[4.1.1 Definitions 25](#_Toc43473640)

[4.1.2 General notes 26](#_Toc43473641)

[4.1.3 Description 26](#_Toc43473642)

[4.1.4 Non-functional requirements 27](#_Toc43473643)

[4.1.5 Functional requirements 27](#_Toc43473644)

[4.2 Solution architecture 29](#_Toc43473645)

[4.3 Detailed design 30](#_Toc43473646)

[4.3.1 Android Application Architecture 30](#_Toc43473647)

[4.3.2 Google Sign-in flow 33](#_Toc43473648)

[4.3.3 Link bank account flow 34](#_Toc43473649)

[4.3.4 Local Room Database Design 35](#_Toc43473650)

[4.3.5 Remote Database Design 37](#_Toc43473651)

[4.4 Implementation 38](#_Toc43473652)

[4.5 Testing 57](#_Toc43473653)

[5 User Guide and Experimental Results 61](#_Toc43473654)

[5.1 Usage 61](#_Toc43473655)

[5.2 Experimental Results 72](#_Toc43473656)

[6 Conclusions 78](#_Toc43473657)

[6.1 Future improvements 78](#_Toc43473658)

[7 References 80](#_Toc43473659)

# Introduction

## Motivation

Most people are usually neglecting how they spend their money. In the past decades the card payments are continuing to grow in popularity and people are tending to spend more. This is due to the impression that our money is only a number on the screen. This fact is leading us to spend more and to care less about their importance.

All over the world, too many people with different income amounts, are struggling to manage their budgets. I am also seeing around me, that the majority of people are only thinking about how to make more money to solve their financial issues, but they don’t realize this is redundant if their debts will increase proportionally.

Nowadays, mobile phones are part of our everyday life. We carry them everywhere we go, we check them on average 60 times a day and we spend almost 3 hours every day on the phone [1]. According to Statista [2],3.5 billion people are owning a smartphone and this number is growing with 300 million people per year. This tool, which we have in our hands every day, it is capable to do a lot of things which makes our life easier. From checking the weather, reading news, making bookings, calls, meetings to doing payments, checking balance and even help us to improve the way we are managing our budget.

Banks are starting to incorporate into their mobile applications tools like “Analytics” in order to improve transactions tracking, money saving and budget administration. This is very useful in case of having all the bank accounts only at one bank. But things get more complicated when we have multiple types of bank accounts (e.g. credit, debit, savings) at multiple banks. Also, bank applications usually do not allow users to add transactions to those analytics as for example to track cash payments.

What defines 2020 is the power of getting information. We are now one click distance to find an answer for what we need. In bank industry, this thing is very new. Banks used to hold the data very secured, only for them or eventually for partner banks. Today, thankfully to the initiative of Open Banking, we can have third party applications made any organization, which are able to display to the end users, data about their bank accounts and even allow them to make payments.

## The problem

Firstly, the root cause of all our financial issues is the lack of financial education. We are not used to write all the incomes and expenses down to a paper, to eliminate the unnecessarily monthly payments and to learn how to avoid useless acquisitions. Surely, at one point of our life, we tried that, but this is not easy to do and maintain, and most of the people quit as soon as they start.

Even if we are determined to do that, in nowadays, it is almost impossible to keep that tracking on a paper. We would need to use special software, which is generally hard to use for the average user and which usually require a monthly subscription. Writing down every transaction we made is hard and requires a lot of time. To keep the tracking of a family expenses is even harder.

Simply writing down the transactions on the mobile phone every time we make one and sum up at the end of the day with reports gathered from each bank account will never work. If every member of the family has more than one account at more than one bank, the tracking and the ability to improve the way we spend our money would require too much effort.

## The solution

This overwhelming process of keeping the track of expenses can be simplified a lot using the power of information.

Mobile phones and Internet are enablers for digitalization of almost everything. In our case, the incomes and expenses which used to be written down on a paper. The most important thing we have nowadays, is the information. How we get that information, how we process it and, the most important thing for a user, is how we display it.

From already 5 years, we can get that information from any bank. We can process information from any bank account and decide how to display it in a manner which helps the end users to evaluate their spending in relation to their income.

The solution I came with, in order to solve all the above issues, is an Android application in which any end user of a smartphone can add any type of transaction, set-up recursive transactions, categorize those in virtual wallets and link real bank accounts to them. By linking the bank account to a wallet, the application will assure automatic synchronization of transactions. All these functionalities are intuitive and easy to use by any Android user. Also, the application will offer all these functionalities for free as the income will be generated by applying the Freemium Business Model. It will contain only ads with minimal impact on user experience and some usage limitations on the key functionalities of the application.

I chose to develop the application on the Android platform because it has 70.68% [3] of smartphone market share, it is free to develop application and it is very cheap to publish on the Android Marketplace.

The application will enable user’s interest to manage their finances and will cover the lack of financial education we are currently have in our society.

# State-of-the-art

## Overview

On the market, there are a lot of applications available for tracking expenses. Many of them are destinated for businesses and for people which already have some experience on managing their budget. In general, applications designed for an average user, require a monthly subscription in order to use the application or some special functionalities like linking bank account and cloud synchronization. The applications that are free usually neglects the user interface and the user’s experience.

## Main competitors

Researching the Android and IOS Marketplace, I found three main potential competitors at this time. In the next paragraphs, I will make a comparison by exposing the benefits and detriments of using *Budgetize* or any of the following leaders in the Smartphones Marketplace.

One application I found which almost combines all the principles I had in mind when I developed *Budgetize* is *Wally*. *Wally* is available at the moment only on the IOS application store. I still want to compare my application to *Wally* because it was the only one, I could find, to be able to link my bank account for free and to import my history transactions.

*Wally*is free to use, but also has some paid functionalities. It has a wide range list of supported banks which can be linked to the application, it also allows to organize the transactions on categories and wallets, we can add transactions manually and it is able to display information about the introduced and imported data in a lot of ways, all of those for free. Until there we are sharing the same functionalities, but what differentiates *Budgetize* from *Wally* are features like creating any number of wallets and setting their own currency for free. But the most important advantage of *Budgetize* is the user experience.

From the first touch of *Wally*,I have been overwhelmed of too many configurations and views, the lack of animations, crashes in some cases and how unintuitive the application is to use at the first sight. For sure *Wally* has a lot more ways to display the information about the data we introduced/imported, has more configurations and it has some interesting features for the premium version, but the target of *Budgetize* is to be appealing to anyone who installs the application, to demonstrate it is very easy to manage the budget and to track the expenses.

Another big competitor is *Spendee*. The strongest elements of *Spendee* are the reports and the analytics, which are very well displayed and animated. The application has a medium difficulty to use and it is fairly intuitive. One important feature of *Spendee*, which *Wally* doesn’t seem to have at least in the free version, is the biometric access to the application. I consider that feature extremely important for an application which is handling sensitive data like bank account information.

Even having all the benefits from above, the downside of the application is that you need to add each transaction manually without having any other possibility in the free version. For importing transactions from bank accounts and to create multiple virtual wallets, there are multiple premium versions which require monthly or yearly payment, or a big one-time payment. Another downside is the design. The application’s design is very similar to most of the applications from Android market. There is too many information in each screen, and it is not displayed in a smart way, easy to understand and visualize. From my opinion, a budgeting application must keep a simple design, and to display only key information. More information should be incapsulated inside a secondary screen.

Only on Android *Spendee* has over 1.000.000 downloads and over 28.000 reviews with a rating of 4,1 of 5 [4]. Those numbers demonstrate that there is a need in area of financial management.

The last competitor I would like to mention is *Bankin’*. This Android application also has over 1.000.000 downloads and almost 40.000 reviews with a rating of 4.7 of 5. The biggest advantages of this applications are the ability to link the bank accounts, analytics, and multiple ways of displaying the transactions or view them based on their type. All those features are free, also the application have premium functionalities like a personal coach but unfortunately that content is only available in French. Also, other parts of the applications are displayed only in French and this cannot be changed. Another downside is that users cannot track all their spending because the application does not provide the functionality to manually add new transactions, it only provides the ability to import transactions from bank accounts. Even this may be sufficient for some users, this application has a limited number of banks and I was not able to find any supported bank from Romania. This makes the application to have no usage in some countries or for some users which have accounts at unsupported banks. The design of that application is basic, without having any innovation in my opinion, and this design pattern can be found in a lot of other applications, especially for budget tracking.

In Figure 2‑1 we have a competitive matrix, illustrating how *Budgetize* positions on the market based on the user interface, user experience and the available functionalities.

Budgetize

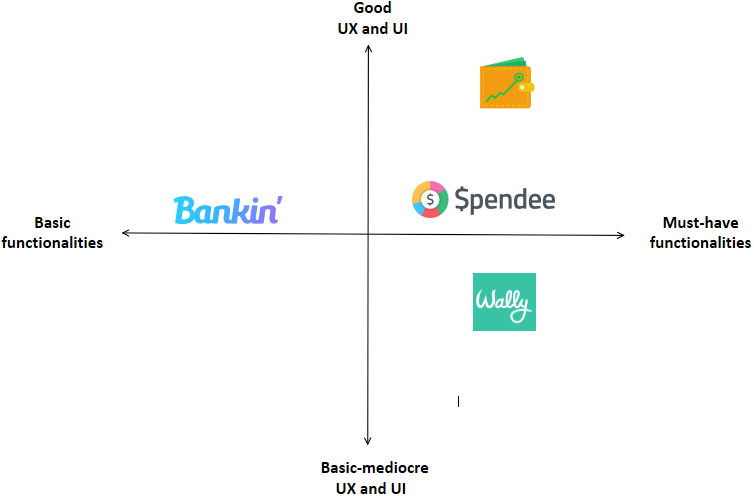


Figure ‑ Competitive Matrix

## Conclusions

*Budgetize* represents a combination between a collection of great functionalities, the easiness of using the application, an appealing user interface and a revolutionary design. The design of the application represents a concept of incapsulating the information in a smart way in each screen. The idea of this design is to maintain the simplicity of tracking the incomes and expenses. *Budgetize* combines multiple elements in order to gain a spirit of originality, clearly differentiating from the rest of the applications in its market segment. Also, *Budgetize* has been developed with security awareness, offering the end user credibility and safety.

Having into consideration the above-mentioned features of *Budgetize*, I definitely think those will create its own place on the Android Marketplace and enable the opportunity of acquiring a lot of users.

# Theoretical Foundation

## Concepts

### Object-oriented programming

Object-oriented programming is a programming paradigm consisting of organizing the data into objects. The object represents the instantiation of a class, considering the class as a blueprint.

When the object is instantiated, it can be “personalized” by giving values to the constructor. In general, classes can contain variables, constructors, destructors, getters and setters, methods, inner classes, etc. but those can differ from a programming language to another. The behavior of the class is defined through its methods.

The main principles of object-oriented programming are: Encapsulation, Abstraction, Inheritance and Polymorphism [5]. Those principles are used in order to build clear and modular structures and to have reusable code and more flexibility [6].

The encapsulation is realized by setting the access modifier to the class, constructor, variable method, or data member. Encapsulation is simplifying the maintenance of the application, makes the application easier to understand and it is also reducing the human errors [7].

Abstraction is very important because it reduces the complexity of the code, it is avoiding code duplication, increases the usability and it is increasing the security of the application. In Java for example, the abstraction is realized by using abstract classes and interfaces [8].

Inheritance is a concept which helps the developer into organizing the code in a hierarchy. This is assuring the reusability of the code and maintains a defined structure of the objects [5].

Polymorphism is an important concept because it is unlocking the ability to override and overload. These are realized using inheritance and child objects can override and overload parent’s methods. This concept consolidates the structure of the program and helps to reuse code.

### Android Activity Lifecycle

Every Android **Activity** has a lifecycle. A lifecycle means that the Android **Activity** can go through different stages as Figure 3‑1 illustrates. These stages are defined in order to save resources like random access memory. Another reason is to have applications with different priority. This is empowering the Android Operating System to know which **Activities** can be destroyed in critical situations or which **Activities** are not unused anymore and still consume a lot of memory or computational power.

Whenever the **Activity** is changed, it can be paused or destroy. The developer must be aware of this lifecycle in order to avoid application crashes. This can be done by using the Android **Activity’s** class callbacks: **onCreate()**, **onStart()**, **onResume()**, **onPause()**, **onStop()**, **onDestroy()** [9]. We are able to override every of these methods in our **Activity** and correctly handle what happens with **Activity** content in order to avoid crashes or memory leaks. Overriding **onCreate()** is mandatory in order an **Activity** to start and work.

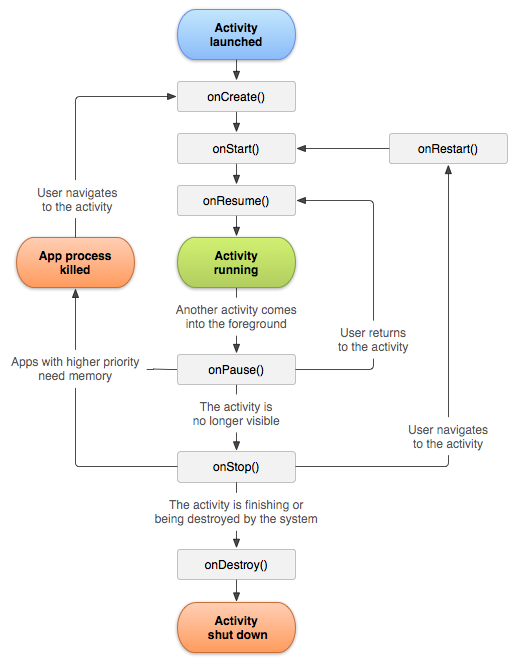


Figure ‑ A simplified illustration of the activity lifecycle (reference: [9])

### Model-View-ViewModel (MVVM)

The Model-View-ViewModel is an architectural pattern, consisting in separating the front-end from the back-end. This pattern is widely used in mobile and desktop applications because it is moving the complicated logic from the View into the View Model. This allows the View only to manage the way it is displaying the data read from the View Model.

This pattern is very important to be used in Android development in order to avoid “God” **Activities**. In Model-View-ViewModel, it is very important for the View Model to do not know anything about the View. The View Model role is to get data from the Model, process it and to offer methods available to provide this processed data to the View. The View must know how to use the View Model, what it can request and what it will receive. Its purpose is just to display the provided data without doing a lot of process with it. In the other way, the View can also be responsible to send the data to the View Model in order to process it, save it and even return a response. Communication between the View Model and the View can be realized using bindings, commands, or events (Figure 3‑2).

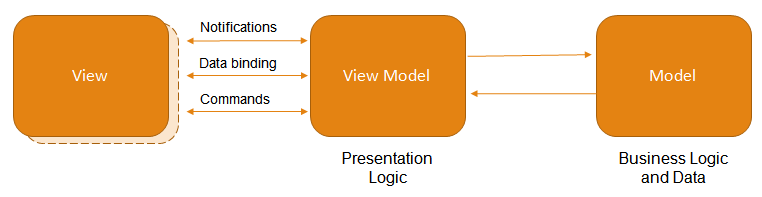


Figure ‑ Model-View-ViewModel diagram (reference: [52]).

In Android, the View from Figure 3‑2 is represented by the **Activity**. A very important property of the ViewModel in Android is that it will be created on the **onCreate()** method of the **Activity** and destroyed only when the **Activity** is finished (Figure 3‑3). It is not affected by actions like screen rotations, where an **Activity** can be destroyed for instance.

Moving the logic of the application in the ViewModel is a very good choice in order to have a well-organized code. Adding the back-end logic in the **Activity**, the programmer must make all that logic lifecycle aware, which will lead to adding a lot more code to the **Activity** and complicating the initial logic very much. This can be easily avoided by using the Model-View-ViewModel architectural pattern.

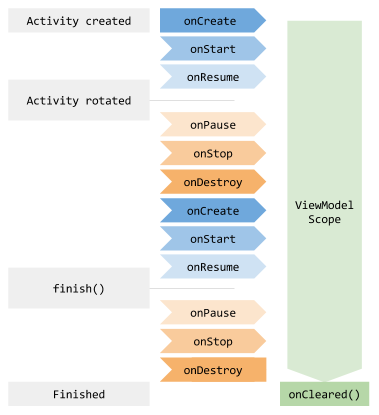


Figure ‑ Illustration of ViewModel Lifecycle (reference: [55])

Model-View-ViewModel is making the code reusable, modular, maintainable, and most important, it enables the possibility of collaborative work. Different types of programmers can work on the same screen for example, one very skillful on the user-interface design and one very good at the implementation of back-end logic. Any layer from Figure 3‑2 can be replaced easily, because of the modularity that Model-View-ViewModel enables. Because the front-end is separated from the back-end, we can easily maintain the code, refactor, add new functionalities, UI elements or even change the provider of data.

### Android LiveData

**LiveData** is a lifecycle-aware data holder which follows the publisher-subscriber architectural pattern. It is used together with an **Observer**, as **LiveData** is notifying the **Observer** when the data is changed. We can override the **onChanged()** method of the **Observer** and there we can implement what to happen when the data is changed. In this way, we can keep the **Views** from the UI permanently updated.

Being lifecycle-aware, it means that the **LiveData** will notify the **Observer**, from the **Activity**, only when the **Activity** is in an active state. When the **Activity** is destroyed, the **Observer** is automatically unsubscribed to the **LiveData**. Those actions are making **LiveData** to avoid crashes in the case the **Activity** gets destroyed. Also, because the **Observers** gets unbound automatically, they are cleanup from the memory in order to avoid memory leaks. As all this measurements are done automatically by the **LiveData**, there is no more manual lifecycle handling needed [10].

We can communicate between View, ViewModel and Model using **LiveData** very easily, with lifecycle-awareness. So, by using the Model-View-ViewModel pattern in combination with **LiveData**, we will get a clear architecture, easy to maintain and easy to develop on.

### SQL

Structured Query Language (SQL) is a programming language used and designed to work with relational databases. With SQL we can manipulate data from database. SQL can contain clauses, expressions, predicates, queries, statements, and insignificant whitespaces [11].

### RESTful Web Service

A RESTFul Web Services are web services build on the REST architectural style. This architectural style is based on a client-server communication, using HTTP as a communication protocol. The main philosophy of REST is to build lightweight, simple, and fast applications.

Using REST, servers can expose services in order to be accessed by various clients. Those services are called “endpoints”. REST, as well as SQL and HTTP, follows the CRUD paradigm. CRUD is an acronym for the operations create, read, update, and delete. Those are the essential operations to work with data. In REST, those operations are named the same as in HTTP: PUT, GET, POST and DELETE. [12]

Every REST endpoint is a mapping to one of the four enumerated operations. Using REST, we can easily build and use APIs. REST eases the way we transmit and use data.

### RESTful API

API stands for Application Programming Interface. A REST API is a collection of endpoints which are accessible over the HTTP.

## Used Technologies

### JAVA Programming Language

Java is one of the most popular programming languages in present according to GitHub [13]. It was released in 1995, developed by Sun Microsystems. Java is an Object-Oriented Programming language, where classes and objects are the core of the language. Java is mostly known because of its Virtual Machine, which is able to run on any computer architecture. This allow us to write Java applications which can run on any machine and any operating system. In order to run Java application, Java Runtime Environment (JRE) needs to be installed on the machine. When the Java code is compiled, it results in Java Bytecode which is executed by the Java Virtual Machine. Some platforms are able to directly execute Java Bytecode, without the need of installing JRE, as the hardware is built in this manner. [14]

An important principle of Java is to be simple, object oriented and familiar [15]. Java follows the C/C++ syntax, but it a simplified form, and it has less low-level access.

A java important feature is the automatic memory management handled by the garbage collector. In other Object-Oriented Languages, we have to manually deallocate objects from memory after we used them. In Java, this job is done automatically by the garbage collector. This kind of thing has some advantages and disadvantages. One advantage is that the programmer doesn’t have to worry about deallocation of the objects or about possible crashes if the deallocation has been done too early, thankfully to the automation of the memory management. This is speeding up the development process and keeps a cleaner code. A downside is that we cannot deallocate from memory objects which we are sure we are not using them anymore and they may occupy a lot of memory. This may lead to performance issues. Having more flexibility in C++ offers us the ability improve application’s performance but with the tradeoff that we have to write much more code and to be more careful with the memory allocation and deallocation. [16]

Android applications are built on Java. Another big advantage of Java is that it is continuously updated, and new versions are released constantly. They also release some versions with long term support.

### XML (eXtensible Markup Language)

XML is a markup language and a file format mostly known for being human and machine readable. XML files have a lot of applications in web area, for code generation, distributing and carry data, etc. It is mainly composed of tags and elements. XML can also have empty tags (tags which doesn’t contain any value between the start-tag and the end-tag). Inside the start-tag, we can have attributes. An attribute is a name-value pair. [17]

XML also has a feature named “XML declaration”, which is describing information about the XML file itself. For example, it can describe the version of the XML and the encoding. [17]

XML are used in programming a lot. There are a lot of XML parsers for each programming language, as well as XML builders. In Java for example we have the well-known SAX Parser and DOM Parser. Also, XML is used for example by the SOAP messaging protocol as message format. XML is also used intensively in Android Development. Using XML, we define the **Activity** layout, the **Fragment** layout, the **Manifest** file, or other files like colors, strings, styles, etc.

### Gradle

Gradle is a high performant build automation tool used to build any type of software. It is open-source and it follows the concepts of Apache Ant and Apache Maven. Regarding syntax, instead of using XML like Apache Maven, it uses Apache Groovy. [18]

Gradle is tightly bound to Java, as Apache Groovy is a Java compatible syntax and it runs on the Java Virtual Machine. This enables the Gradle ability to run on any platform, if the JVM is installed or if the hardware is able to execute Java Bytecode. Even Gradle runs on the Java Virtual Machine, it is not limited to run only Java project. A downside of Gradle is that it is supporting repositories and filesystems only from Apache Maven and Apache Ivy. [18] Gradle is already integrated in some big IDEs like Android Studio, Eclipse, IntelliJ or NetBeans. [19]

### Apache Maven

Maven, as Gradle, is a build automation tool used for multiple types of languages. In order to build a project with Maven, we have to create a Project Object Model (POM) file, where we write all the dependencies we need. POM files are eXtensible Markup Language (XML) files. A very important functionality of Maven is the ability to write and use Plugins. Plugins can be used to perform different tasks like running builds, tests, generating project files etc. [20]

### Android Operating System

Android is the mostly used operating system in the smartphone industry with a market share of 70.68% [3]. It has been launched on 23 September 2008, almost 1 year after the first IOS release, by Google. This operating system was used first on the smartphones, but now we can find it also in watches, smart glasses, cars, cameras, smart TVs, home appliances and many other devices. [21]

Regarding Figure 3‑4, the Android Operating System is made of 6 important layers. The core layer of Android is based on a modified version of the Linux Kernel, because it is providing the standard functionalities to communicate with the hardware. That core layer has implementations for handling Drivers and Power Management.

The next layer is called Hardware Abstraction Layer (HAL) and it is an interface between the Kernel and the upper level layers. This layer contains libraries which are used by the upper level Java APIs. Without HAL, we would not be able to use the Bluetooth, Sensors or Camera effortless directly from Java.

The third layer is the Android Runtime. This layer is assuring that each application from Android is running in its own Runtime (an instance of Android Runtime) and in its own process. ART was introduced in Android 5.0 Lollipop, prior to it, Android used the Dalvik process virtual machine. Both ART and Dalvik VM uses the Dalvik bytecode and stores the bytecode into .dex (Dalvik EXecutable) files. ART replaced the Dalvik VM because it is more performant due to its new features [22]. The first new feature is composed of two new compilation methods, Ahead-of-time (AOT) and Just-in-time (JIT), which are in fact just some compile optimizations [23]. The last two features of ART are the improved Garbage collector and better debugging support. [24]

The fourth layer the Android Operating System is represented by the Native C/C++ Libraries used in the Android Runtime layer and Hardware Abstraction Layer. Those, for example, offer support for drawing and manipulating 2D and 3D graphics in the Android applications. Those libraries can also be accessed by the developers using the Android NDK, in order to implement some functionalities or algorithms directly in C/C++ [25].

The fifth layer is the Java API Framework. This consists in Content providers, View System and Managers. Those are the APIs offered by Android, which the developers are using a lot for creating Android applications. For example, the Resource Manager is used to get non-code resources like layout files, drawables, strings, and colors.

The final layer is made of the System Applications. Here are the core applications of any Android device, for example the Telephone application, Agenda, SMS, Camera, etc. Those applications come pre-installed to any android smartphone.

