TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

IT System Development

Mihhail Skripnik 193956IADB

Semi-automated Budget Management Application

Bachelor's thesis

Supervisor: Tauseef Ahmed
PhD in Electronics and
Telecommunication
Engineering

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

IT Süsteemide Arendus

Mihhail Skripnik 193956IADB

Poolautomaatne eelarvehalduse rakendus

Bakalaureusetöö

Juhendaja: Tauseef Ahmed
Elektroonika ja
Telekommunikatsiooni
tehnika Doktorikraad

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mihhail Skripnik

Abstract

In this dissertation, the author develops a mobile application to significantly facilitate and speed up the process of home accounting. The dissertation aims to provide users with a cross-platform mobile application following the author's concept outlined in this thesis. The application should not limit user-configurable functionality, such as categories, tags, the number of connected banks, and joint budget members. One of the main features of the final product is the functionality that allows users to receive banking transactions for further manual categorization automatically.

The volume of the practical part is limited to devices running iOS and Android mobile operating systems. The main goal of the project is to provide the user with a ready-made mobile solution for joint budget management, using the client and server parts developed by the author within the framework of this thesis and corresponding to the author's vision of the application that is designed to solve the identified problems. The application's name is a short and memorable abbreviation – MYBE, formed from the phrase "Manage Your Budgets Easily". The application created as part of the thesis does not imply any commercial use without significant modifications to comply with all the requirements and standards for this type of software.

The final application consists of two parts: the server part is responsible for processing, storing, and interacting with user data, and the client part is responsible for the visual representation of user data. The server and client parts interact via a secure HTTPS protocol. All data provided by the user is strictly used within the framework of this dissertation. At the same time, the processing of user data by third-party services is carried out only after the user's commitment and is regulated by the relevant supervisory authorities.

This thesis is written in English and is 45 pages long, including 10 chapters, 26 figures, and 1 table.

Annotatsioon

Poolautomaatne eelarvehalduse rakendus

Selles lõputöös arendab autor mobiilirakendust, mis oluliselt lihtsustab ja kiirendab koduarvestuse protsessi. Lõputöö eesmärk on pakkuda platvormшst sõltumatut mobiilirakendust, mis järgib autori poolt välja toodud kontseptsiooni. Rakendus ei piira kasutaja seadistatavaid funktsioone, nagu kategooriad, sildid, ühendatud pankade arvu ja ühiseelarve liikmed. Lõpptoote üks peamisi omadusi on funktsionaalsus, mis võimaldab kasutajatel automaatselt vastu võtta pangatehinguid edasiseks käsitsi kategoriseerimiseks.

Praktiline osa on piiratud seadmetega, millel on iOS ja Android mobiiloperatsioonisüsteemid. Projekti põhieesmärk on pakkuda kasutajale mobiilset valmis lahendust ühise eelarve haldamiseks, kasutades selleks lõputöö raames välja töötatud kliendi ja serveri osad, mis vastavad autori nägemusele rakendusest, mis on loodud tuvastatud probleemide lahendamiseks. Rakenduse nimi on lühike ja meeldejääv lühend - MYBE, mis on moodustatud fraasist "Manage Your Budgets Easily". Lõputöö osana loodud rakendus ei tähenda ärilist kasutamist ilma oluliste muudatusteta, et täita seda tüüpi tarkvara kõiki nõudeid ja standardeid.

Lõplik rakendus koosneb kahest osast: serveriosa vastutab kasutajaandmete töötlemise, salvestamise ja nendega suhtlemise eest ning kliendiosa vastutab kasutajaandmete visuaalse esituse eest. Serveri ja kliendi osad suhtlevad turvalise HTTPS-protokolli kaudu. Kõiki kasutaja esitatud andmeid kasutatakse selle lõputöö raames rangelt. Samas toimub kasutajaandmete töötlemine kolmandate osapoolte teenuste poolt alles pärast kasutajapoolset nõusolekut ja seda reguleerivad vastavad järelevalveasutused.

See lõputöö on kirjutatud inglise keeles ja on 45 lehekülge pikk, sisaldades 10 peatükki, 26 joonist ja ühte tabelit.

List of abbreviations and terms

API	Application Programming Interface.
BIC	Bank Identifier Code.
CSS	Cascading Style Sheets.
ERD	Entity Relationship Diagram.
ETS	Erlang Term Storage.
FAQ	Frequently Asked Questions.
HTTPS	HyperText Transfer Protocol Secure (HTTPS) is the secure version of hypertext transfer protocol, the primary protocol used to send data between a web browser and a website. HTTPS is encrypted to increase the security of data transfer. [1]
ID	Identifier.
IT	Information Technology.
JDBC	Java Database Connectivity API.
JSON	JavaScript Object Notation. The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.
MVCC	Multi-Version Concurrency Control - is an advanced technique for improving database performance in a multi-user environment.
MVP	Minimal Viable Product.
MYBE	Manage Your Budgets Easily - the abstract name of the application created within the framework of this dissertation.

OAuth 2.0	It is an authorization protocol that allows one service (application) the right to access user resources on another service. The protocol eliminates the need to trust the username and password of the application and allows the service to issue a limited set of rights [23].
PSD2	The revised Payment Services Directive is a European regulation that requires banks to develop mechanisms to enable third-party providers to work securely, reliably, and rapidly with the bank's services and data on behalf and with the consent of their customers [27].
REST	REpresentational State Transfer - is a set of architectural principles attuned to the needs of lightweight web services and mobile applications [14].
RSA	Rivest–Shamir–Adleman is a public-key cryptosystem widely used for secure data transmission.
SQL	Structured Query Language.
UI	User Interface.
URI	A Uniform Resource Identifier is a character sequence that identifies a logical (abstract) or physical resource. A URI distinguishes one resource from another.
UTC	Coordinated Universal Time (Universal Time Coordinated).
VM	Virtual Machine.

Table of contents

List of figures

List of tables

1. Introduction

In today's world, many of us keep track of how and what we spend money on. It is an integral part of human nature to streamline life: whether it is counting steps on a walk, counting calories in dinner, or, for example, hours spent at the computer. Accounting for personal and family finances has actual and practical meaning: it helps to understand how much we spend and not to be left without funds by the time our bills and loans are paid.

People started keeping track of the budget a long time ago: at first, it was notebooks, however, sometime later, they were replaced by Excel and other analogues. Now, we can find many solutions for doing home accounting, including a wide variety of functionality differing by the automation level in the market. However, despite this, the modern market is still not able to offer a solution that would correspond to the author's vision