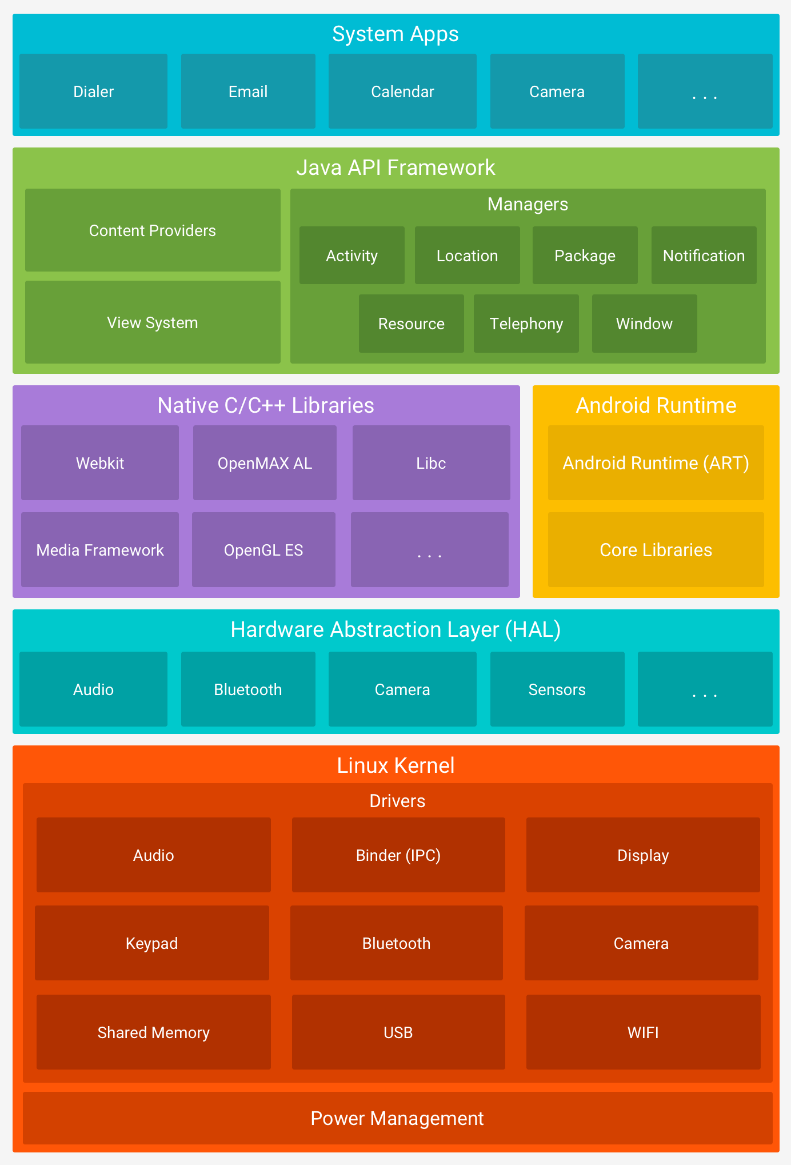


Figure ‑ The Android Software Stack (reference: [24] )

### Spring Boot

Spring Boot is a Java framework developed by Pivotal Team, which help the developers to build micro services. Spring Boot speeds up the process of building a micro service, by incorporating a servlet container like Tomcat, Jetty or Undertow, and being pre-configured and ready to run. Also, it provides mappings to create REST endpoints, easing the process of creating a Java REST API. It also provides well-written documentation, with code snippets, which guides the developer on how to create “Hello, World” applications. [26]

### IntelliJ IDEA

IntelliJ is an in integrated development environment developed by JetBrains, used to developed Java applications mainly. Beside Java, we can also develop applications in Kotlin, Groovy or Scala as all these programming languages uses the Java Virtual Machine and the code is compiled to Java Bytecode. It is written in Java and Kotlin and it is available for Windows, Linux and MacOS. [27]

IntelliJ has been released in January 2001 and currently there are two version available: Ultimate and Community. Community is the free version, with an Apache License 2.0, used to develop mainly Java applications, with less support and less frameworks available. The ultimate version is focused more for enterprise and web applications [28]. In my case, I used IntelliJ only to create a back-end Java server, where I am exposing REST services using Spring Boot. IntelliJ community edition fulfilled my needs, and I was able to realize what I purposed with it.

The main advantages of using IntelliJ are the design, the easiness of writing code using the Generate functionality and Smart code completion, the ability to navigate through the hierarchy of classes, the built-in version control system(VCS) and many more functionalities and tools available.

### Android Studio

Android Studio is an integrated development environment, build by Google, with the purpose of easing the process of developing Android applications. It is mainly based on IntelliJ IDEA made by JetBrains, with modifications and plugins which are used in Android Development. Android Studio has been released by Google on 16 May 2013. It is written in Java, C++ and Kotlin and it is available on Windows, Linux and MacOS.

Android Studio supports the same programming languages as IntelliJ IDEA, but from October 2017, it also supports the programming language developed by JetBrains, Kotlin. From 2019, this is becoming the default Android programming language, replacing Java. The main advantage of Kotlin is that it is fully interoperable with Java.

One of the main advantages of using Android Studio is the Android Virtual Device. This is a well optimized emulator which can simulate an Android smartphone on which we can install our application for testing reasons or debugging. Another important feature of Android Studio is the Layout Editor where we can view the layout in different screen configurations, we can drag and drop UI components offered by Android, and we can easily modify the XML layout files seeing the changes in real-time. Another difference from IntelliJ is that Android Studio has Gradle built in, and we can easily import any library we need from Gradle or Maven.

Summing up the advantages enumerated above with the fact that this application is updated periodically, and I observed great improvements for the emulator, I think Android Studio is the best choice to develop Android applications. [29]

### MSSQL

Microsoft SQL Server is a management system for relational databases. The first version of SQL Server 1.0 was released in 1989 by Microsoft [30] .It is mainly used to store and retrieve data, using Microsoft’s own version of Structured Query Language (SQL). [31]

In order to connect and access the Microsoft SQL server from Java, we need to use Microsoft’s Java Database Connectivity API(JDBC) driver.

### Room Database (SQLite)

Room database is a component of the Android Jetpack, meant to simplify the development process of an Android application, launched by Google in 2017. It is an abstraction layer between our application and an SQLite database, helping programmers to avoid boilerplate code. As Room is an Object Relational Mapping (ORM) library, we can easily map Java Objects to SQLite Objects. Another advantage using Room, is that the query errors are detected at compile time.

Room Database is developed to work with **LiveData**. Using Model-View-ViewModel in combination with Room Database, assures the data is observable directly in the database. This means that every change in the database will reflect in the UI (user interface). Also, it saves a lot of time, avoiding writing a lot of boilerplate code to achieve the same objectives, and most importantly, it maintains a clear structure and an organized code. [32]

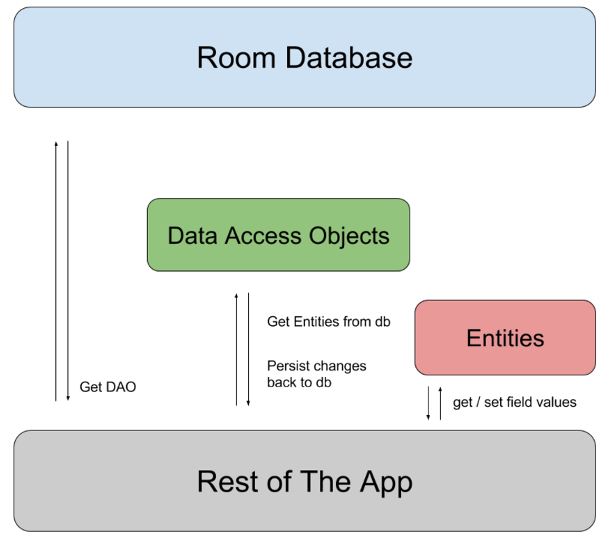


Figure ‑ Components of Room Database (reference: [32])

In order to communicate with the Room Database, we need to define Data Access Objects and Entities (Figure 3‑5). An Entity is a java class which represent the mapping between the Java object and a table row. In the entity we define the fields and its types of the object, create a constructor and getters and setters. With androidx library, we can define the table name, set primary keys, foreign keys, indexes, and map every member of the class to a column.

The Data Access Object is an interface where, using androidx library, we can define the queries to the database using annotations. The advantage of Room is that, at the compile time, we can see if we have any errors in the query written in the annotations. With those queries we can retrieve, insert, update, or delete or even observe data from the database.

We can build a new Room Database for our application by creating a class which has to extend the Room Database class and adding the annotation **@Database** above the class declaration. In this class we can also define the way to make Migration in case of the schema is changed.

### Room Explorer

Room Explorer is a 3rd party library taken from GitHub, used mainly to view the data from our application’s Room Database. This library allows us to start an **Activity** through which we can view, insert, delete, or update the content of our tables. We are even able to manually write and execute queries using this explorer.

This library is very useful to use for debugging, to see if data is missing when actions are performed, etc. This library has an Apache License 2.0. [33]

### Leak Canary

Leak Canary is an API a library used to detect leaks of resources which can cause **OutOfMemoryError**. It also has the feature to indicate the root cause of the leak, displaying it in its own application. We can simply import the library via Gradle, and the LeakCanary application will automatically install to the device that we deploy our application. At the runtime of our application, Leak Canary will keep track of each **Activity** and it will look if the resources are leaked. If it detects leaks, it will pop-up notifications, allowing us to investigate the root cause of each leak. We can also see information about the leaks in the Logcat. [34]

This library is very useful for debugging, testing and for increasing the quality of our applications.

### JSON

JavaScript Object Notation (JSON) is a data format used for transmitting key-value objects. This data format is considered to be lightweight and human readable. It was standardized in 2013 as ECMA-404 [35]. Even it was derived from JavaScript, almost every modern programming language has libraries to parse and generate JSON files [36]. Using those libraries, we can easily serialize and deserialize our specific programming language objects or even arrays of objects. Those processes are light and are enabling the possibility to interchange objects between different programming languages and synchronizing different type of databases, even from SQL to NOSQL, using an intermediary server, very easily and efficiently. Considering all of the above advantages, JSON is for a reason one of the best and most used data formats in present.

In Figure 3‑6, we can see the format of an JSON object. Every object starts and ends with a curly bracket. An empty object consists of a whitespace. Otherwise, we have a whitespace after the start bracket, a string, which represents the key, another whitespace, a colon which separates the key from the value, and the value. Then, we may have a comma which is marking the start of another key-value pair, or the end bracket which marks the end of the JSON object.



Figure ‑ JSON Object (reference: [53])

In Figure 3‑7 we have the representation of an array in the JSON format. Every array in JSON starts and ends with a square bracket. Between the brackets we can have an empty array, marked by a whitespace, or we can have values separated by commas.

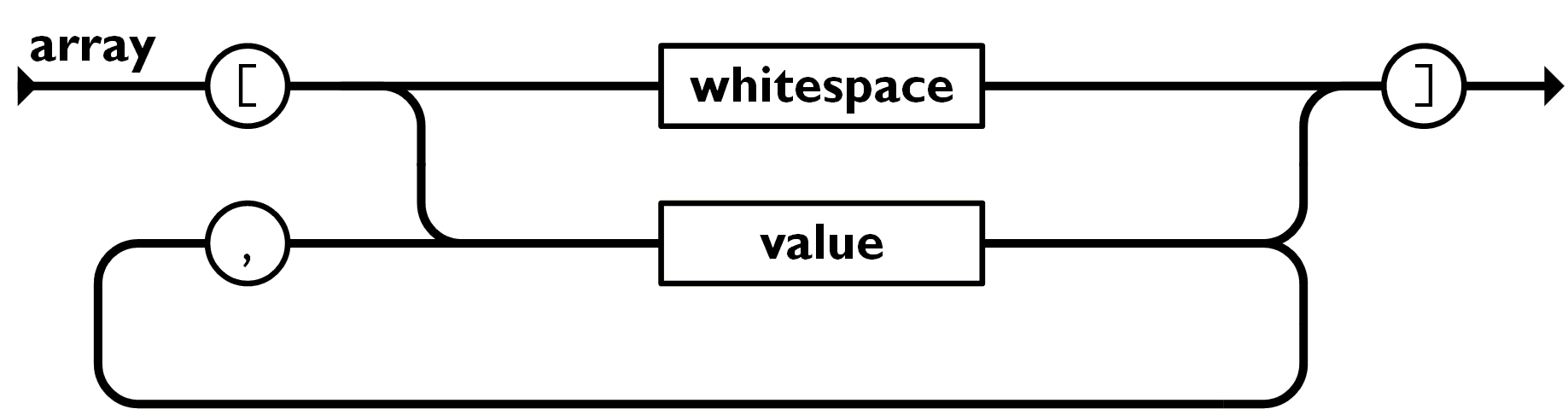


Figure ‑ JSON Array (reference: [53])

In Figure 3‑8, we have the possible types of values in the JSON format. Every value starts and ends with a whitespace and in between the we may have a string, number, object, array, a boolean value, represented by true or false, or a null.

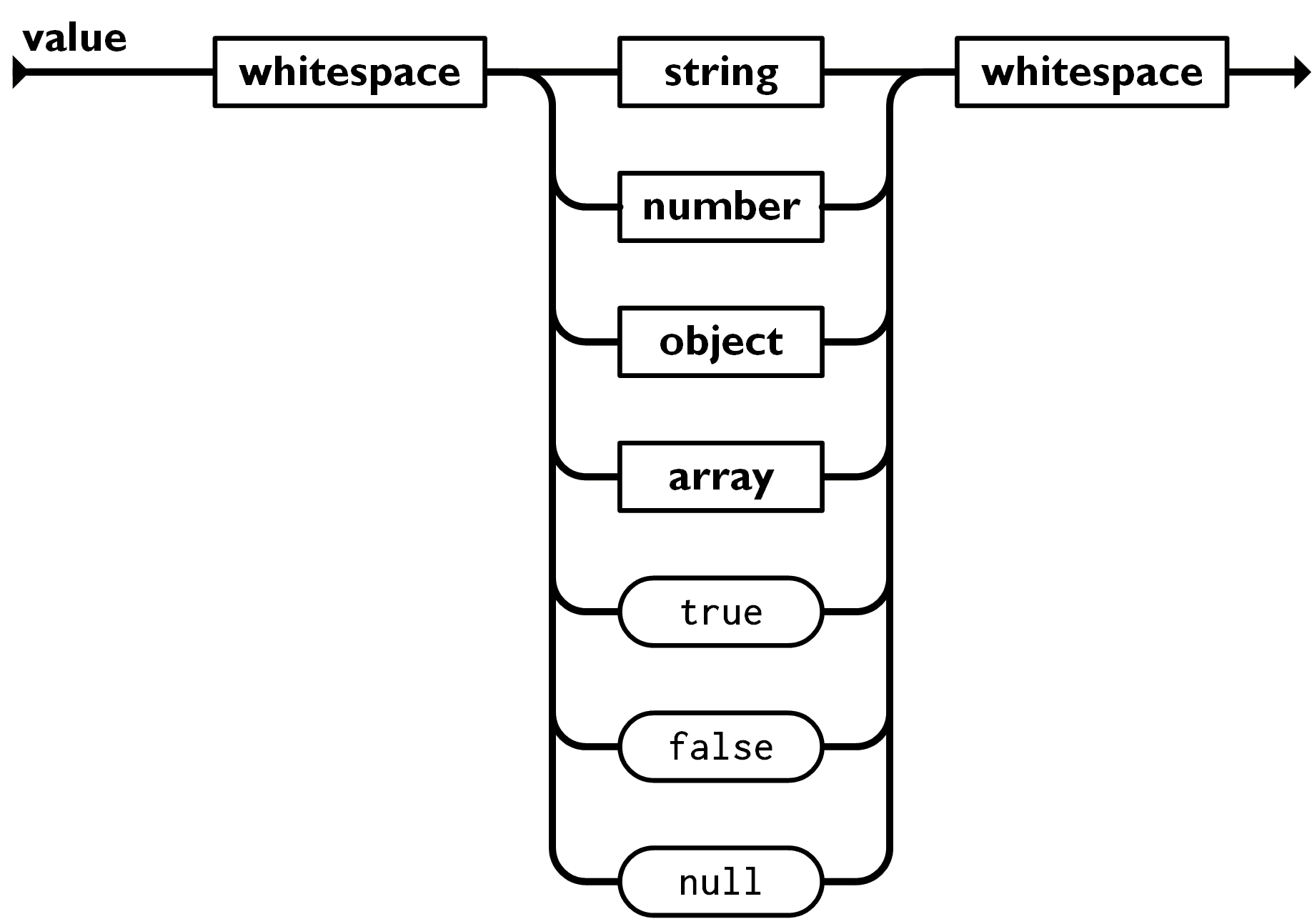


Figure ‑ JSON Value (reference: [53])

### GSON

GSON is Java library used to serialize and deserialize Java objects to and from JSON objects. This library has been developed by Google and it has an Apache License 2.0 [37]. Using GSON we can very easily synchronize databases, send objects between Android **Activities**, gather data from other REST APIs, and many other purposes.

### OAuth 1.0 Revision A

OAuth is an authorization protocol which allows a Consumer to access the protected resources of a Service Provider, without requiring the Users to introduce their Provider Credentials to the Consumers. The service provider is a web application which allows access via OAuth. The User is represented by an individual who has an account at the Service Provider. The Consumer is an website or an application which access the Service Provider using OAuth on behalf of the User. [38]

OAuth 1.0 Revision A appeared on 23 April 2009 in order to fix some security flow. It is widely used by a lot of companies to allow their users to provide third party applications access to their data. The advantage is that these users will provide their credentials directly to the provider. Also, they will be asked to give the consent that the third-party application will be able to access and process their data, without knowing anything about their credentials. [39]

In Figure 3‑9 we have the authentication flow for OAuth v1.0a. This is the “process in which Users grant access to their Protected Resources without sharing their credentials with the Consumer” [38]. OAuth uses two types of tokens, generated both by the Service Provider. The first one is called the Request Token. This token is used by the Consumer to ask the user to grant access to the Protected Resources. The Request Token must be used only once, without being used for other purposes, and it is recommended to have a limited lifetime. The second one is the Access Token. This token is used only to access the User’s Protected Resources on his behalf. Using this token, the Service Provider may limit access to only some Protected Resources and may define its lifetime. Also, the Service Provider must allow the User to revoke Consumer’s access anytime. Looking at the Figure 3‑9, there are three main steps in order to realize the authentication. In the first step, the Consumer request and obtain an unauthorized Request Token. Then, the consumer must redirect the User to the Service Provider in order to authorize the Consumer to have access to User’s Protected Resources. At the last step, after the User’s granted access to the Consumer, and the consumer received the authorized Request Token, the Consumer has to exchange this token for an Access Token. Then the User is allowed to access Protected Resources from the Service Provider, through the Consumer application. [38]

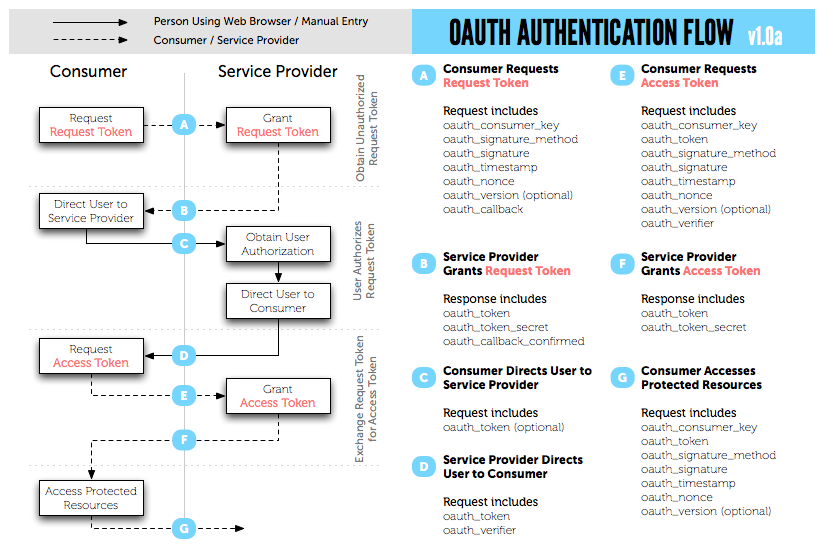


Figure ‑ OAuth v1.0a Authentication Flow (reference: [38])

### OkHttp

OkHttp is a 3rd party library released by Square Inc in May 2013 with an Apache License 2.0. The initial version was 1.0.0, but their last version is 4.7.2 released on May 2020. Versions 4.0.0+ works on Android 5.0+ and Java 8+ and adds support for TLS 1.2 [40]. In this project we will use the latest version (4.7.2) as it is a stable version with a lot of features and fixes. [41]

Its main purpose is to allow Android applications to make http requests. These requests can be Synchronous or Asynchronous. It is very easy to build a request, to send it and to intercept the response. OkHttp is a library which saved me a lot of time and effort, because it has simplified the way I consume my REST services from my back-end server, and the process of downloading the bank logos.

### Open Bank Project API

Open Banking Project is an open source middleware platform developed by TESOBE. It is meant to provide APIs on which Banks and fintechs can connect easily. The APIs are thought in order to be EU PSD2, and other regulations, compliant. PSD2 is a European regulation which ensure the banks that are allowing third party applications to access their customer data is done on the customer’s behalf, with his consent, in a secured and reliable way. [42]

This kind of platform helps the banks to connect to third party applications and enables an entirely new opportunity for fintechs. All these actions are in the benefit of the end user. Now he is not anymore dependent to the bank applications. He can merge data from multiple banks in one application, manage the expenses better, visualize in more ways his expenses, detect frauds by himself, has a personal AI assistant to improve his spending and many other opportunities which fintechs may offer.

This Open Bank concept is relatively new on the market, as before 2015, banks where holding their customer’s data only for them, governments and maybe for some partner banks. Now, this concept is offering to the customer the opportunity to decide which application can access their data, in order to process it in different ways, analyze it and display it in a more efficient way. This concept also, offers a higher transparency in the banking world.

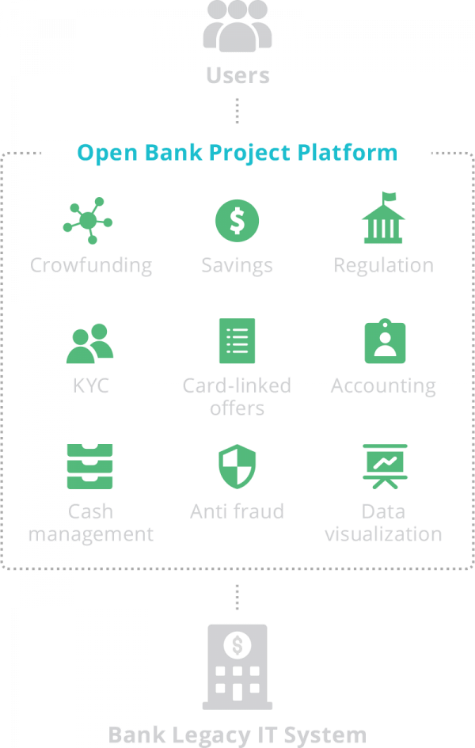


Figure ‑ Open Bank Project diagram (reference: [43])

There are a lot of banks which in present are offering PSD2 APIs. The problem comes when a fintech want to make an application which offer the customer the ability to use his data from any bank. Every Bank has developed the APIs in their own way, so for around 25.000 Banks worldwide, it may be impossible for fintechs, to have a different implementation to access each bank APIs. The purpose of the Open Bank Project is to standardize those bank APIs. This Open Bank Project platform is meant to offer over 270 common APIs which banks usually offers. To this platform the old banks systems may connect with minimal implementation effort, and from the other side, the fintechs can consume those rest APIs very easily, as Figure 3‑10 illustrates. This means the fintechs will require only one implementation for a given API, and this implementation will be able to access user’s data from any bank supported by Open Bank Project. [43]

Open Bank Project offer a sandbox with test data and test banks, which any person or fintech can use in order to develop financial applications. Having the implementation done on the sandbox, the movement to the real open banking system is easy to make. In order to have access to the real system and real bank data, the developer must have a company and to obtain eIDAS certifications.

### Google Sign-In

Google Sign-In is an API developed to provide a secured authentication using the already associated Google accounts from the device. This API uses the OAuth 2.0 flow and token lifecycle to grant 3rd party applications access to Google user’s account. The integration of this API in an 3rd party application is done very easily by using Google’s website   
“Console Developers”. There we can obtain a Client ID for any google API and use it in our application to have access to that API. We can also use Google Sign-In API with a back-end server in order to validate the person who’s making the requests. Using the API, we can generate an ID token which contains the encrypted user’s data. We can decrypt the user’s data by using the private key from the server. This ID token can be viewed as a Request Token. After we have validated it, we can generate a Session ID for the further requests of that authorized user. [44]

### Google Biometrics

Google Biometrics is an API available on Android Jetpack. Using this API, we can easily access and use the biometric hardware the phone is providing, in order to validate the user. We can restrict the access to our application, which significantly improves the security.

We can use **BiometricManager** class in order to find if there is any biometric available on the device. **BiometricManager** is the successor class of the **FingerprintManager** [45]. **FingerprintManager** was released in Android 9, but it was replaced in Android 10 by **BiometricManager**, as the new biometrics hardware technologies appeared in the smartphone industry. If this validation is successful, and there is a valid or multiple biometrics available on the device, we can forward use the **BiometricPrompt** class. With **BiometricPrompt** class, we can show the dialog screen which request biometrics in order to make the authentication, we can validate the authentication, and we can implement follow-ups in case of different authentication statuses.

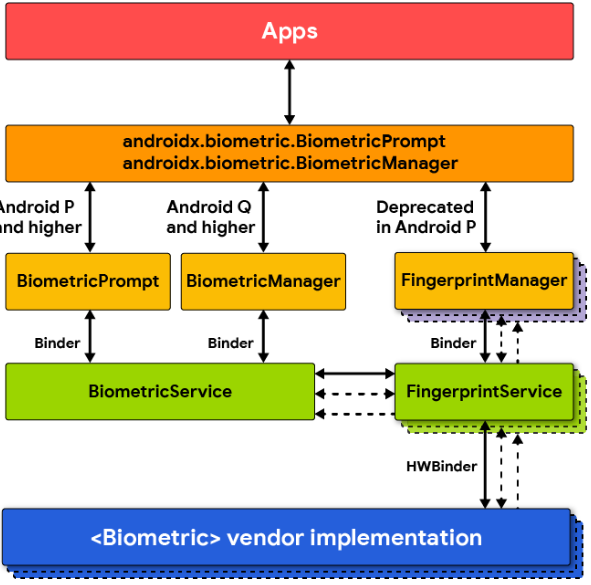


Figure ‑ BiometricPrompt architecture (reference: [46])

In Figure 3‑11 we can see the architecture of **BiometricPrompt** and the authentication flow. [46] We can also allow fallback to non-biometric credentials, if the biometric is not working for instance, the user may authenticate using his PIN, password, or pattern.

### GridListViewAdapter

GridListViewAdapteris an UI third-party library made by Biraj Patel [47]. It helped me to develop the screen where I am displaying all the available banks to link (Figure 5‑17). I have created a java object called **AvailableBank**, with which I can inflate the list. Each object from the list is saved in a **ViewHolder**, in order to be able to identify the bank that the user selects.

### AndroidExpandingViewLibrary

AndroidExpandingViewLibrary is an UI third-party library made by Diego Bezerra [48]. It offers an easy solution to personalize an **ExpandingListView**. Each element from the list has an icon which, on pressing, it expands a sub list. The library is modular, and easy to personalize, helping me to achieve the design I thought of. I used it to display the categories and transactions of a wallet (Figure 5‑10) and the screen from where we import transactions (Figure 5‑22).

### ViewPagerCards

ViewPagerCards is an UI third-party library made by Rúben Sousa [49]. This library offers support to easily personalize a horizontal **ViewPager** with **CardView** elements. The design, as the author says, is inspired from a screen of Duolingo. I used this library in order to display the added banks accounts screen (Figure 5‑18).

# Solution Development

## Requirement specifications

### Definitions

**Cloud Database:**

MSSQL database from the server side.

**Local Database:**

SQLite database from the user’s Android smartphone.

**Basic functionalities:**

A set of functionalities the user has access by default, from the first start of the application, without needing an internet connection, an account, or media files permissions.

**Special Functionalities:**

A set of functionalities for which in order to be used, the user may require internet access, sign-in with google, access to read and write to media storage and consent that the application can access user’s bank account on their behalf, only to import the transactions, without requiring the application to know anything about their bank account credentials.

**User:**

Anyone person who installed and uses *Budgetize* application.

**Wallet:**

Entity which encapsulate categories and transactions. It also has assigned a name, a financial status, which is automatically updated, a financial goal, and a currency.

**Category:**

Entity which can hold transactions. A category can have a name, a description, and an icon.

**Transaction:**

An entity which can be manually created by the user or imported from a bank account. A transaction has a name, an amount, a currency, a date, can be assigned or not to a category, can be an income or an expense, and can be recursive or not.

**Orphan Transaction:**

A Transaction that is not assigned to any category.

**Financial Status:**

From my perspective, this term refers to how much money a user currently has.

**Financial Goal:**

From my perspective, this term refers to how much money a user wants to achieve.

**Focused Wallet:**

As the wallets are displayed with a horizontal **ViewPager**, each wallet represents a page of the **ViewPager**. Only a page is displayed at a time on the screen, and user can navigate through them by swiping to left or right. I call the Focused Wallet, the page on which the user currently is.

**Transaction Screen:**

The screen which contains the categories and the transactions of a wallet.

### General notes

If any text cannot be displayed entirely, the first letters or words will be displayed followed by ellipsis. In order to view the entire text, the user must long press on it, and a tooltip with the entire text will pop-up.

From a graphical designer point of view, strong shades of green and red can never be combined in a screen. I have to mention that all the green, red and yellow views, are dynamically set from the code, based on the positive, negative, or neutral value from the view, in order to provide a strong feedback to the user. Green represents something good, expressing a good feeling to the user. Red serves as something bad, which needs to be corrected, or to reflect on. Yellow means something intermediary, or mediocre. Not good, not bad, but something to consider.

The idea of having multiple wallets, is to offer families the opportunity to have all the incomes and expenses in one place. With the cloud functionality, the data is permanently synchronized between all the devices.

### Description

*Budgetize* is an Android Application designed to help people to track their expenses and achieve a budget goal. Its purpose is to display the information in a simpler way than most of the applications from the Android Marketplace. Also, the ability to import transactions from any bank in the world, is a must have in order to keep the tracking of the expenses, without having to waste a lot of time writing them down manually. It also offers the ability to manually add the transactions in the application, in the case of cash payments, or if the user does not want to use the import transactions from bank account functionality.

The application has been developed using Android Studio, which is the perfect tool for realizing the back-end and the front-end. The layout editor helps the developer to view and position each UI element on different screen configurations. For the back-end, the IntelliJ functionalities increased the development process. Also, the debugging tool is very advanced and in combination with the Android Virtual Device (AVD) Manager, we can deploy our application on multiple emulated devices with different screen sizes. We can even deploy the application on real devices, and we have the ability to debug our application on them.

The application communicates with a Spring Boot back-end server. It uses HTTP as a communication protocol, and JSON as communication data object. We are using the back-end server in order to validate user’s access to the Special Functionalities, and to allow the users to keep their application data synchronized with a cloud database. We are using HTTP and JSON, because we already need to consume REST APIs from other third-party providers, and because it is easy to create and consume REST services. Back-end server has been realized using IntelliJ IDEA and Spring Boot framework.

The data is stored locally on an SQLite database, and can be saved on the server on an MSSQL database.

### Non-functional requirements

In order to use *Budgetize*, an Android Smartphone is needed, running Android version 9.0+ (PIE).

Internet access is needed when the Special Functionalities are used, to download bank logos, link bank accounts, import transactions, google sign-in, and data synchronization to cloud.

At the first start of the application, the user must allow *Budgetize* to access phone’s media files. This requirement is needed for downloading bank logos and save them internally, in order to reduce the internet traffic consumption and to allow us to deploy new supported banks, without the need of updating the application.

User’s phone must feature any biometric security hardware to allow the user to login in the application. This is to enforce the security of the application, taking into account, the application may contain sensitive user data like bank accounts and transaction history.

User is required to have a registered biometric login.

User consent is needed to allow *Budgetize* to access Bank Account data when the link bank account functionality is used.

### Functional requirements

Users must be able to:

* Access the application only after a successful authentication. Biometric authentication is the only accepted method of authentication. Other methods as pin, pattern, etc., are forbidden. This is to enforce the security of the application and to maintain.
* Create multiple Wallets, Categories and Transactions.
* Change the currency of a Wallet.
* Edit any data of a Wallet, Category or Transaction.
* Delete any Wallet, Category or Transaction.
* View added Transactions.
* Filter the added Transactions.
* Change the currency of a Wallet.
* Link/unlink any bank account from any bank in the world.
* Provide consent to the application to access his bank account data without to provide any bank login credential to the application.
* Synchronize a bank account to any wallet.
* Stop the synchronization of a bank account to a wallet.
* Revoke the consent on which the application can access any of his bank account.
* Remove a bank account from the application.
* Login with Google account.
* Logout from the Google account.
* Keep the application data synchronized to the cloud.
* Access his data from multiple devices if the cloud functionality is used.
* Allow other google accounts users to view or edit his application data.
* Revoke other google accounts users to view or edit his application data.
* Know if the data is synchronized to the cloud, in progress to synchronize or unable to synchronize.

Application functional requirements:

* A Category is an entity of a Wallet, which can contain unlimited number of transactions. Categories have, as persistent data, a name, a description, and an icon. At the runtime of the application, Categories must display, beside name, description, and icon, the sum of all the contained transactions.
* A Transaction is an entity of a Wallet, which can be assigned to a category or not. Persistent data of a transactions is: name, amount, category, date, occurrence, type and currency. Type of the transaction can be income or expense. Occurrence can be none, every day, every two days, every working day, every week, every two weeks, every month, every two months, every three months, every six months, or every year. Transactions must display all these data to the user. Transactions can be created by the user manually or imported from the bank.
* A user must be able to filter the added transactions based on a selected date and timeframe. For example, the user must be able to view all transactions from day x, month x, or from year x.
* The user must be able to select the Currency of the Wallet. This will display all the transactions from the respective wallet in the chosen currency.
* When a bank account is synchronized to a Wallet, at the synchronization time, all the transactions must be imported to the respective Wallet to a new Category, where the name is the bank account name, the description is the bank name, and the icon must represent a card. The color of that category must be different from the normal convention of positive, negative, and neutral. It must be a color suggestive to a bank account like gold, silver, bronze. Also, a synchronized bank account, must ensure that every new transaction from the respective bank account is added automatically to the respective Wallet and Category.
* If a user has activated the cloud functionality, all the inserts, modifications, or deletions from the local database must be replicated to cloud database.
* Requests to the server must be validated.
* Communication between Android Client and the server must be done over a secured channel. Also, communication with sensitive data between the Android Application and other APIs must be done as well over a secured channel.

Design requirements:

* The design of the application must be simple. Every screen must contain minimal important information and embedded more details in sub screens.
* Application must have a predefined color scheme, which are going to be used in each screen. We must have colors which reflect the positive, negative, or neutral balance of each wallet, category, and transaction.

## Solution architecture

The architecture I designed for this project can be seen in Figure 4‑1. We have the Android application called *Budgetize*, an SQLite Room Database, the Spring Boot Server, the MSSQL Database from the server and several third-party APIs.

First, we have the brain of the architecture, the Android application. All the calls and requests to the other components from Figure 4‑1 are initiated from the Android application. The smartphone of the user contains an SQLite database called Room Database which can be used by any application. Each application has access only to its database data. All the persistent data of the *Budgetize* is saved in this database.

The Android application also communicates with several third-party APIs. The first one is Google Sign API. This API is used to provide the user a secured and easy way to authenticate to the application. The authentication enables new functionalities like cloud synchronization and bank account transactions synchronization. The second one is Google Biometrics. It allows us to validate user’s access to the application based on his Android configured Biometrics. In order to access any Google API, we need to use the OAuth 2.0 protocol for authentication and authorization.

The last one is the Open Bank Project API, which provides to the user the opportunity to allow *Budgetize* to access his bank account data on his behalf. The communication between *Budgetize* and Open Bank Project is realized using HTTP, using as communication object JSON, and the authentication and authorization is done via OAuth 1.0 protocol.

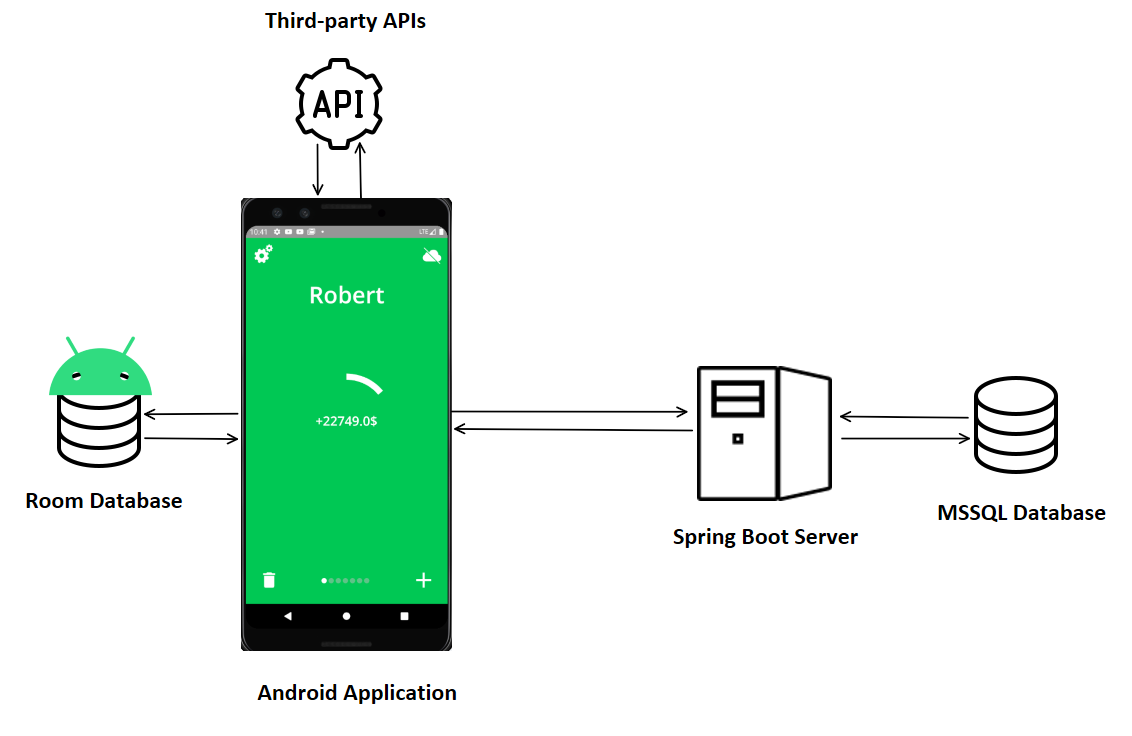


Figure ‑ Solution Architecture (icons reference: [54])

If the user is logged in with his Google Account to the application, he will have access indirectly to the back-end Spring Boot Server. This server is used to validate the Google Sign-in Token ID, and to provide the user a Session ID. Based on this Session ID, the server will validate the incoming requests in order to provide access to resources. The back-end server communicates with an MSSQL Database in order to store users and their Android application data.

## Detailed design

### Android Application Architecture

The Android application has been firstly built, without following any architecture or pattern. After the **Activities** grown very much, the logic got very complicated and the application became unstable, I decided to refactor the entire application. I started to study a lot about how Android works and how to design Android applications smartly. I found out on Developers Android website, a very good guide [50] on how the Android application should be structured. The following structure has been slightly modified in order to fulfill to the needs of the application.

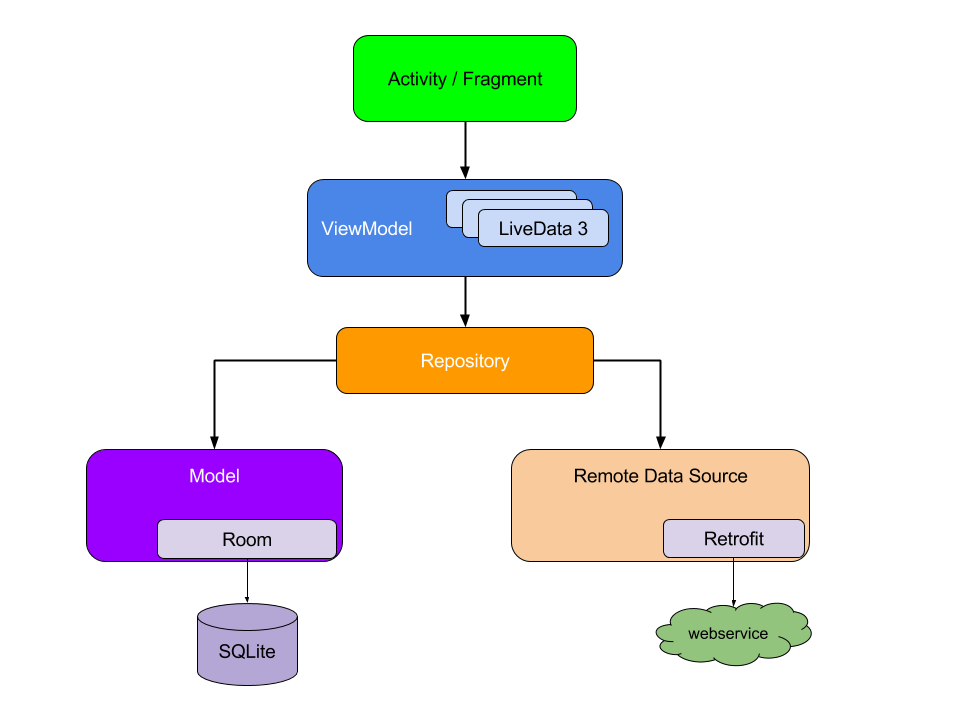


Figure ‑ Android application architecture diagram (reference: [50])

The architecture from Figure 4‑2 respects the Model-View-ViewModel architectural pattern. It follows the “Separation of concerns” concept, as it indicates that we should not write all our code into the **Activity**/**Fragment** and the back-end should be separated from the front-end. In this case, the front-end is represented by the **Activity**/**Fragment**, and the back-end by the rest parts of the figure.

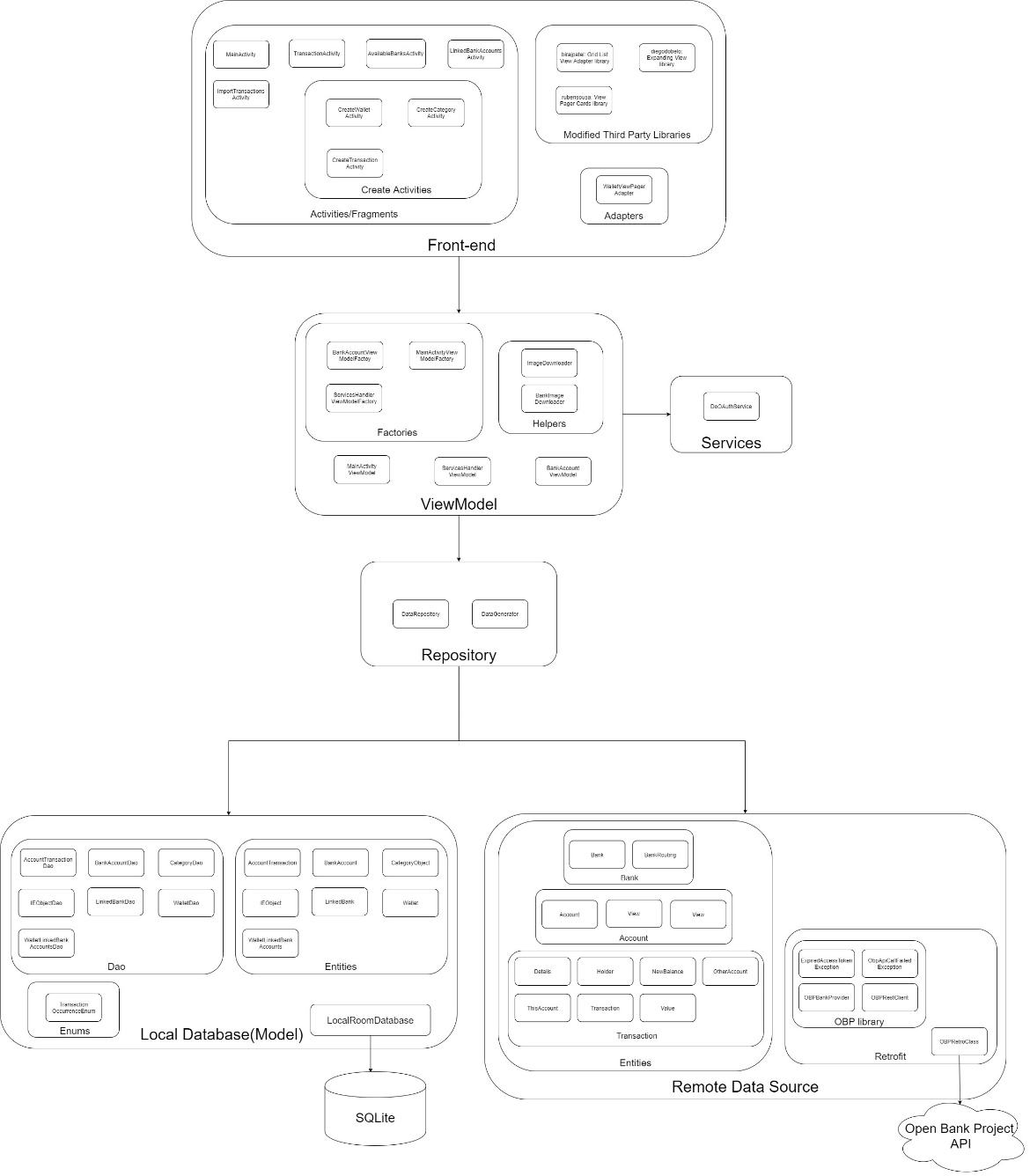


Figure ‑ *Budgetize* Android application architecture diagram (Created with: [56])

In Figure 4‑3, we can see *Budgetize*’s architecture, with all the packages and their contents. The project structure is based on the architecture from Figure 4‑2.

I will start to explain the architecture from Figure 4‑3 in a top-down manner. The front-end package contains **Activities**/**Fragments**, Adapters and some modified third-party libraries I used (Figure 4‑4). The Adapters and third-party libraries are used by the **Activities**/**Fragments** in order to display the data from the **ViewModels** in a simple, beautiful and modular way. Everything that is after the front-end is considered to be the back-end of the application.

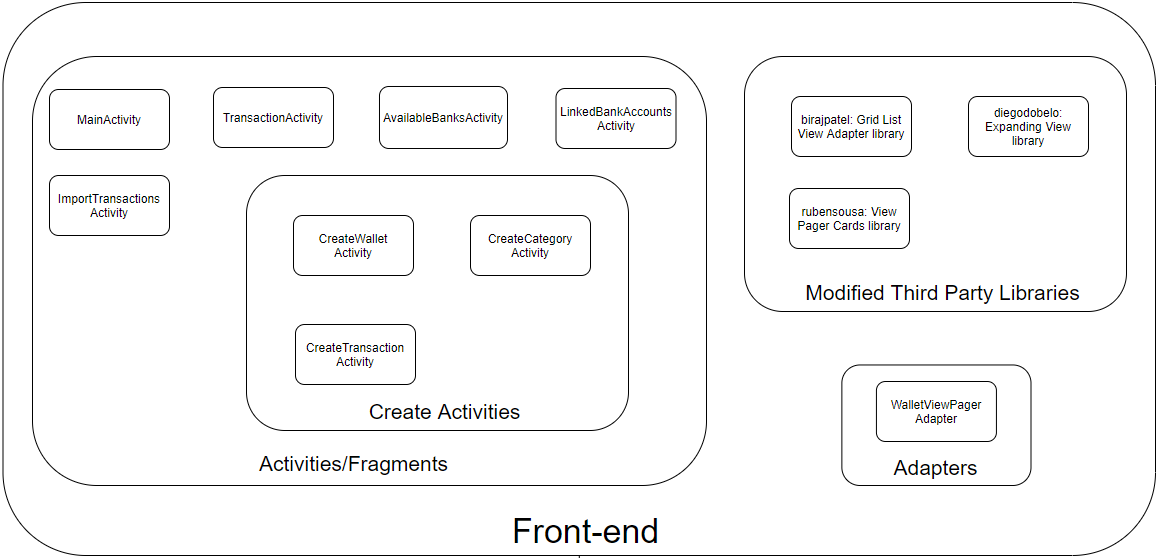


Figure ‑ Front-end (Created with: [56])

The **ViewModel** is responsible to handle the application logic, retrieve data from the Repository, process these data if needed before making it available to the front-end, and start/stop the Services. In the **ViewModel** package we also have the Factory of each **ViewModel**, and some Helper classes (Figure 4‑5).

In the right part of the **ViewModel** layer, we have the Services (Figure 4‑5). I started to create and use services, as a solution when the logic got too complicated in the **Activities**, but when I discovered the architecture from Figure 4‑2, and I started to replace those services, with an Model-View-ViewModel implementation. Currently, we have left only one Service after the refactorization. It is called **DoOAuthService**, and it is used to perform the OAuth authorization flow for the Open Bank Project API.

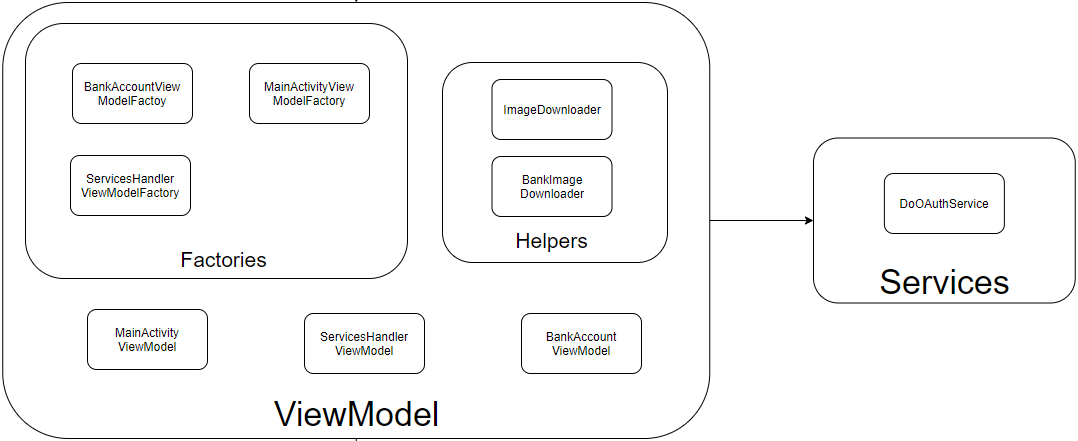


Figure ‑ ViewModel and Services (Created with: [56])

The next layer on the architecture is the Repository. This package represents an abstraction layer over the Model layer and the Remote Data Source layer. Repository package is mainly composed of the **DataRepository** class. This class offers us methods with which we can easily obtain the data we need in the **ViewModel**, without needing to know where these data come from. So, the **DataRepository** class access the needed Data Source and provides to the **ViewModel** the needed data. We can even implement here a logic to retrieve the data from multiple data sources, both local and remote, and combine these data to provide it to the **ViewModel**.

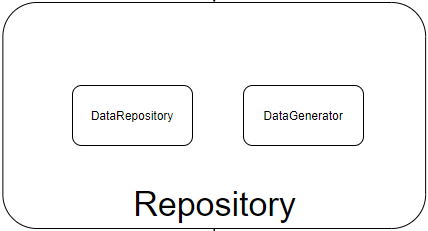


Figure ‑ Repository (Created with: [56])

The last layer is represented by the Data Sources. We have a Local Data Source (Figure 4‑7) which retrieves data from an SQLite Database from the device and a Remote Data Source (Figure 4‑8) with retrieves data from REST APIs. I included only the Open Bank Project API as a remote Data Source, because the communication with the other APIs is still performed directly from **MainActivity**.

The Local Database (Figure 4‑7), or the Model as it is displayed in Figure 4‑2, is the layer used to save the application data locally. We are mapping the Java objects to the SQLite database, by using the Room abstraction layer. In order to use this layer, for each Java object, we need to create a Data Access Object (DAO) and an Entity (the object from Java itself). DAO is an interface where we can define the queries to the SQLite database for an object. The Entity is a Java object mapped to an SQLite table. We also have some Enums, which are not mapped to the database, and a class named **LocalRoomDatabase**. This class extends the **RoomDatabase** class and contains all the logic of the Room Database. There we define the DAOs, the Entities, the builder of the database, the migration plan, the database seeds, etc.

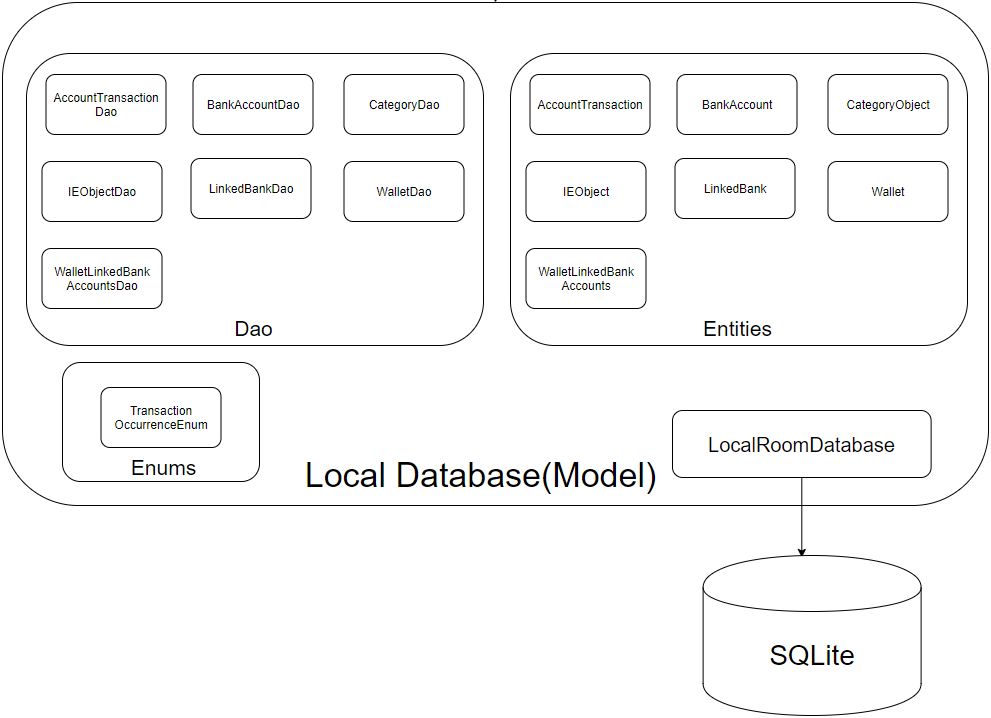


Figure ‑ Local Database (Model) (Created with: [56])

The Remote Data Source from Figure 4‑8 is represented by Entities and Retrofit packages. The entities are the objects defined by the used APIs. In our case we have the entities defined by the Open Bank Project API, used to retrieve data about banks, accounts and transactions. These objects are converted to the format we use in the Local Database, right after the retrieval. The retrieve is made using the **OBPRetroClass** from Retrofit package. The sub-package of the Retrofit is represented by the OBP library. This library is composed of some modified files from the basic application [51] which OBP offers. The purpose of that application is to demonstrate how realize the OAuth authorization and how to start the OBP API flow.

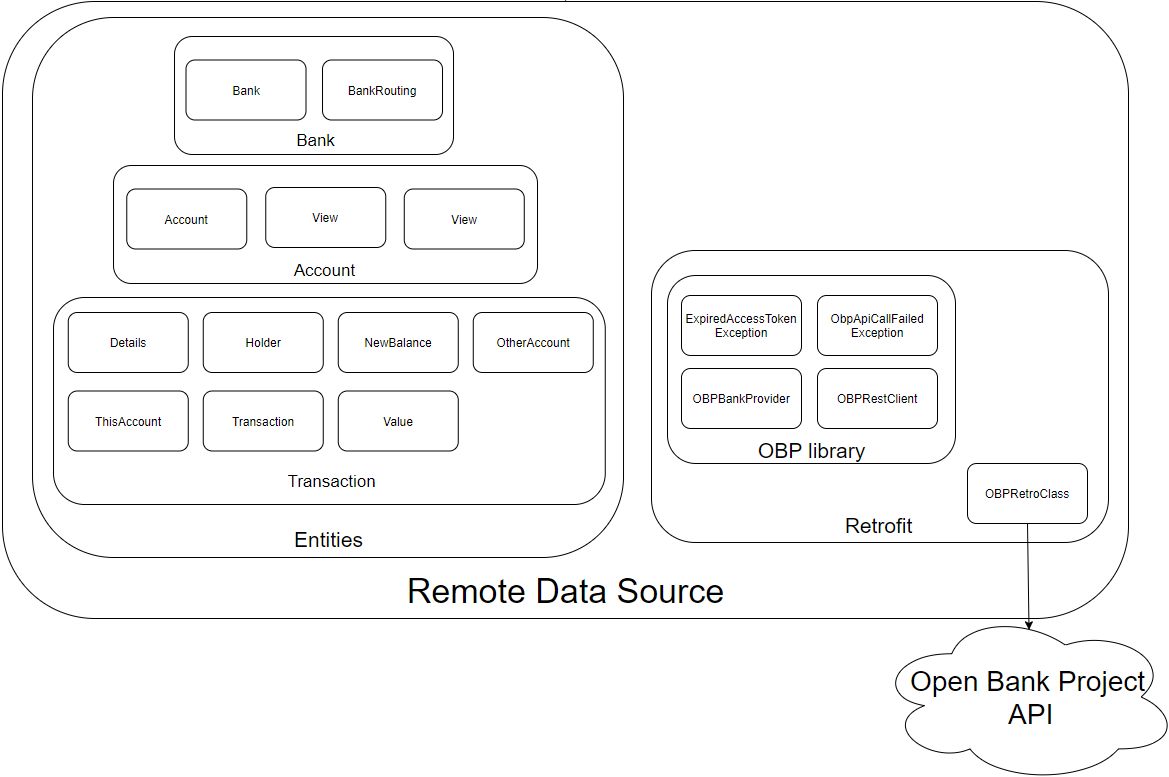


Figure ‑ Remote Data Source (Created with: [56])

### Google Sign-in flow



Figure ‑ Google Sign-in Flow (Created with: [56])

In Figure 4‑9 we have the google sign in flow with the back-end server authorization. The flow is triggered by the User when he presses the crossed cloud button from the main screen. Google sign-in process starts, an overlay screen is showed to the User where he can proceed with an already connected Google account, or he can connect with a one. After the User has selected a Google account, the Android application starts to prepare the request.

As all Google APIs are using as an authorization protocol OAuth 2.0, the authorization request needs to contain the client ID. The client ID is provided by Google at the registration of the application to a Google’s API, on Google APIs Console.

After the authorization request is built, the application sends it to Google Sign-In API Authorization Server asynchronously. A loader is started to let the User know the authorization is in progress.

A listener is registered in order to intercept the response. The response contains data about the User’s Google account and a Token ID which will be sent to our back-end server.

After we receive the response from Google, we send the Token ID to our back-end server, in order to receive a Session ID. The authorization request sent to our back-end server is also asynchronous because we do not want to block the UI.

The back-end server exposes a REST endpoint which validates the Token ID, generates a Session ID and sends it to the Android application as a response.

When the back-end server receives the request, it creates a new one which contains the received Token ID and sends it to Google Sign-In API Authorization Server synchronously. This is done in order to verify its validity and if it is indeed generated by Google. After the back-end server receives the response from Google, we search the google account in our MSSQL database to see if he is already registered. If not, we save him to the database in order to be able to offer cloud services.

After the database step, we generate a Session ID, and respond with it back to the Android application. The session ID will be intercepted by the listener we created, and it will be used to make other requests to our back-end server. Also, when the response is received, the loader will stop, the UI will be updated, and a message with the status of the login will be displayed on the screen.

### Link bank account flow

I want to mention first that this flow has been successfully implemented for Open Bank Project Sandbox API, which contains test users from test banks with test data. In order to be able to access real banks, users or data, a company with several certificates is needed. The following explained flow is developed on a sandbox environment and some things may differ on a real one. The main focus is on the concept that using a middleware platform between banks and fintechs, reduces significantly their efforts to connect to each other, and provide users new possibilities in the financial domain.

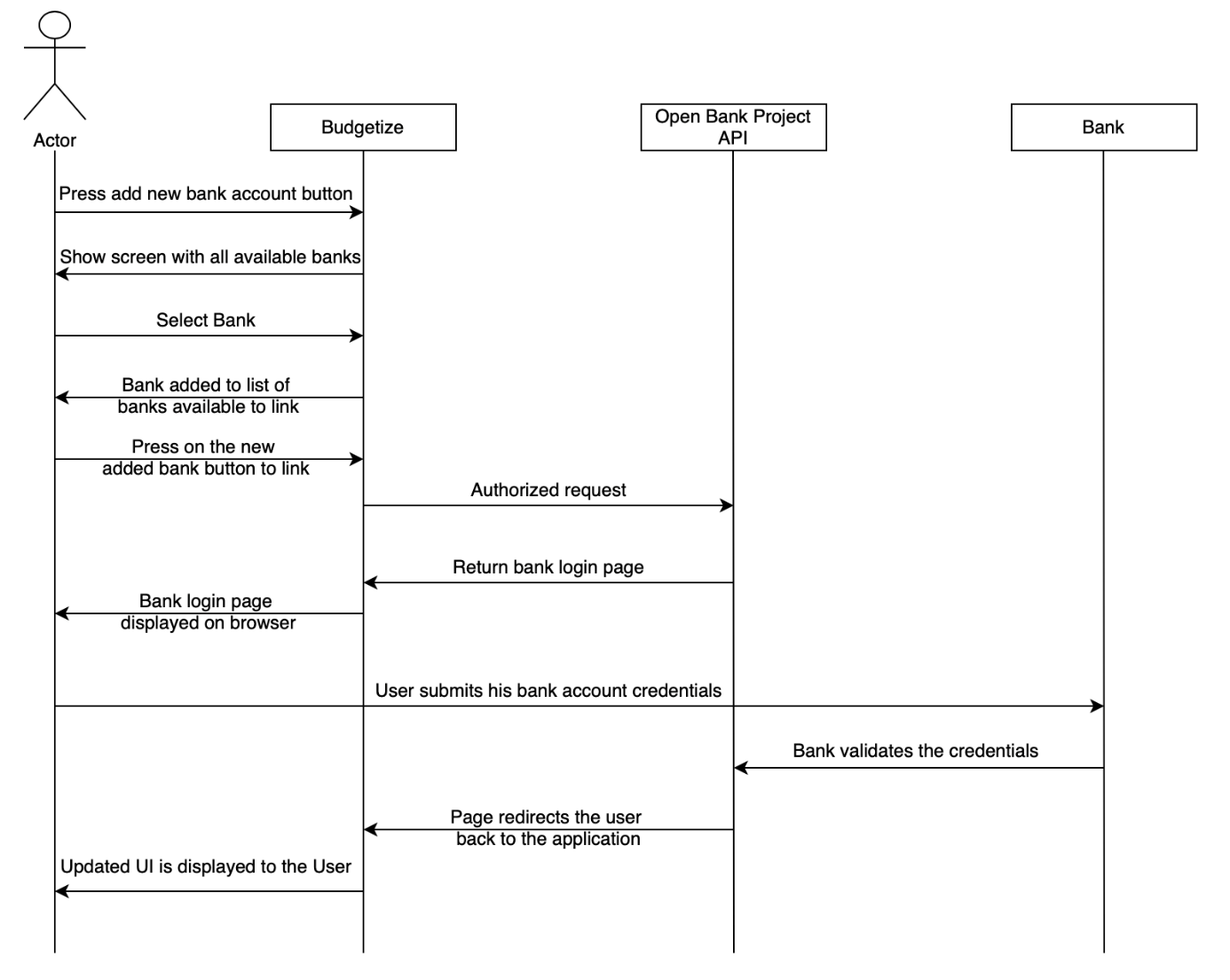


Figure ‑ Link Bank Account Flow (Created with: [56])

The flow, from Figure 4‑10, starts when the User press “Add new bank account” button. A new screen will open showing the User all the available banks. The User selects a bank from this list, and he will be returned to the previous screen, where the selected bank will be added. The User can repeat this step to add any number of banks to this screen. Every bank from this screen indicates the link status to the User. Also, it contains a button which on press it can link or unlink a bank account from the respective bank.

When the User clicks the button to link a bank, in background, the application prepares the HTTP request. For this, it needs to create an OAuth authorized URL which we will open using a new **Intent**. The URL will also contain a redirect URL, which the bank, after a successful User login, will know to redirect the User back to the application in the same screen the User was, with an updated UI.

The User will provide his bank account credentials directly to the bank, giving his consent that *Budgetize* will be able to access his bank account data on his behalf. The bank will validate User’s credentials and will provide a response. The response will redirect the User back to the Android application, where the User will see the UI updated accordingly. Also, the response will contain an access token, with which we can make further requests. With that access token we can obtain the User bank accounts, and the transactions from those bank accounts. We save into the local Room Database the added banks, together with the link status. The access token for each bank account is saved into the shared preferences, in pair with added bank id.

Now, after the User successfully linked his bank account to the application, he is able to import transactions into his wallets.

### Local Room Database Design

As I already mentioned in Chapter 3.2, the local Room Database is an abstraction layer between the Java and an SQLite Database. In the local Room Database, we have seven tables which can be seen in Figure 4‑11. Every table has a unique identifier which is the primary key of the table. I have also created foreign keys and indexes between the following tables, in order to be able to add the Cascade Delete constraints. Having those constraints, if a wallet is deleted for example, the corresponded category will be deleted, and this will trigger the deletion of the corresponding bank account, transactions, and so on. As a summary, this approach exempts us from deleting the related objects manually from the Java code.

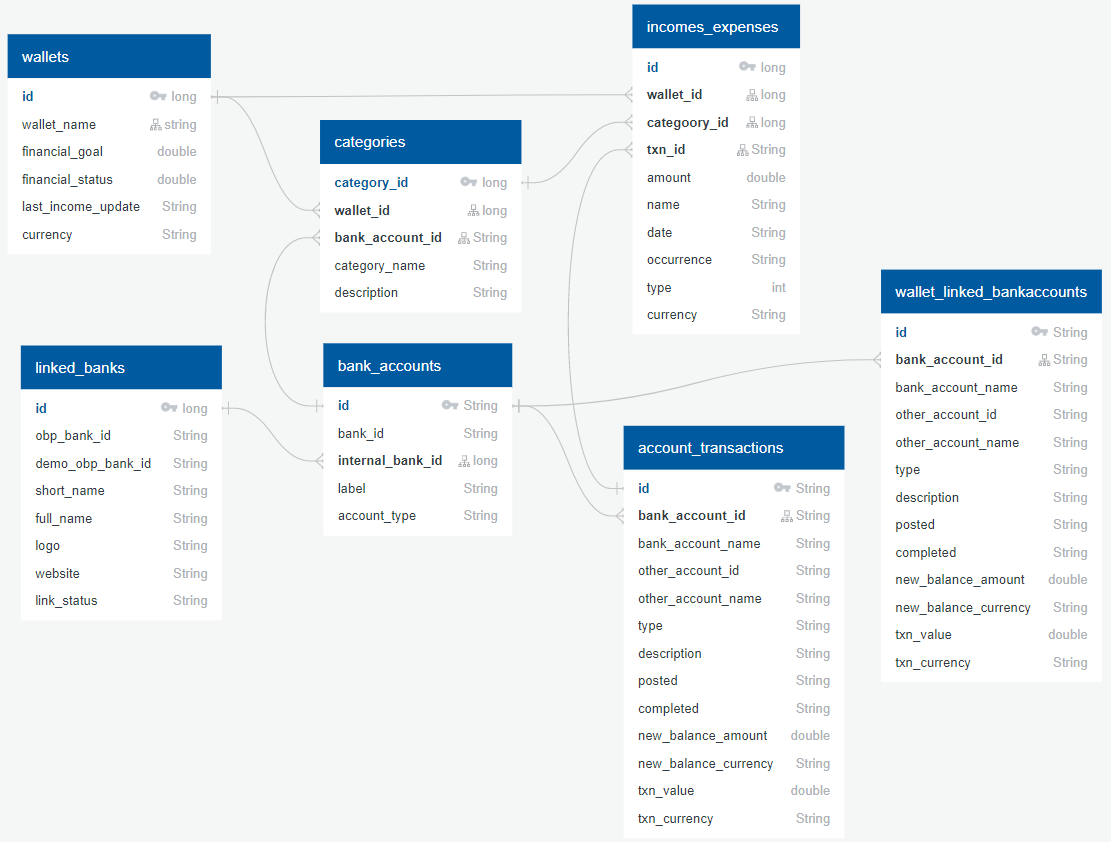


Figure ‑ Local Room Database Schema Diagram (Created with: [57])

In Code 4‑1, we can see the Java code from **WalletLinkedBankAccounts** class which contains annotations in order to map the Java object to the SQLite table.

@Entity(tableName = **"wallet\_linked\_bankaccounts"**,foreignKeys = {  
 @ForeignKey(onDelete = ***CASCADE***, entity = Wallet.**class**,  
 parentColumns = **"id"**, childColumns = **"wallet\_id"**),  
 @ForeignKey(onDelete = ***CASCADE***, entity = BankAccount.**class**,  
 parentColumns = **"id"**, childColumns = **"bank\_account\_id"**),  
 @ForeignKey(onDelete = ***CASCADE***, entity = LinkedBank.**class** , parentColumns = **"id"**, childColumns = **"linked\_bank\_id"**)}  
 , indices = {  
 @Index(**"wallet\_id"**),  
 @Index(**"linked\_bank\_id"**),  
 @Index(**"bank\_account\_id"**)})

**public class** WalletLinkedBankAccounts {  
 @PrimaryKey  
 @NonNull  
 @ColumnInfo(name = **"id"**)  
 **private long id**;  
  
 @ColumnInfo(name = **"wallet\_id"**)  
 **private long wallet\_id**;  
  
 @ColumnInfo(name = **"linked\_bank\_id"**)  
 **private long linked\_bank\_id**;  
  
 @ColumnInfo(name = **"bank\_account\_id"**)  
 **private** String **bank\_account\_id**;

Code ‑ Example of Room table definition.

### Remote Database Design

The remote database from the back-end server is a Microsoft SQL Server. The structure of the table is very similar with the local SQLite database (Figure 4‑8). In this database, I have created a new table called “users” with the primary key “userID”. We are storing all the users in order to be able to identify, save and retrieve their application data. All other tables are identical to the tables from the local Room Database, excepting that I add to each of them a new field called “userID” in order to identify to which user the data belongs to. Also, we have added two new foreign keys, one between wallets and users and another one between linked\_banks and users.

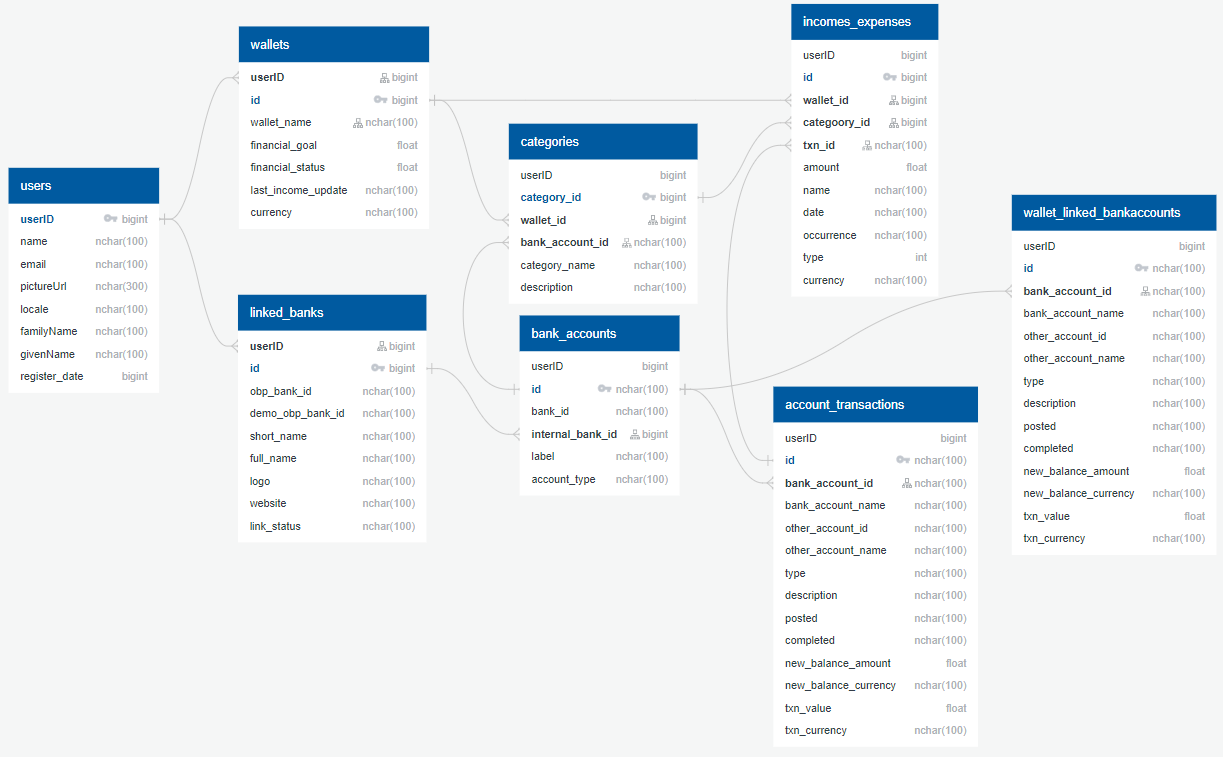


Figure ‑ Remote MSSQL Database Schema Diagram(create with: [57])

## Implementation

The Android application has been developed using Android Studio. The back-end server has been developed using the Spring Boot framework. Both back-end server and Android application are written in the Java programming language. The communication between the client and server is realized using RESTful services.

Both client and server projects have around 10.000 lines of Java code, therefore, in this chapter, I am going to exemplify portions of code only from the main functionalities.

I will start explaining what happens in the background when the application is started. In the **MainActivity** class in **onCreate** method (Code 4‑2), I am initializing the **ViewModels**, ask the user for permissions, start the biometric authentication, do the logic for the first start of the application and initialize the Google Sign-in login, listeners, intro screen and currencies.

@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 *//Initialize the ViewModels* initViewModels();  
 *// Request permissions* askForPermissions();  
 *// Start the biometric authentication and restrict the access to the application* **try** {  
 appAuthentication();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 Toast.*makeText*(**this**, e.getMessage(), Toast.***LENGTH\_LONG***);  
 }  
 *// Download get all new transactions from the linked bank accounts* doLogicOnFirstAppStart();  
 *// Check if the user is already logged in with the Google Account  
 // if yes, try to get a session ID from the back-end server  
 // in order to be able to use the Special Features* initLogin();  
 *// Initialize all buttons listeners* initListeners();  
 *// Initialize elements for empty screen case* initIntroScreen();  
 *// Retrieve from exchangeratesAPI all currency rates from today* MainActivityViewModel.*prepareCurrencies*();  
}

Code ‑ **onCreate** method

The **initViewModels** (Code 4‑3) method is initializing the three **ViewModels** which I am using in this **Activity** and to get the list of available banks.

**private void** initViewModels() {  
 **mainActivityViewModel** = **new** ViewModelProvider(**this** , **new** MainActivityViewModelFactory((ApplicationObj) **this**.getApplication()))  
 .get(MainActivityViewModel.**class**);  
  
 **bankAccountViewModel** = **new** ViewModelProvider(**this**,  
 **new** BankAccountViewModelFactory((ApplicationObj) **this**.getApplication()))  
 .get(BankAccountViewModel.**class**);  
  
 **servicesHandlerViewModel** = **new** ViewModelProvider(**this**,  
 **new** ServicesHandlerViewModelFactory((ApplicationObj) **this**.getApplication()))  
 .get(ServicesHandlerViewModel.**class**);  
 *// Get a list of all available banks* **servicesHandlerViewModel**.getAllAvailableBanks();  
}

Code ‑ **initViewModels** method

**getAllAvailableBanks** method from Code 4‑4, access the repository in order to get all the available banks from Open Bank Project API.

**public void** getAllAvailableBanks() {  
 **repository**.getAllAvailableBanks(***mObservableAvailableBanks***);  
}

Code ‑ **getAllAvailableBanks** method

We send as a parameter a **MutableLiveData** which can observe every change on the list of banks. We are sending it in order to be able to post the new list of banks (Code 4‑5).

**public static final** MutableLiveData<List<Bank>> ***mObservableAvailableBanks*** = **new** MutableLiveData<>();

Code ‑ **mObservableAvailableBanks** variable

The post will be observed in the **ViewModel** and the Code 4‑6 will be performed. The list of available banks will be updated and if the application just started the bank logos will be updated as well.

***mObservableAvailableBanks***.observeForever(banks -> {  
 *// Clean bank list  
 banksList*.clear();  
 *// Add all banks  
 banksList*.addAll(banks);  
 *// Perform this action only at the start of the application* **if** (!*onFirstCreation*) {  
 *// Download all bank logos* syncAllImages();  
 *onFirstCreation* = **true**;  
 }  
});

Code ‑ Observer’s **onChanged** method from of **mObservableAvailableBanks** variable

**syncAllImages** method (Code 4‑7) will begin to download all new bank logos on a new **Thread**.

**public void** syncAllImages() {  
 Thread syncImagesThread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **bankImagesDownloader**.doSync(*banksList*);  
 }  
 });  
 syncImagesThread.start();  
}

Code ‑ **synchAllImages** method

To perform that action, we will use the **BankImageDownloader** class. **doSync** method will check which bank logo is not available on the device, and it will download and save them on the external storage device if available, or on the internal device. We are performing the download on 10 threads simultaneously. Those threads are managed by a **ThreadPoolExecutor** (Code 4‑8).

**public class** BankImagesDownloader {

**private** ThreadPoolExecutor **executor** =   
 (ThreadPoolExecutor) Executors.*newFixedThreadPool*(10);  
**private** File **fileLocation**;  
  
**public** BankImagesDownloader() {  
 **if** (ImageDownloader.*isExternalStorageWritable*()) {  
 **fileLocation** = **new** File(Environment.*getExternalStorageDirectory*() +  
 **"/Android/data/"** + ApplicationObj.*getAppContext*().getPackageName()  
 + **"/BankIcons"**);  
 } **else** {  
 **fileLocation** = **new** File(ApplicationObj.*getAppContext*().getFilesDir()  
 + **"/BankIcons"**);  
 }  
}

Code ‑ **BasicImageDownloader** class constructor and members

The download and save status of each bank’s logo is treated accordingly using listeners (Code 4‑9).

**public void** doSync(List<Bank> banks) {  
 *// Get the location where to save the logos* File folder = **new** File(**fileLocation**.getPath());  
 **boolean** success = **true**;  
 *// If the folder does not exist, create it* **if** (!folder.exists()) {  
 success = folder.mkdirs();  
 }  
 *// Log for Debug purposes* Log.*d*(**"Dir created successfully? "**, success + **"!"**);  
 **if** (success) {  
 **for** (**int** i = 0; i < banks.size(); i++) {  
 File file = **new** File(**fileLocation**.getPath()  
 + File.***separator*** + banks.get(i).getId() + **".png"**);  
 **int** finalI = i;  
 **final** ImageDownloader downloader =  
 **new** ImageDownloader(**new** ImageDownloader.OnImageLoaderListener() {  
 *// Here we handle the different statuses of the bank logo download* @Override  
 **public void** onError(ImageDownloader.ImageError error) {  
 Log.*d*(**"Error code "**, error.getErrorCode() + **": "** + error.getMessage());  
 error.printStackTrace();  
 }  
  
 @Override  
 **public void** onProgressChange(**int** percent) {  
 }  
  
 @Override  
 **public void** onComplete(Bitmap result) {  
 *// Set the format to save the file* **final** Bitmap.CompressFormat mFormat = Bitmap.CompressFormat.***PNG***;  
 *// Create the file on the disk* **final** File imageFileLocation = **new** File(**fileLocation**.getPath()  
 + File.***separator*** + banks.get(finalI).getId() + **".png"**);  
 *// Save the downloaded bitmap to the imageFileLocation* ImageDownloader.*writeToDisk*(imageFileLocation,  
 result,  
 **new** ImageDownloader.OnBitmapSaveListener() {  
 @Override  
 **public void** onBitmapSaved() {  
 *// Debug purposes* Log.*d*(**"SAVED"**, **"IMAGE SAVED TO LOCATION:"** + imageFileLocation.getAbsolutePath());  
 }  
  
 @Override  
 **public void** onBitmapSaveError(ImageDownloader.ImageError error) {  
 *// Debug purposes* Log.*d*(**"Error code "**, error.getErrorCode() + **": "** +  
 error.getMessage());  
 error.printStackTrace();  
 }  
 }, mFormat, **false**);  
 }  
 });  
 *// Check if the file if Bank's Logo is already downloaded* **if** (ImageDownloader.*readFromDisk*(file) == **null**) {  
 String logoUrl = banks.get(i).getLogo();  
 *// Check if Bank's Logo URL is valid* **if** (logoUrl != **null** && !logoUrl.contentEquals(**""**)) {  
 *// Start the DownloadTask on a new Thread* **executor**.execute(**new** DownloadTask(downloader, logoUrl, **true**));  
} } } } }

Code ‑ **doSync** method

All the flow from above is performed at the start of the application, in order prepare the data for all the application screens, creating a good user experience.

Returning to the **onCreate** method from **MainActivity** class, the next called method is **askForPermissions** (Code 4‑10). There we have the code which asks the user to allow the application to access his photos, media and files. The request message is displayed on the Figure 5‑1.

**private void** askForPermissions() {  
 *// We check to see if we have permission to write on external storage* **if** (ContextCompat.*checkSelfPermission*(**this**,  
 Manifest.permission.***WRITE\_EXTERNAL\_STORAGE***)  
 != PackageManager.***PERMISSION\_GRANTED***) {  
 *// If we don't have the permission, we need to ask the user for it  
 // This code will pop-up a request message to the user's screen* ActivityCompat.*requestPermissions*(**this**,  
 **new** String[]{Manifest.permission.***WRITE\_EXTERNAL\_STORAGE***},  
 **WRITE\_EXTERNAL\_STORAGE\_PERMISSION**);  
 } **else** {  
 **servicesHandlerViewModel**.syncAllImages();  
 }  
}

Code ‑ **askForPermissions** method

Also, I had override the **onRequestPermissionResult**  method (Code 4‑11) from **AppCompatActivity** in order to catch the response of the user from Figure 5‑1. If the user denies the access, I am showing a new **AlertDialog** where I am explaining that without that permission, he will not be able to use the “Import transactions from bank accounts” functionality (Code 4‑12).

@Override  
**public void** onRequestPermissionsResult(**int** requestCode,  
 @NotNull String[] permissions,  
 @NotNull **int**[] grantResults) {  
 **if** (requestCode == **WRITE\_EXTERNAL\_STORAGE\_PERMISSION**) {  
 *// Parse all the request permissions results* **for** (**int** i = 0, len = permissions.**length**; i < len; i++) {  
 String permission = permissions[i];  
 *// User denied the access permission* **if** (grantResults[i] == PackageManager.***PERMISSION\_DENIED***) {  
 **boolean** showRationale = shouldShowRequestPermissionRationale(permission);  
 **if** (!showRationale) {  
 } **else if** (Manifest.permission.***WRITE\_EXTERNAL\_STORAGE***.equals(permission))

{  
 *// Here we show a new message to the user  
 // explaining why it is important to allow WRITE\_EXTERNAL\_STORAGE*

*// permission* showRationaleExtDev();  
 }  
 } **else** {  
 **servicesHandlerViewModel**.syncAllImages();  
 }  
 }  
 }  
}

Code ‑ **onRequestPermissionsResult** method

**private void** showRationaleExtDev() {  
 *// Create new onClickListener for Yes and No buttons* DialogInterface.OnClickListener dialogClickListener =   
 **new** DialogInterface.OnClickListener() {  
 @Override  
 **public void** onClick(DialogInterface dialog, **int** which) {  
 **switch** (which) {  
 **case** DialogInterface.***BUTTON\_POSITIVE***: {  
 *// Yes button clicked  
 // Do nothing* **break**;  
 }  
 **case** DialogInterface.***BUTTON\_NEGATIVE***: {  
 *// No button clicked  
 // We display the ask permission again to the user* askForPermissions();  
 **break**;  
 }  
 }  
 }  
 };  
 *// Build the dialog* AlertDialog.Builder builder = **new** AlertDialog.Builder(**this**);  
 *// Set the message, the onClick listeners and show the dialog* builder.setMessage(getString(R.string.***permission\_denied\_write\_external\_storage***))  
 .setPositiveButton(**"Yes"**, dialogClickListener)  
 .setNegativeButton(**"No"**, dialogClickListener).show();  
}

Code ‑ **showRationaleExtDev** method

After the **askForPermissions** method, I written a try-catch block, where I am calling the **appAuthentication** method (Code 4‑13), and in case that this methods generates an exception, we print its stack trace to the logs and write the error message on the screen in order to let the user know why he is not able to use the application.

In the **appAuthentication** method, we firstly check if the device supports a biometric login, or if there is any biometric login configured to the device.

*//Biometric app Authentication***private void** appAuthentication() **throws** Exception {  
 BiometricManager biometricManager = BiometricManager.*from*(**this**);  
 *// Check if the device contains any biometric hardware available  
 // or if there is any biometric login registered to the device* **switch** (biometricManager.canAuthenticate()) {  
 **case** BiometricManager.***BIOMETRIC\_SUCCESS***:  
 Log.*d*(**"MY\_APP\_TAG"**, **"App can authenticate using biometrics."**);  
 biometricLogin();  
 **break**;  
 **case** BiometricManager.***BIOMETRIC\_ERROR\_NO\_HARDWARE***:  
 Log.*e*(**"MY\_APP\_TAG"**, **"No biometric features available on this device."**);  
 **throw new** Exception(**"No biometrics available"**);  
 **case** BiometricManager.***BIOMETRIC\_ERROR\_HW\_UNAVAILABLE***:  
 Log.*e*(**"MY\_APP\_TAG"**, **"Biometric features are currently unavailable."**);  
 **throw new** Exception(**"No biometrics available"**);  
 **case** BiometricManager.***BIOMETRIC\_ERROR\_NONE\_ENROLLED***:  
 Log.*e*(**"MY\_APP\_TAG"**, **"The user hasn't associated "** +  
 **"any biometric credentials with their account."**);  
 **throw new** Exception(**"No biometrics available"**);  
 }  
}

Code ‑ **appAuthentication** method

If the user can be authenticated to the application with his configured biometrics, the authentication process starts by calling the **biometricLogin** method.

In the **biometricLogin** method (Code 4‑14), we are preparing the dialog which inform the user on how to perform the authentication and we show it. We had also added a callback to this dialog, in order to handle the status of the authentication. If the authentication failed, we just inform the user by “Toasting” a message on the screen. If the user authenticated successfully, the login screen will disappear, and the application will show up almost instantaneously as some data has been prepared in advance.

**private void** biometricLogin() {  
 *// There we prepare the biometric dialog  
 // informing the user about the steps which he have to perform* **executor** = ContextCompat.*getMainExecutor*(**this**);  
 *// We register a callback to make the application react  
 // to a successful or unsuccessful validation* **biometricPrompt** = **new** BiometricPrompt(MainActivity.**this**,  
 **executor**, **new** BiometricPrompt.AuthenticationCallback() {  
 @Override  
 **public void** onAuthenticationError(**int** errorCode,  
 @NonNull CharSequence errString) {  
 **super**.onAuthenticationError(errorCode, errString);  
 *// We close the dialog and we inform the User that  
 // the authentication has failed* Toast.*makeText*(getApplicationContext(),  
 **"Authentication error: "** + errString, Toast.***LENGTH\_SHORT***)  
 .show();  
 **biometricPrompt**.cancelAuthentication();  
 }  
  
 @Override  
 **public void** onAuthenticationSucceeded(  
 @NonNull BiometricPrompt.AuthenticationResult result) {  
 **super**.onAuthenticationSucceeded(result);  
 Toast.*makeText*(getApplicationContext(),  
 **"Authentication succeeded!"**, Toast.***LENGTH\_SHORT***).show();  
 authUpdateUI(**true**);  
 animate\_firstPagePb();  
 MainActivityViewModel.*firstStart*=**false**;  
 }  
  
 @Override  
 **public void** onAuthenticationFailed() {  
 **super**.onAuthenticationFailed();  
 Toast.*makeText*(getApplicationContext(), **"Authentication failed"**,  
 Toast.***LENGTH\_SHORT***)  
 .show();  
 **biometricPrompt**.cancelAuthentication();  
 }  
 });  
 **biometricPrompt**.authenticate(**promptInfo**);  
}

Code ‑ **biometricLogin** method

The next called method is **doLogicOnFirstAppStart** (Code 4‑15).

**public void** doLogicOnFirstAppStart() {  
 **tabLayout** = (TabLayout) findViewById(R.id.***tabDots***);  
 **bankAccountViewModel**.appFirstStartLogicInit();  
}

Code ‑ **doLogicOnFirstAppStart** method

This method initializes the **tabLayout** (this is an UI element, used to display the page on which the user is currently on, by using dots) and calls a method from the **BankAccountViewModel** class named **appFirstStartLogicInit** (Code 4‑16). That method is responsible to initialize the **LiveData** objects and to start observing the changes from the Room Database. Also, it is getting all the new created accounts from a linked bank and refreshes the transaction list for each bank account.

**public void** appFirstStartLogicInit(){  
 *// Get all the linked banks from the local database* **linkedBanksList** = **this**.getAllLinkedBanks();  
 *// Start observing changes on linked\_banks table* **linkedBanksList**.observeForever(linkedBanks -> {  
 **for** (LinkedBank linkedBank : linkedBanks) {  
 *// For every linked bank, we retrieve the consumer and provider  
 // in order to be able to request from the linked bank  
 // new user's transactions* **if** (!OBPRestClient.*consumers*.containsKey(linkedBank.getId())) {  
 OBPRestClient.*retrieveConsumerAndProvider*(linkedBank.getId());  
 }  
 *// We retrieve from OBP API, new user's accounts  
 // from the already linked bank accounts  
 // and we insert them into the database* **this**.getAccounts(linkedBank.getId());  
 }  
 });  
 *// We retrieve the bank accounts from the database* **bankAccounts** = **this**.getAllBankAccounts();  
 *// Start observing changes on bank\_accounts table* **bankAccounts**.observeForever(accounts -> {  
 *// Refresh list of bank accounts  
 accountList*.clear();  
 *accountList*.addAll(accounts);  
 *// Get all the new transactions for each bank account  
 // and we insert them into the database* **for** (BankAccount bankAccount : *accountList*) {  
 getTransactions(  
 bankAccount.getInternal\_bank\_id(),  
 bankAccount.getBank\_id(),  
 bankAccount.getId());  
 }  
 });  
}

Code ‑ **appFirstLogicInit** method

The next important method in the start-up flow, is **initLogin** (Code 4‑17). This is used to check if the user has already signed-in with a Google account. If there is an account found, we are validating it on our back-end server, in order to obtain a Session ID. With this Session ID, the user will be able to access the Special Functionalities.

**private void** initLogin() {  
 *//Build the GoogleSignInOptions* GoogleSignInOptions gso = **new** GoogleSignInOptions.Builder(GoogleSignInOptions.***DEFAULT\_SIGN\_IN***)  
 .requestIdToken(getString(R.string.***server\_client\_id***))  
 .requestEmail()  
 .build();  
 **mGoogleSignInClient** = GoogleSignIn.*getClient*(**this**, gso);  
 *// We check if there if the User already Signed-in with Google* **if** ((**account** = GoogleSignIn.*getLastSignedInAccount*(**this**)) != **null**) {  
 **googleSignInTokenID** = **account**.getIdToken();  
 *// We found google account linked to the application  
 // Validate on server to obtain session ID.  
 // (In order to perform validated requests to Back-end server)* Thread validateUserThread = **new** Thread(**this**::serverValidateUser);  
 validateUserThread.start();  
 }  
}

Code ‑ **initLogin** method

In the previous code, we are able to see that we are starting the **serverValidateUser** method (Code 4‑18) on a new thread, in order to avoid blocking the UI thread. **serverValidateUser** method, activates a loader until the response is received from the server, starts the **ValidateUserTask** and define the **onTaskCompleted** callback. In the **onTaskCompleted** method, we are handling different types of responses and we update the UI accordingly.

*//Communication with my RESTful API***public void** serverValidateUser() {  
 *// We activate a progress bar loader, in order to let the user know  
 // that the sign-in request is in process* MainActivity.**this**.runOnUiThread(**this**::activateLoader);  
 *// We create a new ValidateUserTask and define the onTaskCompleted listener* **new** ValidateUserTask(**new** OnTaskCompleted() {  
 @Override  
 **public void** onTaskCompleted(String responseBody) {  
 *// When the task has been completed, firstly we deactivate the progress bar loader* MainActivity.**this**.runOnUiThread(() -> deactivateLoader());  
 *// Then we check the responseBody received from the server  
 // in order to see if there is any exception, or a valid Session ID* **if** (responseBody.toUpperCase().contains(**"EXCEPTION"**)) {  
 **if** (responseBody.toUpperCase().contains(**"CONNECTEXCEPTION"**)) {  
 MainActivity.**this**.runOnUiThread(() -> {  
 *// Inform the user there is no internet connection* Toast.*makeText*(MainActivity.**this**,  
 **"NO INTERNET CONNECTION"**, Toast.***LENGTH\_SHORT***).show();  
 });  
 } **else if** (responseBody.toUpperCase().contains(**"SOCKETTIMEOUTEXCEPTION"**)) {  
 MainActivity.**this**.runOnUiThread(() -> {  
 *// Inform the user that Budgetize's server is not available* Toast.*makeText*(MainActivity.**this**,  
 **"SERVICE CURRENTLY UNAVAILABLE"**, Toast.***LENGTH\_SHORT***).show();  
 });  
 }  
 *// If any issue occurs, we logout the user also from the google sign in account  
 // disabling his access to the special features also* signOut();  
 } **else** {  
 *// If there is no exception in the responseBody  
 // we retrieve the sessionID and we update the UI accordingly.* **sessionID** = responseBody;  
 MainActivity.**this**.runOnUiThread(() -> {  
 updateUI(**account**);  
 });  
 }  
 **endTimeGoogleSignInFlow** = System.*currentTimeMillis*();  
 Log.*d*(**" Time took Sign-in Flow"**,  
 **" "** + (**endTimeGoogleSignInFlow** - **startTimeGoogleSignInFlow**));  
 }  
 }, **googleSignInTokenID**).doInBackground();  
}

Code ‑ **serverValidateUser** method

We use the interface from bellow as a callback mechanism. **ValidateUserTask** receives as parameters the callback and the Google Sign-in Token ID (Code 4‑19). Using the OkHttp library, we build and send a new request. The request contains the server’s URL + endpoint and the body. In body we add the Token ID received from google. In the response we expect to receive the Session ID, or an exception. The response will be treated in the in the previous code.

**public interface** OnTaskCompleted {  
 **void** onTaskCompleted(String responseBody);  
}  
  
**private class** ValidateUserTask {  
 String **googleSignInTokenID**;  
 OnTaskCompleted **listener**;  
  
 ValidateUserTask(OnTaskCompleted listener, String googleSignInTokenID) {  
 **this**.**listener** = listener;  
 **this**.**googleSignInTokenID** = googleSignInTokenID;  
 }  
  
 **protected** String doInBackground(String... urls) {  
 **try** {  
 *// We use OkHttp library to build the request* OkHttpClient client = **new** OkHttpClient();  
 *// Create the body of the request, which contains the Google tokenID* RequestBody formBody = **new** FormBody.Builder()  
 .add(**"tokenID"**, **googleSignInTokenID**)  
 .build();  
 *// We build the request by setting the url+endpoint and the CRUD operation* Request request = **new** Request.Builder()  
 .url(**serverBaseLink** + **"/validateUser"**)  
 .post(formBody)  
 .build();  
 *// We call and wait for a response* Call call = client.newCall(request);  
 Response response = call.execute();  
 *// We forward the response* onPostExecute(Objects.*requireNonNull*(response.body()).string());  
 } **catch** (Exception e) {  
 *// We catch different type of exceptions and forward them* onPostExecute(e.toString());  
 Log.*e*(**"TokenIDtoServer"**, **"Error sending ID token to back-end."**, e);  
 }  
 **return null**;  
 }  
  
 **void** onPostExecute(String response) {  
 *// We use the callback* **listener**.onTaskCompleted(response);  
 }  
}

Code ‑ **onTaskCompleted** callback and **ValidateUserTask** class

In Code 4‑20 we have the **initIntroScreen** method, where we prepare an empty screen, in the case the user does not have any Wallet created.

**private void** initIntroScreen() {  
 **introText** = (TextView) **this**.findViewById(R.id.***intro\_text***);  
 **introArrow** = (ImageView) **this**.findViewById(R.id.***intro\_arrow\_imgview***);  
}

Code ‑ **initIntroScreen** method

The last method from **onCreate** is **initCurrencies** (Code 4‑21). On a separate **Thread** we retrieve all the currency rates in Euro, with which we will apply the currency filter to the wallets.

**public static void** initCurrencies() {  
 Thread t = **new** Thread() {  
 @Override  
 **public void** run() {  
 **try** {  
 String response = HttpUtils  
 .*get*(**"https://api.exchangeratesapi.io/latest?base="** + **"EUR"**);  
 Gson gson = **new** Gson();  
 *currency* = gson.fromJson(response, Currency.**class**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 };  
 t.start();  
 **try** {  
 t.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

Code ‑ **initCurrencies** method

After the **onCreate**, **onResume** method (Code 4‑22) will be called. In this method, we initialize the **ViewPager**, the wallet list **Observer** and we register the **Observer** to detect changes on the wallets table from the Room Database.

@Override  
**public void** onResume() {  
 **super**.onResume();  
 *// Initialize the ViewPager* initViewPager();  
 *// Initialize the wallet list observer  
 // At every Wallet change in the database,  
 // we refresh the wallet list and the viewpager* **walletListObsever** = wallets -> {  
 MainActivity.*wallets*.clear();  
 MainActivity.*wallets*.addAll(wallets);  
 **adapter** = **new** WalletViewPagerAdapter(wallets, MainActivity.**this**, **mainActivityViewModel**);  
 **adapter**.notifyDataSetChanged();  
 **viewPager**.setAdapter(**adapter**);  
 **viewPager**.setCurrentItem(**mainActivityViewModel**.**lastWalletPosition**);  
 **mainActivityViewModel**.**lastWalletPosition** = 0;  
 **tabLayout**.setupWithViewPager(**viewPager**, **true**);  
 MainActivity.**this**.refresh\_tabs();  
 *// Activate/Deactivate intro screen  
 // based on the number of wallets* **if** (wallets.size() == 0) {  
 activateDeactivateIntroScreen(**true**);  
 } **else** {  
 activateDeactivateIntroScreen(**false**);  
 refresh\_wallets();  
 *// Animate the first page's progress bar* **if**(!MainActivityViewModel.*firstStart*) {  
 animate\_firstPagePb();  
 }  
 }  
 *// animate the progress bar from the first page  
 //animate\_firstPagePb();* };  
 *// we register the wallet list observer* **mainActivityViewModel**.getAllWallets().observe(**this**, **walletListObsever**);

}

Code ‑ **onResume** method

Another important functionality of the application I want to exemplify is the way we retrieve and save new transactions of an account. This happens every time the application is restarted or more explicity when the bank accounts list is retrieved or changed. This starts from **appFirstStartLogicInit** method (Code 4‑16) and continues by calling **getTransactions** method (Code 4‑23). This method takes as parameter the **bankID**, which is our internal ID used to identify the correspondent Token ID. Token ID is used in order to have access to the Open Bank Project API. Then we have **obpBankID** and **accountID** to let Open Bank Project API know from which bank and which account we try to get the transactions. Also, we send the reference to the repository, as the transactions will need to be add into the local database.

*//Operations with bank transactions:***public void** getTransactions(**long** bankID, String obpBankID, String accountID){  
 **repository**.getTransactions(bankID, obpBankID, accountID, **repository**);  
}

Code ‑ **getTransactions** method from the model view

From the **DataRepository** class, we call **getAllTransactions** method (Code 4‑24) from the **OBPRetroClass** with the same parameters.

**public void** getTransactions(**long** bankID, String obpBankID, String accountID,  
 DataRepository dataRepository) {  
 OBPRetroClass.*getAllTransactions*(bankID, obpBankID, accountID, dataRepository);  
}

Code ‑ **getTransactions** method from the Repository class

In **getAllTransaction** method (Code 4‑25) from **OBPRetroClass** we have an **AsyncTask** which starts by retrieving a JSON of all transactions from Open Bank Project API.

**public static void** getAllTransactions(**long** bankID, String obpBankID, String accountID, DataRepository repository) {  
 **new** AsyncTask<Void, Void, Void>() {  
 @Override  
 **protected** Void doInBackground(Void... voids) {  
 **try** {  
 *// Declare the objects we will use  
 // Transaction is the object defined by OBP* ArrayList<Transaction> transactionsArrayList = **new** ArrayList<>();  
 *// AccountTransaction is our transaction object we use internally* List<AccountTransaction> accountTransactions = **new** ArrayList<>();  
 Gson gson = **new** Gson();  
 **long**[] status;  
 StringBuilder thisHolders;  
 *//Retrieve transactions from the Open Bank Project API  
 // We receive a JSONOBject which packs all the transactions* JSONObject transactionsJson = OBPRestClient.*getTransactions*(bankID, obpBankID, accountID);

Code ‑ **getAllTransactions** method

We get that JSON by making an authorized request to the endpoint from Code 4‑26.

**public static** JSONObject getTransactions(**long** bankID, String obpBankID, String accountID)  
 **throws** ExpiredAccessTokenException, ObpApiCallFailedException {  
 **return** *getOAuthedJson*(OBPBankProvider.***BASE\_URL\_OBP\_SANDBOX*** + **"/obp/v4.0.0/my/banks/"** + obpBankID + **"/accounts/"** + accountID + **"/transactions"**, bankID);  
}

Code ‑ **getTransactions** method

When the **JSONObject** is received, we unpack it to obtain an **JSONArray** of transactions (Code 4‑27). This array is parsed and using **GSON** we map each JSON transaction to the **Transaction** Java object and add it to an **ArrayList**.

*// We need to parse the array and get the data we need from each transaction*JSONArray transactionsJsonJSONArray = transactionsJson.getJSONArray(**"transactions"**);  
**for** (**int** i = 0; i < transactionsJsonJSONArray.length(); i++) {  
 *// We convert the JSON transaction to the Transaction Object and add it  
 // to transactionArrayList* transactionsArrayList.add  
 (gson.fromJson(transactionsJsonJSONArray.getJSONObject(i).toString(),  
 Transaction.**class**));  
}

Code ‑ unpack JSON

Then we need to convert these transaction objects to our internal objects. In order to do that, we first check the **transactionsArrayList** size to make sure we retrieved any transaction from OBP. Then we parse the array and we extract only the needed data from the **Transaction** object. We create **AccountTransaction** objects adding that data and we add all these objects to an a new **ArrayList**. This **ArrayList** will be inserted into our local Room Database (Code 4‑28).

*// We check if any transaction has been retrieved from OBP API***if** (transactionsArrayList.size() != 0) {  
 *// We transform the each Transaction object to our AccountTransaction object* **for** (Transaction transaction : transactionsArrayList) {  
 thisHolders = **new** StringBuilder();  
 **for** (Holder holder : transaction.getThis\_account().getHolders()) {  
 thisHolders.append(**","**).append(holder.getName());  
 }  
 thisHolders = **new** StringBuilder(  
 thisHolders.substring(0, thisHolders.length() - 2));  
 *// We create the AccounTransaction object and add it to the ArrayList* accountTransactions.add(  
 **new** AccountTransaction(  
 transaction.getId(),  
 transaction.getThis\_account().getId(),  
 thisHolders.toString(),  
 transaction.getOther\_account().getId(),  
 transaction.getOther\_account().getHolder().getName(),  
 transaction.getDetails().getType(),  
 transaction.getDetails().getDescription(),  
 transaction.getDetails().getPosted(),  
 transaction.getDetails().getCompleted(),  
 transaction.getDetails().getNew\_balance().getAmount(),  
 transaction.getDetails().getNew\_balance().getCurrency(),  
 Math.*abs*(transaction.getDetails().getValue().getAmount()),  
 transaction.getDetails().getValue().getCurrency()  
 )  
 );  
 }  
 *// after we converted all the objects, we insert the list of  
 // AccountTransaction objects  
 // into our local Room Database* status = repository.insertAllAccountTransactions(accountTransactions);

Code ‑ convertion to **AccountTransaction** objects and insertion into the Database

To insert that **ArrayList** into our local Room Database we need to call the **insertAllAccounTransactions** method (Code 4‑29) from the **DataRepository**. This method will call the **insertAllAccountTransactions** from the **AccountTransactionDao** interface.

@Override  
**public long**[] insertAllAccountTransactions(List<AccountTransaction> accountTransactions) {  
 **return mDatabase** .accountTransactionDao()  
 .insertAllAccountTransactions(accountTransactions);  
}

Code ‑ **insertAllAccountTransactions** method from Repository class

In **AccountTransactionDao** we have the Insert annotation with the ignore strategy on conflict (Code 4‑30). This annotation will try to insert the parameter received, in our case **accountTransactions** ignoring the duplicates.

@Insert(onConflict = OnConflictStrategy.***IGNORE***)  
**public long**[] insertAllAccountTransactions(List<AccountTransaction> accountTransactions);

Code ‑ **insertAllAccountsTransactions** method

Moving to the back-end server implementation, the endpoint is accessible and used by the Android client application through the **validateUser** method from Code 4‑31. The call of **validateUser** method is realized from **ValidateUserTask** from Code 4‑19. We firstly check that the ID is different from an empty string and then we call **doValidation** (Code 4‑32) method on it.

@PostMapping(value = **"/validateUser"**)  
**public** ResponseEntity<String> validateUser(@RequestParam(value = **"tokenID"**) String tokenID) {  
 **if** (tokenID.contentEquals(**""**)) {  
 **return** ResponseEntity.*ok*(**"Invalid User"**);  
 } **else** {  
 **try** {  
 **return** ResponseEntity.*ok*(**userValidatorService**.doValidation(tokenID));  
 } **catch** (GeneralSecurityException | IOException e) {  
 e.printStackTrace();  
 }  
 **return** ResponseEntity.*ok*(**"Invalid token"**);  
 }  
}

Code ‑ **validateUser** endpoint

**doValidation** method (Code 4‑32) builds a request to Google based on the received **tokenID** and the private key "xxxxxxxxxxx.apps.googleusercontent.com" which is censored. If the **tokenID** is valid, we will receive a **Payload** which contains all the basic information about the user. Then, verify if the user is already in the database and if not, we add him to the database. Finally, we generate and respond with a session ID which will permits the user to make further requests from his application instance.

*// Build the request*GoogleIdTokenVerifier verifier = **new** GoogleIdTokenVerifier.Builder(transport, mJFactory)  
 .setAudience(Collections.*singletonList*(**"xxxxxxxxxxx.apps.googleusercontent.com"**))  
 .build();  
  
*// Receive the response*GoogleIdToken unpackedTokenID = verifier.verify(tokenID);  
**if** (unpackedTokenID != **null**) {  
 Payload payload = unpackedTokenID.getPayload();

*// Get profile information from payload* **long** userId = Long.*parseLong*(payload.getSubject().substring(0, 10));  
 String email = payload.getEmail();  
 String name = (String) payload.get(**"name"**);  
 String pictureUrl = (String) payload.get(**"picture"**);  
 String locale = (String) payload.get(**"locale"**);  
 String familyName = (String) payload.get(**"family\_name"**);  
 String givenName = (String) payload.get(**"given\_name"**);  
 *// Check if the user is already in the database* **if** (DBConnect.*getUser*(userId) == **null**) {  
 **try** {  
 numberOfInsertedUsers = DBConnect.*addUser*(**new** User(  
 userId,  
 name,  
 email,  
 pictureUrl,  
 locale,  
 familyName,  
 givenName  
 ));  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 System.***out***.println(**"Inserted rows: "** + numberOfInsertedUsers);  
 *// Return a SessionID to validate further server requests* **return** UUID.*randomUUID*().toString();  
} **else** {  
 System.***out***.println(**"Invalid ID token."** + unpackedTokenID);  
 **return "Invalid Token ID"**;  
}

Code ‑ **doValidation** method

## Testing

For testing the android application, I used two libraries. Those libraries helped me in manually testing the application. Also, for easing the manual process of testing the application, at the build time, this happens only at the first access of the application, I am seeding the database with test data. The added test data consists in wallets with categories and transactions. **ieobjects** stands for income or expense objects, as the same object can represent an income or an expense.

These two testing libraries and the data generator are meant only for the debug version of the application, and they will not be present in the production application.

**private static** LocalRoomDatabase buildDatabase(**final** Context appContext,  
 **final** AppExecutors executors) {  
 **return** Room.*databaseBuilder*(appContext, LocalRoomDatabase.**class**, ***DATABASE\_NAME***)  
 .addCallback(**new** Callback() {  
 @Override  
 **public void** onCreate(@NonNull SupportSQLiteDatabase db) {  
 **super**.onCreate(db);  
 executors.diskIO().execute(() -> {  
 *// Generate the test data* LocalRoomDatabase database = LocalRoomDatabase  
 .*getInstance*(appContext, executors);  
 List<Wallet> wallets = DataGenerator.*generateWallets*();  
 List<CategoryObject> categories = DataGenerator  
 .*generateCategoriesForWallets*(wallets);  
 List<IEObject> ieobjects = **new** ArrayList<>();  
 **for** (Wallet w : wallets) {  
 ieobjects.addAll(  
 DataGenerator.*generateIEForCategories*(w.getName()  
 , categories,w.getId()));  
 }  
 *// Seed the generated data into the database  
 insertData*(database, wallets, categories, ieobjects);  
 *// Notify that the database has been created* database.setDatabaseCreated();  
 });  
 }  
 })  
 .allowMainThreadQueries()  
 .build();  
}

Code ‑ **buildDatabase** method

By seeding the database at its each creation (Code 4‑33), it helped me to test the application across multiple devices, having the data already in place. It was also very useful in the case I was clearing the storage or reinstalling the application.

The first library I used is called Canary Leaks, which is described in Chapter 3.2. All I had to do in order to use this library was to add the line from bellow in app’s build.gradle.



This library automatically deploys an application on the Android device, which is permanently scanning my application for different types of leaks. If a leak is detected, a notification with leak information is received. This library can be useful also as a kind of regression testing.

The second library I used for debugging and testing my application is called Room Explorer. Like for the first library, I had to add the line from bellow in the app’s build.gradle.



After importing the library, I decided to start the **Activity** of this explorer by pressing the volume down key (Code 4‑34). The need of using this library came when I started to have multiple tables and some data was not being inserted into the Database. Using this library, I was able to view the content of each table and manually write queries. It was also helpful in order to build and test my queries by pressing the “Custom Query” button from (Figure 4‑13).

*// Method which can catch Key events*@Override  
**public boolean** dispatchKeyEvent(KeyEvent event) {  
 **int** keyCode = event.getKeyCode();  
 **switch** (keyCode) {  
 **case** KeyEvent.***KEYCODE\_VOLUME\_UP***:  
 *// Check if the event is related to the volume down button* **case** KeyEvent.***KEYCODE\_VOLUME\_DOWN***: {  
 *// Check if the caught event is indicating the press action,  
 // not the release action.* **if** (event.getAction() != KeyEvent.***ACTION\_DOWN***) {  
 *// Start the activity of Room Explorer,   
 // giving as parameter also the class of our Room Database  
 // and the name of our application database* RoomExplorer.*show*(**this**,  
 LocalRoomDatabase.**class**,  
 **"budgetize-db"**);  
 **return true**;  
 }  
 }  
 **default**:  
 **return super**.dispatchKeyEvent(event);  
 }  
}

Code ‑ **dispatchKeyEvent** method

The results are displayed in a table. It also displays a message with the status of the query and the number of rows returned, for a select clause, as it is illustrated in (Figure 4‑14).

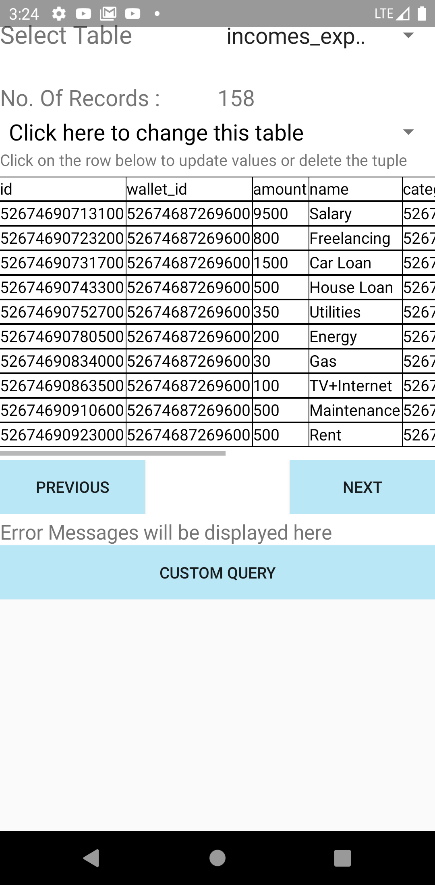


Figure ‑ Room Explorer

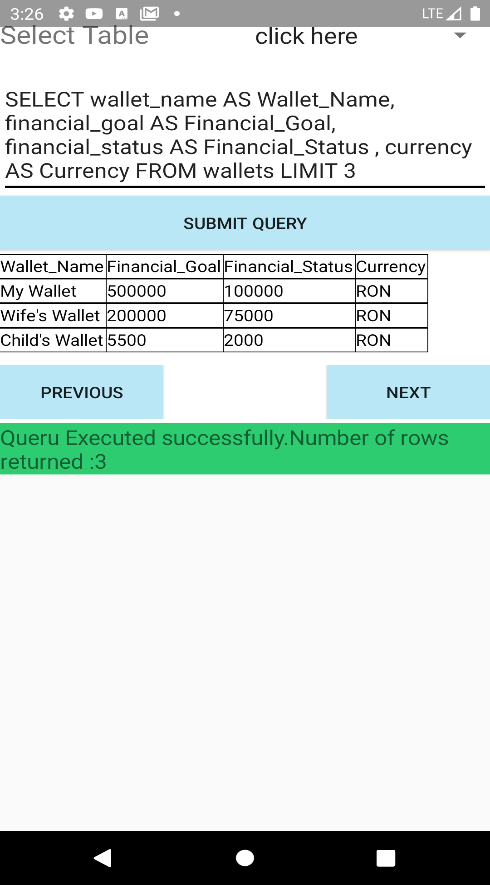


Figure ‑ Room Explorer Custom Query Example

Using the test methods from above, not only helped me to test the application and to avoid introducing new bugs, but also speeded up the development process.

# User Guide and Experimental Results

## Usage

At the first start of the application the user is asked to give access to media files (Figure 5‑1). Right after, the user must press on the biometric button and proceed the biometric login (Figure 5‑2). Then the user has access to the application.

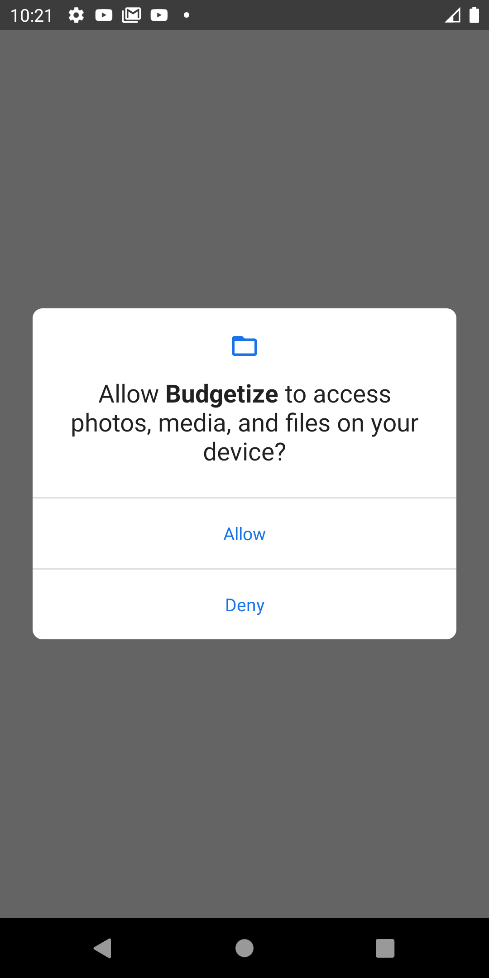


Figure ‑ Media Files Access Request

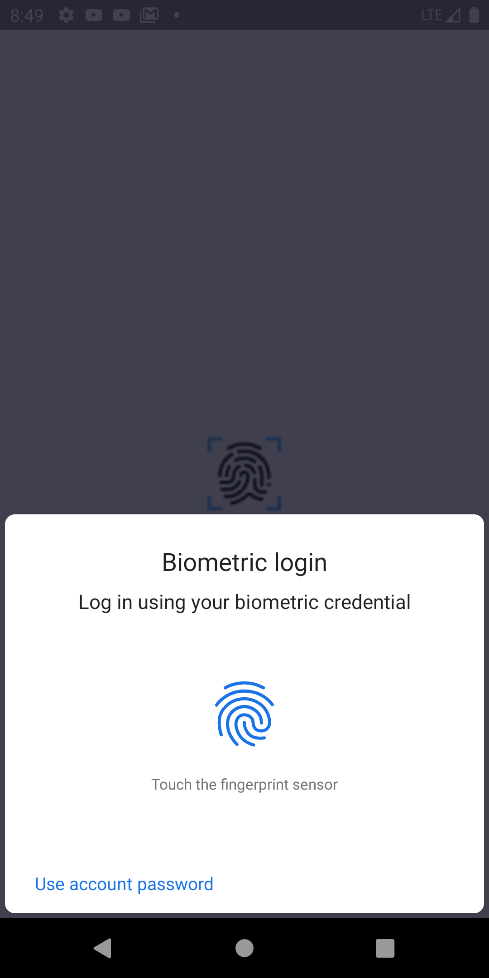


Figure ‑ Biometric Login Screen

At every restart of the application, the user must authenticate using the biometric credentials. On a successfully login, the user will get access to the application (Figure 5‑3), otherwise a warning message will pop-up (Figure 5‑4) and the user must press on the fingerprint button from the center of the screen to in order to be able to perform the biometric authentication again.

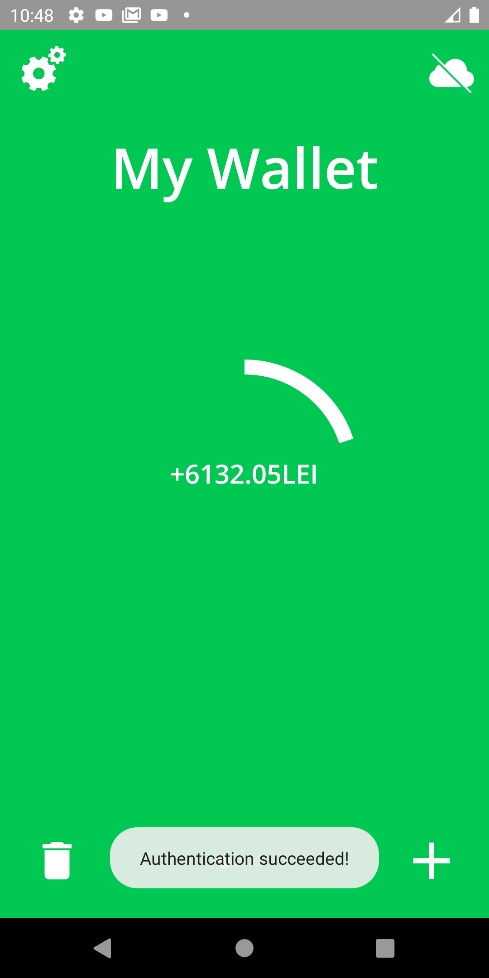


Figure ‑ Main Screen

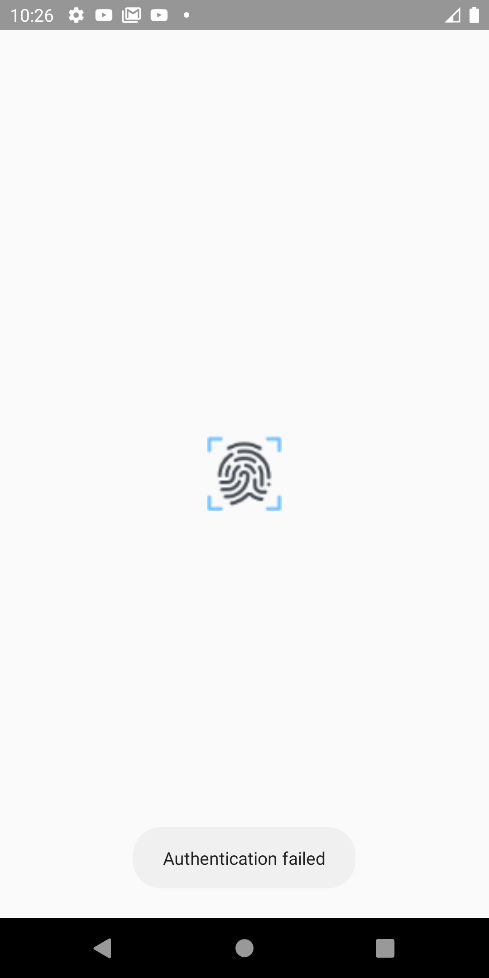


Figure ‑ Biometric Login Screen

When the user gets access to the application, the main screen pops-up (Figure 5‑3). If there are no wallets created, a message indicates the user to create his first wallet (Figure 5‑5). At the creation step, he must select the currency of the wallet, provide a wallet name, his financial status and a financial goal that he wants to achieve (Figure 5‑6). After the “Create Wallet” button is pressed, the user is redirected to the main screen, where he can see the first wallet created (Figure 5‑7). A progress-bar will load, indicating how close the user is to achieve his budget goal. In the middle of the progress-bar is displayed with how much money the user remains at the end of the month, based on the incomes and expenses the user assigned to that wallet. Above the progress bar, we have the name of the displayed wallet.

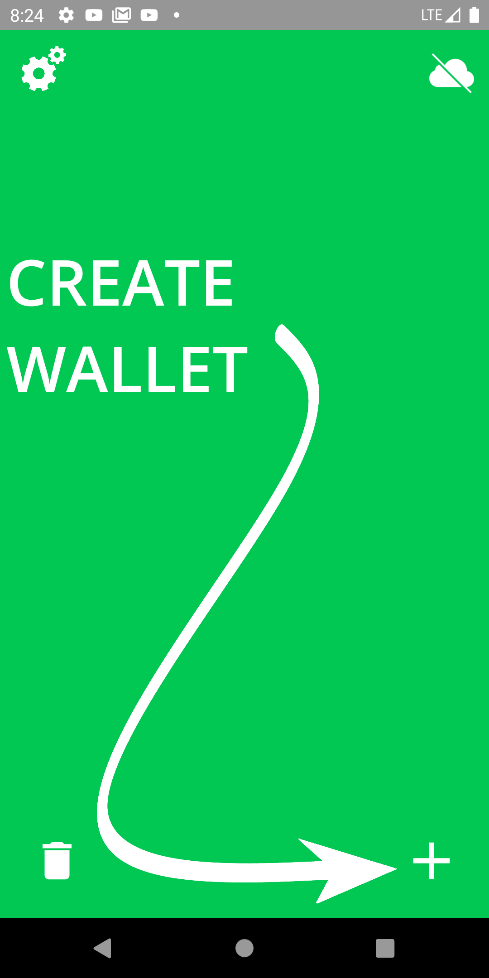


Figure ‑ Empty Main Screen

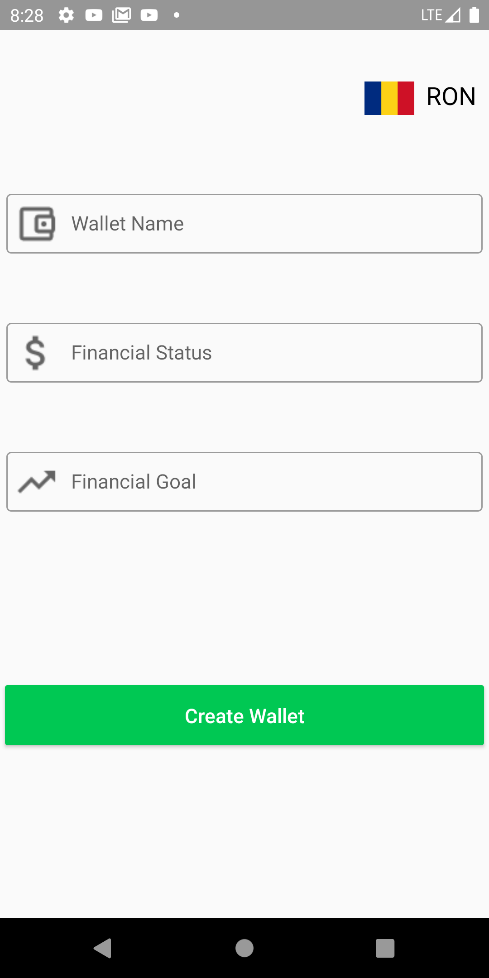


Figure ‑ Create Wallet Screen



Figure ‑ Main Screen

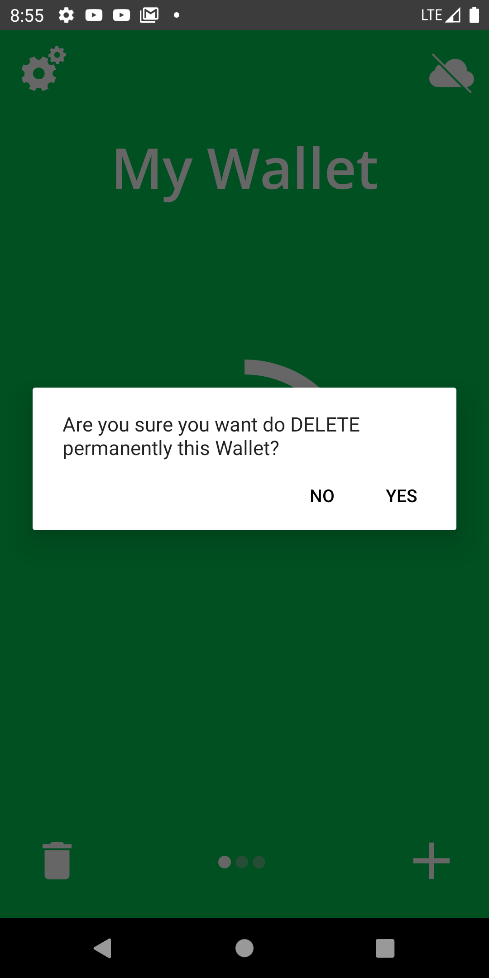


Figure ‑ Delete Wallet Dialog

The user can add multiple wallets and scroll to right and left through them. In the bottom left part of the screen, there is a wastebasket icon with which the user can delete the focused wallet (Figure 5‑8). The deletion of the wallet implies the deletion of all the categories and transactions permanently.

In the left upper part of the main screen there is a settings button, for further settings implementations.

In the right upper part of the main screen is a crossed cloud icon which when is pressed, it starts the Google sign in process (Figure 5‑9) which afterwards activates the Special Features of the application. We will cover that button, and the screens that it enables, after we finish with the Basic Functionalities.

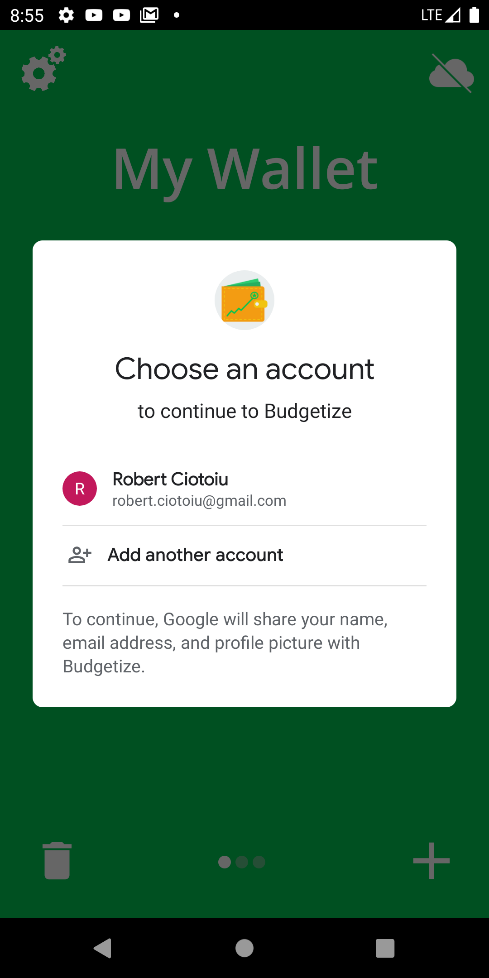


Figure ‑ Google Signin Screen

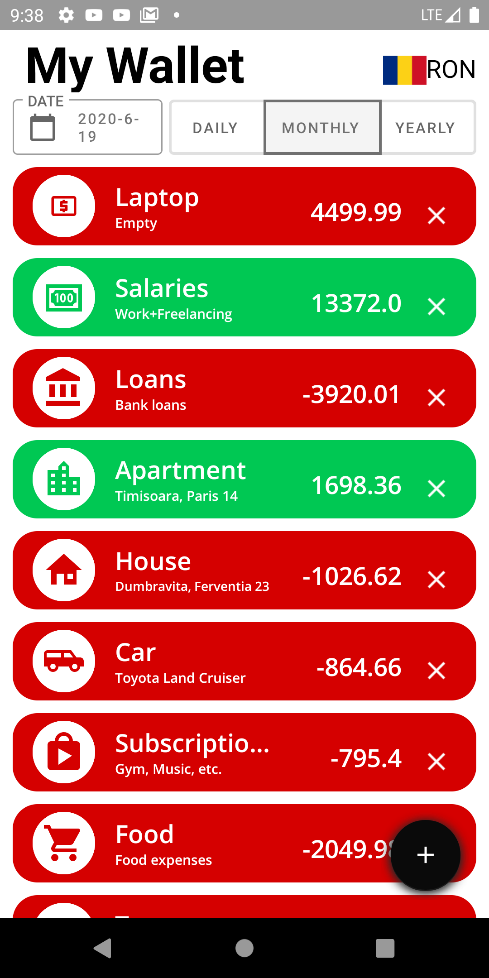


Figure ‑ Transactions’ Screen

If we press inside the wallet’s progress bar (Figure 5‑3), we can enter inside the focused wallet. A new screen pops-up, in which we have in the top left corner the wallet name and in the top right corner the currency. Bellow the wallet name we have the Date Filter, and, in its right, we have the Time Frame filter. Then we have the categories and the transactions. Also, in the bottom right part of the screen we have an “add” button which on press expands 3 sub-buttons (Figure 5‑13). The first sub-button allows the user to create a new transaction, the second one a new category, and the last one to import transactions from the linked bank account. The last one is considered to be one of the Special Functionalities, and it is visible only after the user has linked a bank account to the application. Each of this button redirect the user to a sub-screen, where he can very easily create/import the entity he needs.

By pressing on the “Create Transaction” Button, we will be redirected to the Create Transaction Screen (Figure 5‑11). There the user can select if the transaction is an income or an expense from the top screen buttons, set the name, the amount, the currency, by pressing on the flag, the transaction category, the transaction occurrence and the transaction date. Moving to the Create Category Screen (Figure 5‑12), the user can set a category name and a description, and pick a category icon by pressing on the text box. After the user sets the category name, a relevant icon will be automatically set by the application, in case the user forgets to set it manually.

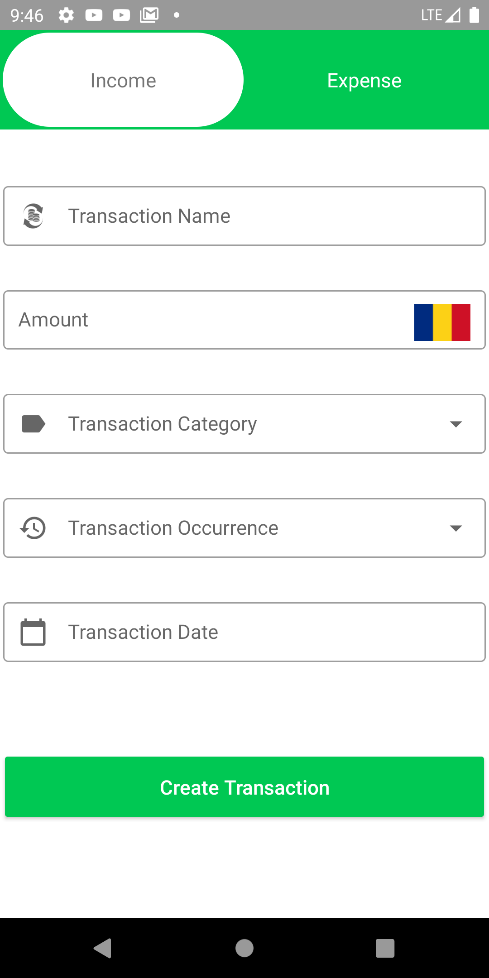


Figure ‑ Create Transaction Screen

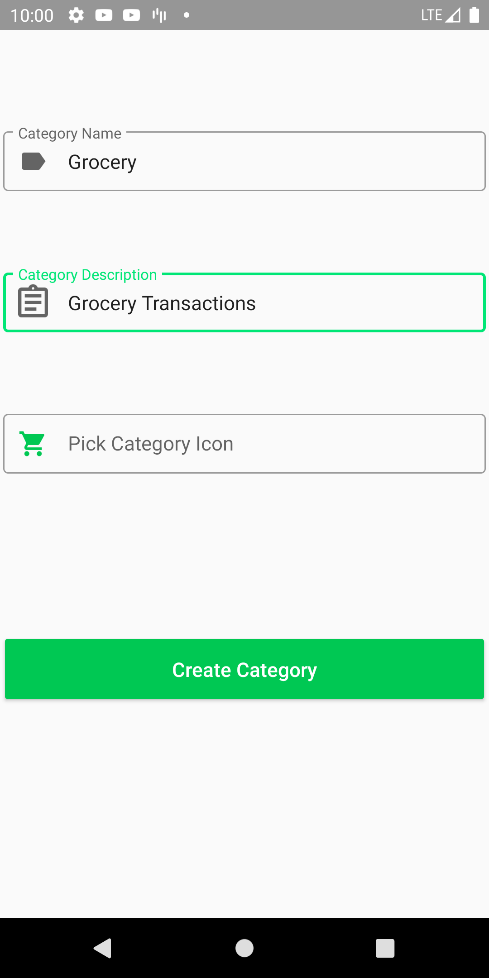


Figure ‑ Create Category Screen

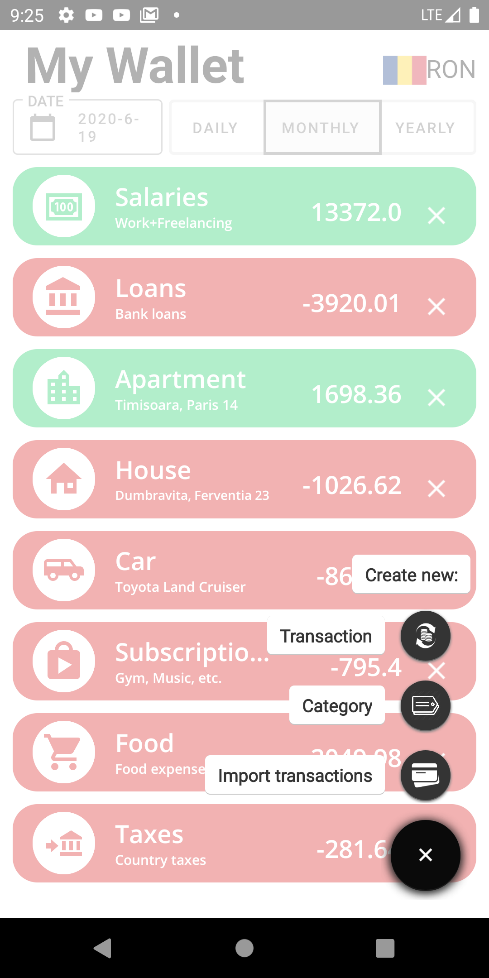


Figure ‑ Create Buttons

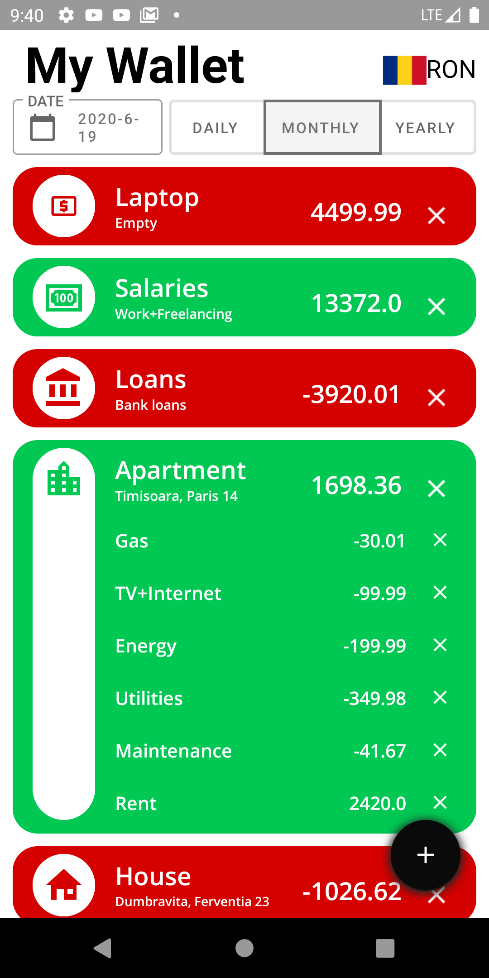


Figure ‑ Expanded Category

Returning to the Transactions’ Screen, after the uses added different types of transactions to this wallet, each of them in different types of currencies, the user can simply display all the transactions in the currency he wants, by using the upper right button from the screen. This simplifies the way of viewing the income of expenses. Also, he is able to filter the transactions selecting the date and the time frame. For example, in order to see all the incomes and expenses from June 2020, the user needs to pick any day from June, and select the Monthly time frame.

The Categories are differentiated from the Orphan Transactions by the icon they have in front of them. Transactions will always have a small dollar bank-note icon (first transaction from Figure 5‑14), while the transactions can have any icon chosen by the user. In order to view the transactions from a category, we can simply press the Category’s icon. The Category will expand and will display all the attached transactions.

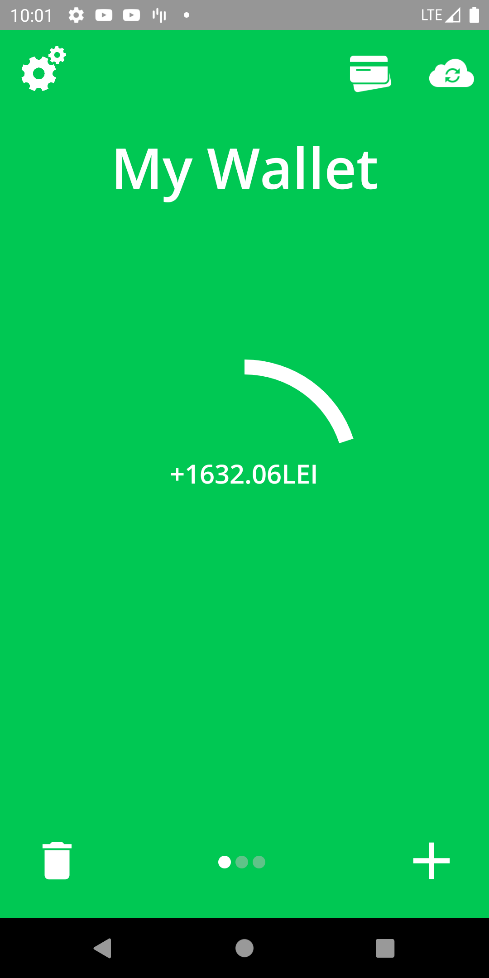


Figure ‑ Main Screen After Google Sign in

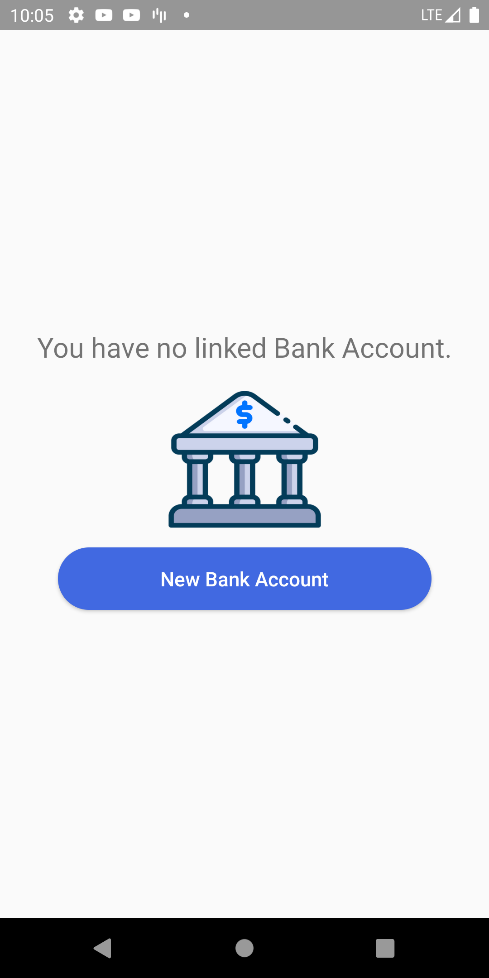


Figure ‑ Empty Bank Accounts Screen

Finishing with the Basic Functionalities, I will move on to describe the Special Functionalities of the application. The first step the user is required to do in order to activate the Special Functionalities is to press on the crossed cloud from the right upper corner of the Main Screen. This will pop-up an overlay screen asking the user to add or choose a Google account, permitting the application to read the name, email address and profile picture of that google account (Figure 5‑9). After a successful login-in, a new icon will appear on the top of the screen (Figure 5‑15). If the user press again on the cloud button, a new Dialog will pop-up, asking the user if he wants to logout.

The new button is positioned in the left of the cloud button and is called bank accounts button. Pressing it, the bank accounts screen will pop-up. At the beginning this screen will be empty, having only a button, an icon, and an informative message (Figure 5‑16).

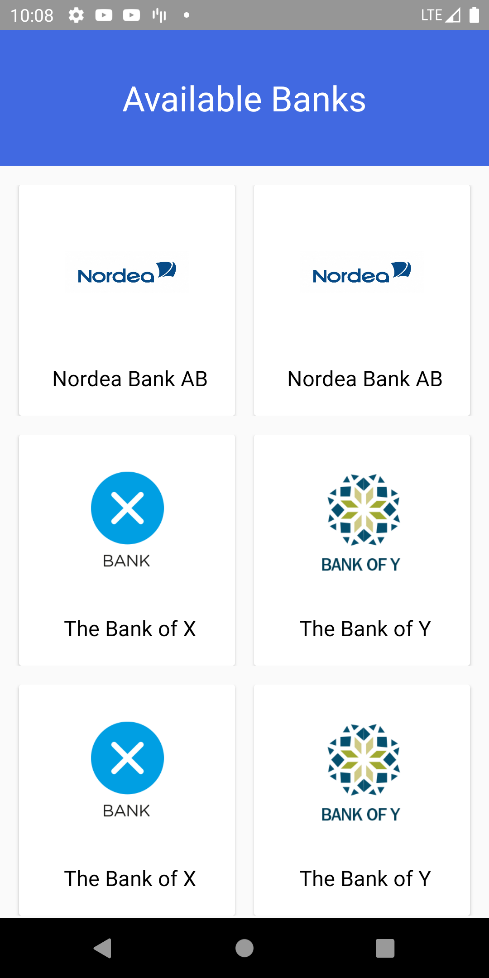


Figure ‑ Available Banks Screen

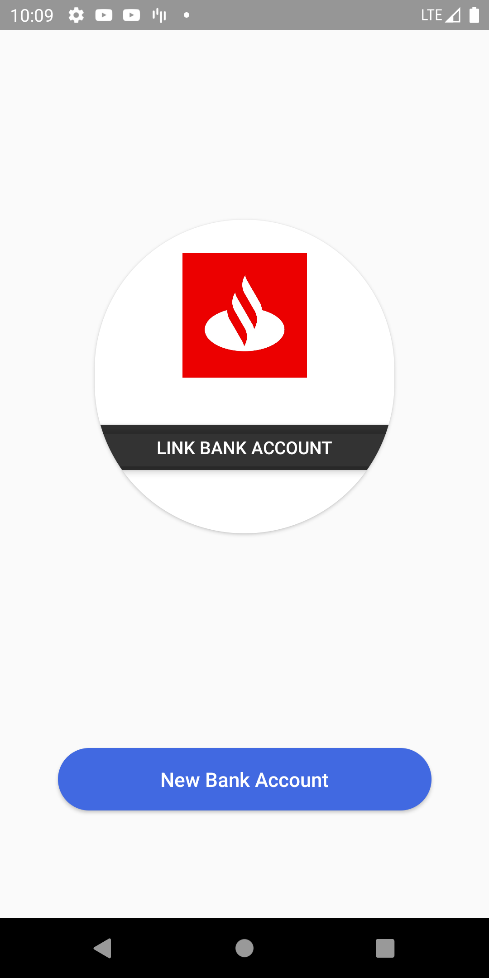


Figure ‑ Populated Bank Accounts Screen

Pressing that button, a new screen pops-up containing all the available banks (Figure 5‑17). The user can search and select the bank where he has an account. This will redirect the user back to the bank account screen. The selected bank will be displayed in a circle card view, containing the bank icon and a button which indicates the link status (Figure 5‑18).

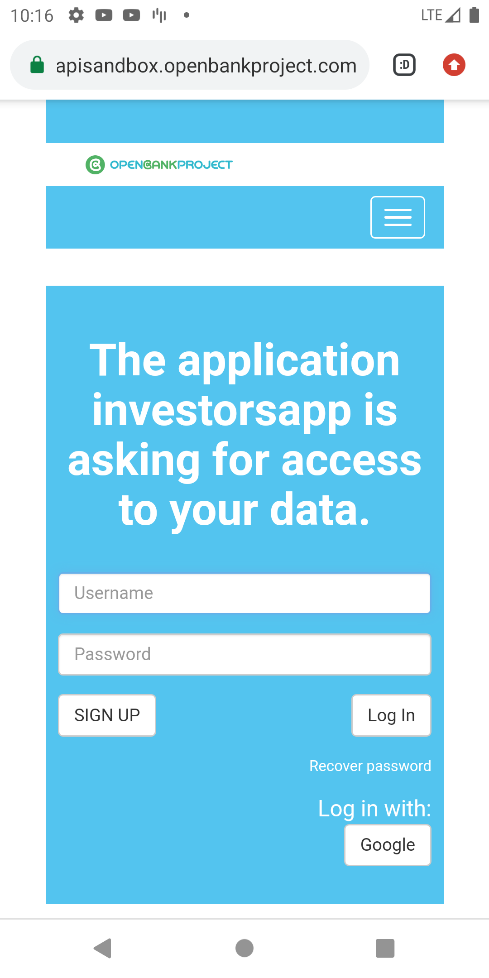


Figure ‑ Bank Login Webpage

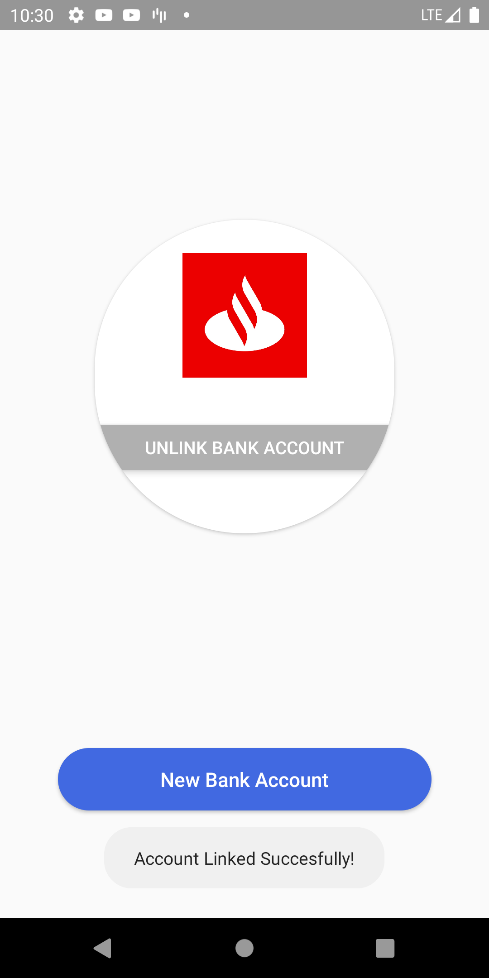


Figure ‑ Bank Accounts Screen With A Linked Bank Account

Initially, this button will have the text “Link Bank Account”. Pressing it, the browser application will start, redirecting the user to the bank account website, in order to provide his credentials directly to the bank (Figure 5‑19), and the consent that *Budgetize* will have the right to access his transaction history on his behalf. After the user validates his credentials to the bank and completes the link process, it will be redirected back to *Budgetize*.On the bank account screen, after a successfully bank account linkage, the button from the card view will have the text “Unlink Bank Account” (Figure 5‑20). The user can add as many bank accounts as he wants and repeat the same link process. He can stop *Budgetize*‘s access to a bank account by pressing the button “Press to unlink”. This will revoke the right of *Budgetize* to further access any data from that bank account and all the data related to that bank account will be deleted from the application. The revoke request is sent automatically to the bank which will abrogate *Budgetize*’s access.

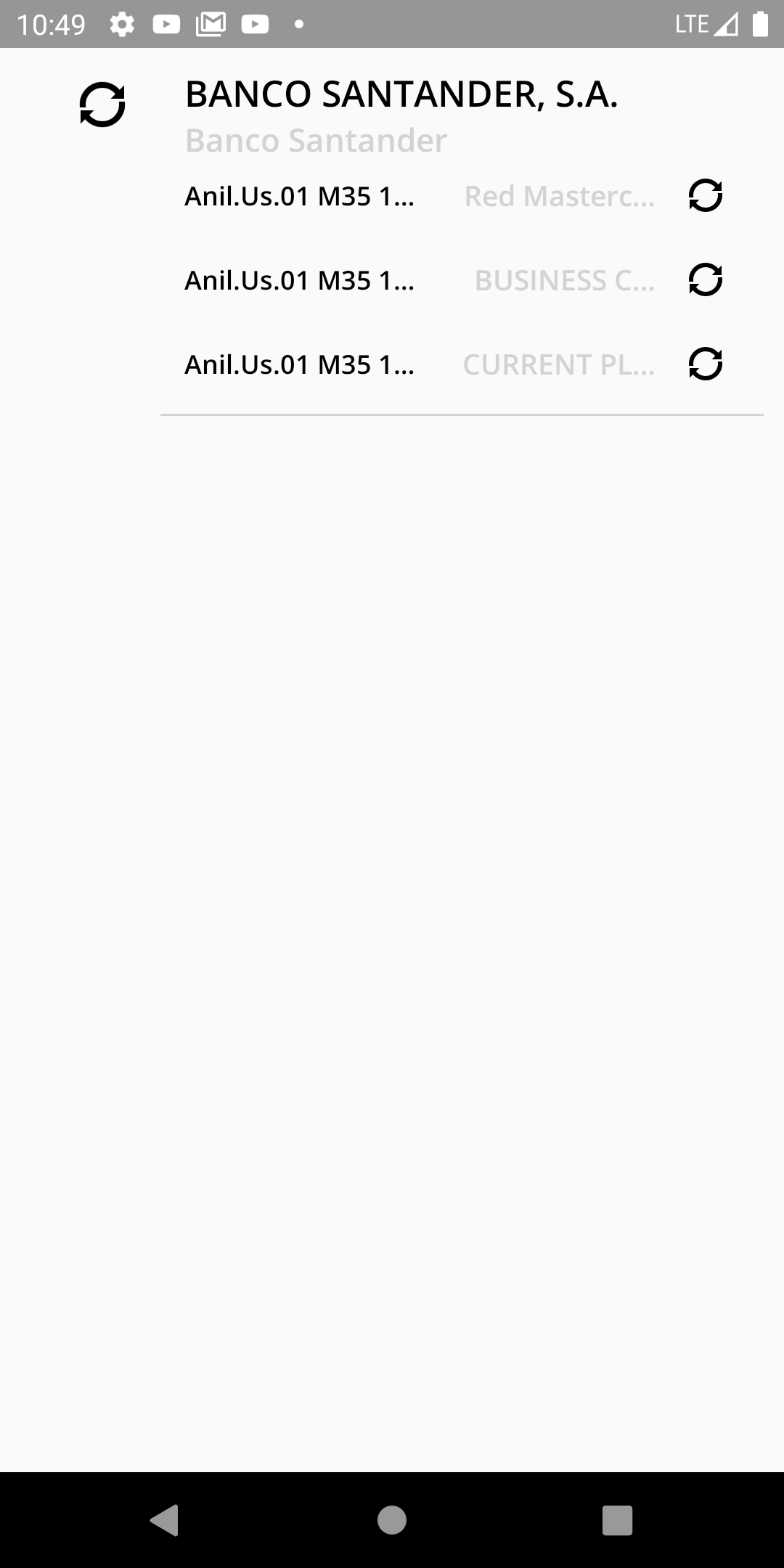


Figure ‑ Import Transactions’ Screen

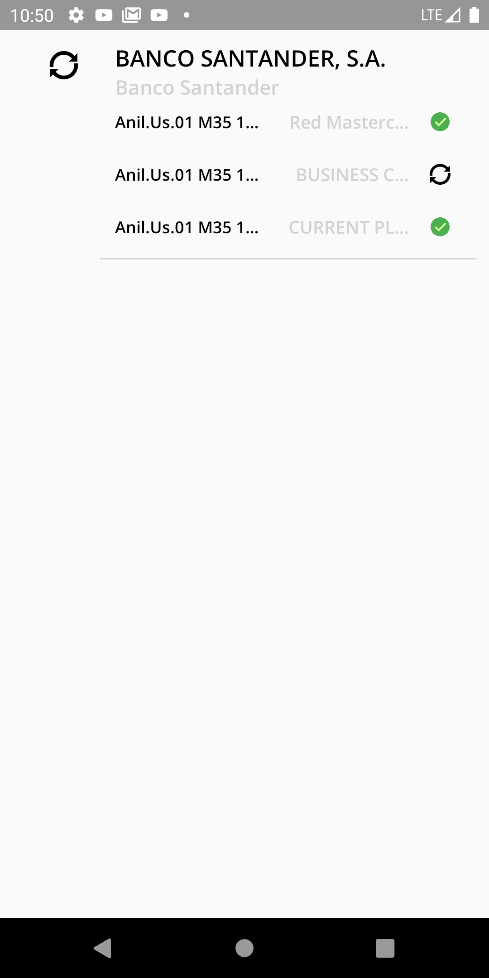


Figure ‑ Import Transactions’ Screen With Linked Accounts

After the user has linked a bank account to the application, we can go inside of any wallet, and use the “Import Transaction” button. This button will pop-up a screen which contains all the linked banks. A linked bank can contain multiple bank accounts. The user can click on each linked bank in order to view all the bank accounts from that bank (Figure 5‑21). The user has the ability to choose to link all the banks accounts from a linked bank, or just some of them. This can be done via the right buttons from each entity (Figure 5‑22). Doing this, all the past and future transactions are added automatically to the respective wallet.

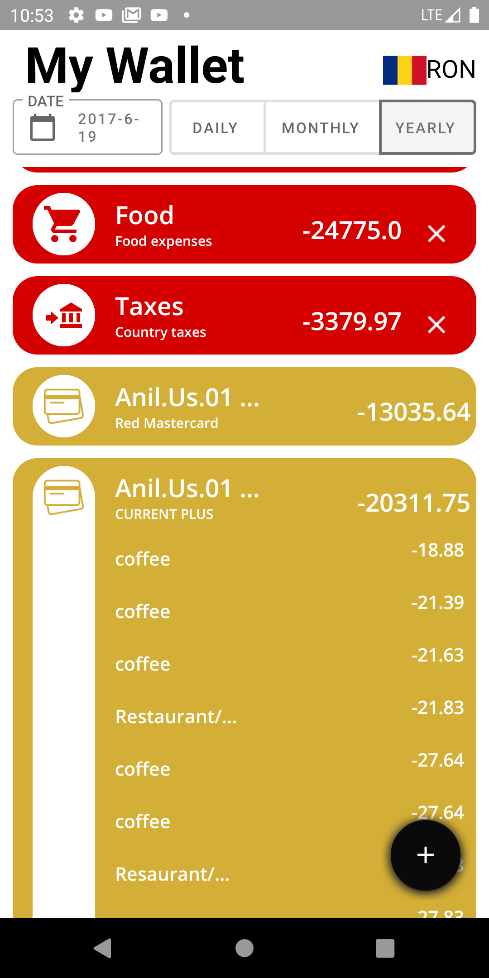


Figure 5‑23 Transactions’ Screen With Imported Bank Accounts’ Transactions

More precisely, a new category will be created, having the color of gold, for each bank account, containing all the transactions from that bank account (Figure 5‑23). This has been done in order the user to easily differentiate the imported transactions from the manually added transactions. This step can be repeated for any wallet.

## Experimental Results

In this chapter, I would like to present some measurements and comparations between the old and new approaches I had in my application. All the measurements will be performed on Pixel 3 Virtual Device, Sony Xperia XZ Premium, Samsung Galaxy S8 and Huawei P20 Lite, which will be found on the horizontal axis. On the vertical axis, we have the measured time in milliseconds. All the tests will be performed on GPRS, 3G and 4G mobile networks for each real mobile device. I added the Pixel 3 Virtual Device in the comparisons as a Ground Truth, because it runs on the same network and machine as the back-end server. The machine which host them has a Ryzen 3700X processor, 16 GB of RAM DDR4 running at 2667MHz, Kingston A400 SATA3 solid-state drive with the Write Speed up to 450MB/s and Read Speed up to 500MB/s and an Nvidia RTX 2060 video card.

The first measurement is based on how much it takes for the application to initialize all the starting components (Figure 5‑24), retrieve all the data, and show them on the main screen. Shortly, this is the measurement for the **onCreate** and **onResume** methods from the **MainActivity** class, which I presented in Chapter 4.4. These results do not include the wait time of the asynchronous calls. We are interested in measuring how much time the user has to wait until the main screen is displayed.

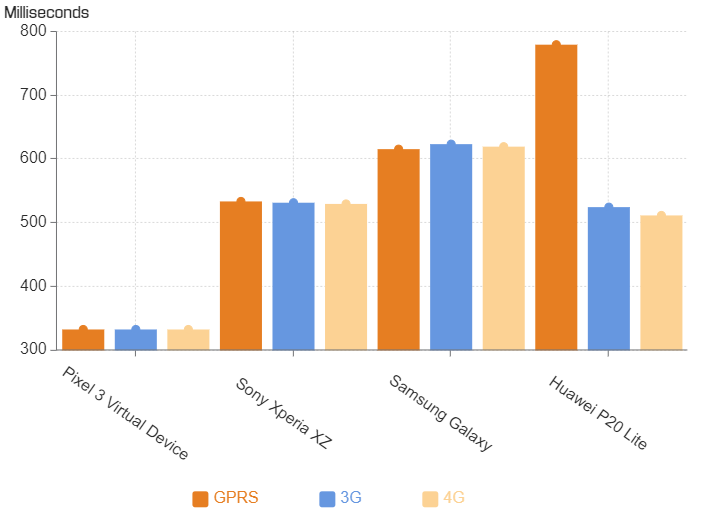


Figure ‑ **onCreate** & **onResume** processing time (Created with: [58])

The longest time registered to perform that flow was 780 milliseconds on a Huawei P20 Lite on GPRS mobile data. Considering that the user must firstly authenticate with his biometrics, the main screen will be display after that almost instantaneous, even the user uses GPRS mobile data.

The next measurement is based on how much it takes to download all the banks’ logo using a single thread, on a fresh installed application (Figure 5‑25).

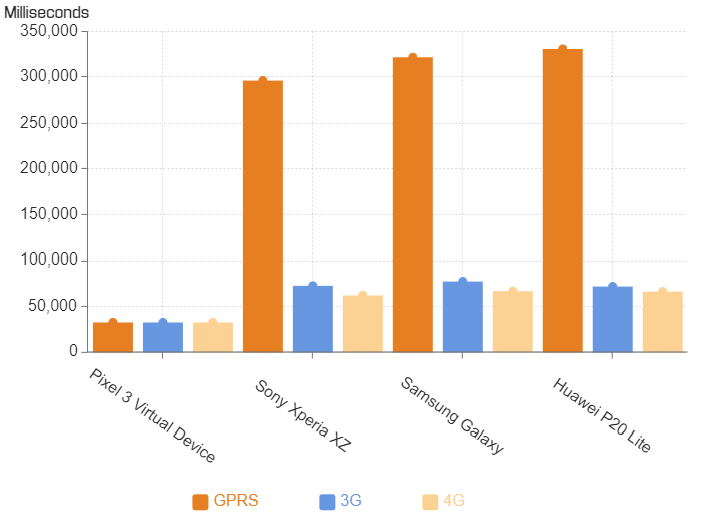


Figure ‑ Fresh installed application banks’ logo download time on one thread (Created with: [58])

It can be seen that the longest time to download and save all the bank logos has been registered on the Huawei P20 Lite on GPRS mobile data. It took around 5 minute and 31 seconds, time in which the user could have Sign in with google and access the “Available Banks” screen. This may result in a negative user experience, as banks could be missing from the list or still loading. Using a better network as 3G or 4G, the bank logos are downloaded in just 1 minute and 7 seconds, which is still not as desired.

As a solution, I made this task to run on multiple threads and we can see the results in Figure 5‑26.



Figure ‑ Fresh installed application banks’ logo download time on multiple threads (Created with: [58])

On a GPRS network, the task is performed 2 minutes faster, a big improvement, but not ideal yet. On the GPRS there can be realized a series of optimizations, but the focus, at the initial development time, was on the 3G and 4G networks. On 3G and 4G, the results are impressive, compared to the GPRS and to the Pixel 3 Virtual Device. For example, the fastest download time of 7 seconds on a 3G/4G network was achieved by the Huawei P20 Lite, meanwhile the slowest time of 9.4 seconds was achieved by the Sony Xperia XZ. My fastest trial to access the available banks’ screen on a fresh installed application was around 11 seconds, so the results on the 3G/4G network are just in time. This is a significant improvement which results in a better user experience.

On an already installed application, to refresh the bank logos and to try download again the unsuccessful ones on a single thread, the results are shown in the Figure 5‑27.

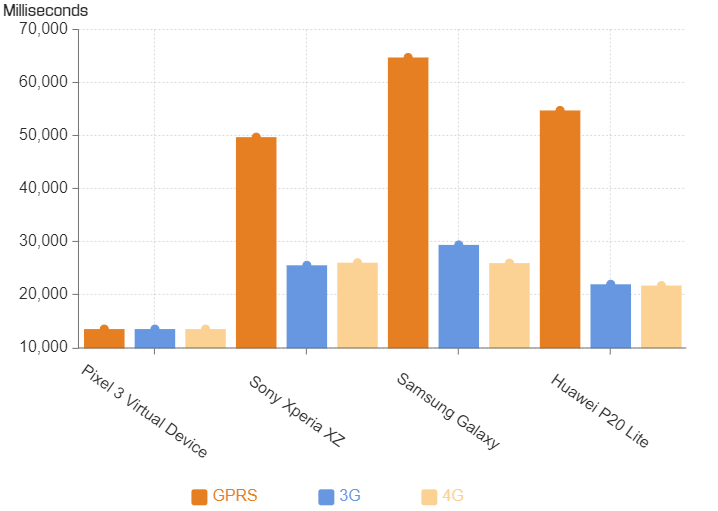


Figure ‑ Refresh banks’ logo using one thread (Created with: [58])

And the same test but on multiple threads can be seen in the Figure 5‑28.

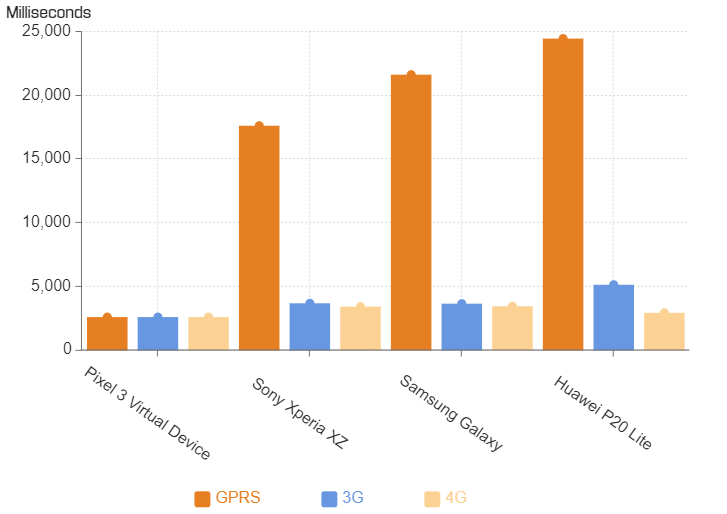


Figure ‑ Refresh banks’ logo using multiple threads (Created with: [58])

We can see that even there is a difference of at least 20 seconds on the 3G/4G networks, and 30-40 seconds on the GPRS mobile data.

The last performed measurement is how much it takes to perform the Google Sign-in flow from Chapter 4.3.2. In this test, I measured only the elapsed time from when the Android application starts to build the authorization request to our back-end server until the Session ID is received.

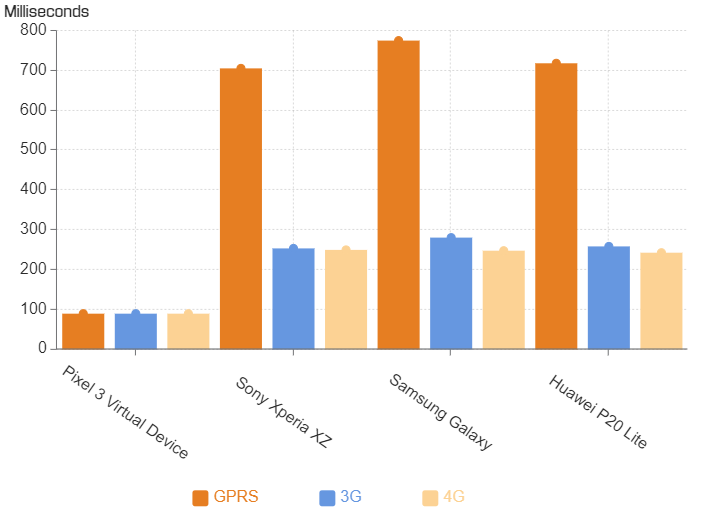


Figure ‑ Google Sign-in flow process time (Created with: [58])

All the results are under 0.5 seconds for a good internet connection (Figure 5‑29). Although, from a user experience perspective, the registered waiting time is insignificant. Even using GPRS the results are under 800 milliseconds, but we are having a loading progress bar in any case, to let the user know that the request is being processed. Also, there is a safe timeout after 10 seconds, avoiding other crashes and errors which may lead to a bad user’s experience.

# Conclusions

The main purpose of *Budgetize* was to demonstrate that the process of managing our money can be eased a lot by having the ability to import the transactions from our bank accounts. Also, the ability to insert transactions manually and to categorize them in a simple way, but not least to personalize them as close as possible to the reality. This was achieved by allowing the user to set the amount, the date, the occurrence, the type, and a different currency for each transaction. Also, the auto category icon chooser, eases user’s life, by automatically picking a suggestive icon for the category, based on the set category name. Another important realized feature is the ability to view in a pleasant and simple way the transactions, filtering by the date and the timeframe, and choosing the currency of the entire wallet. A very important principle I followed in order to develop the application was to have a simplistic, intuitive, good-looking design.

A very big challenge in developing and even designing the application was to find a way to allow linking bank accounts to the application in order to be able to import transactions into the wallets. I firstly studied the documentation of Open Banking APIs from ING and Revolut. Observing that the process of connecting to the API and retrieving transactions is slightly different, and each bank has different conditions to access even their sandbox environment, I decided to search for a middleware platform which provides the possibility to connect to multiple banks with one implementation. I found Open Bank Project and after I studied their documentation, I started the implementation. Incrementally, I managed to realize the complete flow from linking a bank account until importing transactions. I also tried to make the implementation as modular as possible, in order to be able to use the same code to access multiple banks. Finally, I managed to simulate the link process of multiple banks, by allowing to connect multiple bank accounts from the Open Bank Project API Sandbox to the application.

Another big challenge for me was to make a clear, simple, and intuitive design. At least half of the work time was invested on the design part, drawing ideas on the paper, and trying to transpose them into the application. At the beginning it was not easy to work with UI elements like **ViewPagers**, **CardViews**, **TabLayouts**, but after several trials, I understood their working mechanism.

## Future improvements

Even that I managed to achieve my main purpose, described in the conclusions, there are still some improvements needs to be done and some features to finish. Some of the improvements are on the security part. The first one, is to move all the HTTP communication to HTTPS and to encrypt and decrypt the transported sensitive data.

Another security issue needs to be solved is that the access tokens, used to make requests to the bank on the user’s behalf, needs to be encrypted before being written into the shared preferences and decrypted when retrieved.

Beside the security part, there is room to improve the design. It is known that the design wears out and the longest-lived applications from the market, adapts their design regularly. This part will be continuously maintained, following the principles of being simple and intuitive.

Referring to the next features, a screen with different statistics about the incomes and expenses should be developed. Also, I would like to implement a personal assistant using artificial intelligence, which will make smart recommendations on what expenses to reduce and how those suggestions will improve user’s budget on a long term.

Another important feature would be to allow the user to modify the already created wallets, categories, and transactions. For the moment, the user is able only to create and delete those entities.

We have a back-end server which generates a session ID for further requests authorization combined with a database structure which unlocks the possibility of building the cloud data saving feature. Also, because the Android application has a modular architecture as shown in Figure 4‑2, the cloud data saving feature can be implemented effortless.

Another future improvement is to change all the variables which refers to money from **double** to **BigDecimal**.

Finally, as I mentioned in Chapter 4.3.3, we can currently import transactions only from test user accounts, so a future improvement would be to obtain the needed certifications in order to be able to link real bank accounts to *Budgetize*.

# References

|  |  |
| --- | --- |
| [1] | J. MacKay, "RescueTime:blog," RescueTime, 21 March 2019. [Online]. Available: https://blog.rescuetime.com/screen-time-stats-2018/. [Accessed 20 May 2020]. |
| [2] | S. O'Dea, "Statista," Statista, 28 Feb 2020. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Accessed 20 May 2020]. |
| [3] | "Mobile Operating System Market Share Worldwide - April 2020," StatCounter, April 2020. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed 21 March 2020]. |
| [4] | "Google Play," Google, [Online]. Available: https://play.google.com/store/apps/details?id=com.cleevio.spendee&hl=ro. [Accessed 23 May 2020]. |
| [5] | M. Rouse, "TechTarget," [Online]. Available: https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP. [Accessed 05 May 2020]. |
| [6] | R. Half, "Robert Half," Robert Half International Inc., 13 July 2017. [Online]. Available: https://www.roberthalf.com/blog/salaries-and-skills/4-advantages-of-object-oriented-programming. [Accessed 05 May 2020]. |
| [7] | D. Bolton, "ThoughCo.," 10 February 2019. [Online]. Available: https://www.thoughtco.com/definition-of-encapsulation-958068. [Accessed 24 May 2020]. |
| [8] | G. Miglani, "GeeksforGeeks," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/abstraction-in-java-2/. [Accessed 05 May 2020]. |
| [9] | "Android Developers," Google, [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle. [Accessed 25 May 2020]. |
| [10] | "Android Developers," Google, [Online]. Available: https://developer.android.com/topic/libraries/architecture/livedata. [Accessed 25 May 2020]. |
| [11] | "NTC Hosting," NTC Hosting, [Online]. Available: https://www.ntchosting.com/encyclopedia/databases/structured-query-language/#SQL\_Language\_elements. [Accessed 26 May 2020]. |
| [12] | "codecademy," CODECADEMY, [Online]. Available: https://www.codecademy.com/articles/what-is-crud. [Accessed 26 May 2020]. |
| [13] | R. Chan, "Business insider," Business Insider Deutschland GmbH, 22 January 2019. [Online]. Available: https://www.businessinsider.de/international/the-10-most-popular-programming-languages-according-to-github-2018-10/?op=1. [Accessed 26 May 2020]. |
| [14] | "Tutorials Point," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/java/java\_overview.htm. [Accessed 26 May 2020]. |
| [15] | "Oracle," Oracle, [Online]. Available: https://www.oracle.com/technetwork/java/intro-141325.html. [Accessed 26 May 2020]. |
| [16] | "GeeksForGeeks," GeeksForGeeks, [Online]. Available: https://www.geeksforgeeks.org/garbage-collection-java/. [Accessed 26 May 2020]. |
| [17] | "Tutorials Point," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/xml/index.htm. [Accessed 26 May 2020]. |
| [18] | "Gradle," Gradle Inc., [Online]. Available: https://docs.gradle.org/current/userguide/what\_is\_gradle.html. [Accessed 26 May 2020]. |
| [19] | "Gradle," Gradle Inc., [Online]. Available: https://gradle.org/. [Accessed 26 May 2020]. |
| [20] | "Maven Apache," The Apache Software Foundation, [Online]. Available: https://maven.apache.org/what-is-maven.html. [Accessed 25 June 2020]. |
| [21] | B. Petrovan, "Android Authority," Android Authority, 26 February 2012. [Online]. Available: https://www.androidauthority.com/android-everywhere-10-types-of-devices-that-android-is-making-better-57012/. [Accessed 27 May 2020]. |
| [22] | "Source Android," Android, [Online]. Available: https://source.android.com/devices/tech/dalvik/. [Accessed 27 May 2020]. |
| [23] | "Source Android," Google, [Online]. Available: https://source.android.com/devices/tech/dalvik/jit-compiler. [Accessed 27 May 2020]. |
| [24] | "Android Developers," Google, [Online]. Available: https://developer.android.com/guide/platform. [Accessed 27 May 2020]. |
| [25] | "Android Developers," Google, [Online]. Available: https://developer.android.com/ndk. [Accessed 27 May 2020]. |
| [26] | "TutorialsPoint," TutorialsPoint, [Online]. Available: https://www.tutorialspoint.com/spring\_boot/spring\_boot\_introduction.htm. [Accessed 28 May 2020]. |
| [27] | "JetBrains," JetBrains s.r.o, [Online]. Available: https://www.jetbrains.com/help/idea/discover-intellij-idea.html. [Accessed 27 May 2020]. |
| [28] | "JetBrains," JetBrains s.r.o., [Online]. Available: https://www.jetbrains.com/idea/download/#section=windows. [Accessed 27 May 2020]. |
| [29] | "Developers Android," Google, [Online]. Available: https://developer.android.com/studio/intro. [Accessed 27 May 2020]. |
| [30] | D. Calbimonte, "SQLShack," SQLShack, 02 February 2018. [Online]. Available: https://www.sqlshack.com/history-sql-server-evolution-sql-server-features/. [Accessed 27 May 2020]. |
| [31] | A. Jan, "QuickStart," QuickStart, 18 January 2018. [Online]. Available: https://www.quickstart.com/blog/microsoft-sql-server-database-advantages-and-best-practices/. [Accessed 27 May 2020]. |
| [32] | A. Rawat, "Medium," Medium, 2019 February 23. [Online]. Available: https://medium.com/mindorks/using-room-database-android-jetpack-675a89a0e942. [Accessed 28 May 2020]. |
| [33] | W. Karim, "GitHub," GitHub, 08 January 2019. [Online]. Available: https://github.com/wajahatkarim3/RoomExplorer. [Accessed 28 May 2020]. |
| [34] | "LeakCanary," LeakCanary, [Online]. Available: https://square.github.io/leakcanary/. [Accessed 30 May 2020]. |
| [35] | "Ecma International," [Online]. Available: https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-404%201st%20edition%20October%202013.pdf. [Accessed 30 May 2020]. |
| [36] | "W3Schools," W3Schools, [Online]. Available: https://www.w3schools.com/whatis/whatis\_json.asp. [Accessed 30 May 2020]. |
| [37] | Google, "Github," Google, [Online]. Available: https://github.com/google/gson. [Accessed 25 June 2020]. |
| [38] | M. Atwood, D. Balfanz, D. Bounds, R. M. Conlan, B. Cook, L. Culver, B. de Medeiros, B. Eaton, K. Elliott-McCrea, L. Halff, E. Hammer-Lahav, B. Laurie, C. Messina, J. Panzer, S. Quigley and D. Recordon, "oauth," OAuth Core Workgroup, 04 December 2007. [Online]. Available: https://oauth.net/core/1.0/. [Accessed 31 May 2020]. |
| [39] | M. Atwood, D. Balfanz, D. Bounds, R. M. Conlan, B. Cook, L. Culver, B. de Medeiros, B. Eaton, K. Elliott-McCrea, L. Halff, E. Hammer-Lahav, B. Laurie, C. Messina, J. Panzer, S. Quigley, D. Recordon, E. Sandler, J. Sergent, T. Sieling, B. Slesinsky and A. Smith, "OAuth," AOAuth Core Workgroup, 24 June 2009. [Online]. Available: https://oauth.net/core/1.0a/. [Accessed 30 May 2020]. |
| [40] | "OkHttp," Square Inc., [Online]. Available: https://square.github.io/okhttp/. [Accessed 31 May 2020]. |
| [41] | "OkHttp," Square Inc., [Online]. Available: https://square.github.io/okhttp/changelog/#version-3x. [Accessed 31 May 2020]. |
| [42] | "OpenBankProject," TESOBE, [Online]. Available: https://www.openbankproject.com/psd2/. [Accessed 31 May 2020]. |
| [43] | "OpenBankProject," TESOBE, [Online]. Available: https://www.openbankproject.com/openbankingmiddleware/. [Accessed 31 May 2020]. |
| [44] | Google, "Google Developers," Google, [Online]. Available: https://developers.google.com/identity/sign-in/web/sign-in. [Accessed 25 June 2020]. |
| [45] | "Android Developers," Google, [Online]. Available: https://developer.android.com/reference/android/hardware/fingerprint/FingerprintManager. [Accessed 28 May 2020]. |
| [46] | "Source Android," Google, [Online]. Available: https://source.android.com/security/biometric. [Accessed 28 May 2020]. |
| [47] | B. Patel, "https://github.com/birajpatel/GridListViewAdapters," [Online]. Available: https://github.com/birajpatel/GridListViewAdapters. [Accessed 08 June 2020]. |
| [48] | D. Bezerra, "Github," [Online]. Available: https://github.com/diegodobelo/AndroidExpandingViewLibrary. [Accessed 08 June 2020]. |
| [49] | R. Sousa, "Github," [Online]. Available: https://github.com/rubensousa/ViewPagerCards. [Accessed 08 June 2020]. |
| [50] | "Android Developers," Google, [Online]. Available: https://developer.android.com/jetpack/docs/guide. [Accessed 08 06 2020]. |
| [51] | "Github," TESOBE, [Online]. Available: https://github.com/OpenBankProject/Hello-OBP-OAuth1.0a-Android. [Accessed 08 June 2020]. |
| [52] | "DevExpress," DevExpress, 08 April 2020. [Online]. Available: https://docs.devexpress.com/WPF/15112/mvvm-framework. [Accessed 25 May 2020]. |
| [53] | "JSON," JSON, [Online]. Available: https://www.json.org/json-en.html. [Accessed 30 May 2020]. |
| [54] | "Icons8," Icons8 LLC., [Online]. Available: https://icons8.com/. [Accessed 04 June 2020]. |
| [55] | "Android Developers," Google, [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel. [Accessed 05 May 2020]. |
| [56] | JGraph, "JGraph," JGraph, [Online]. Available: https://app.diagrams.net/. [Accessed 08 June 2020]. |
| [57] | D. T. Ltd, "Quick DBD," Dovetail Technologies Ltd, [Online]. Available: https://app.quickdatabasediagrams.com/. [Accessed June 06 2020]. |
| [58] | "Visme," Visme, [Online]. Available: https://www.visme.co/. [Accessed 16 June 2020]. |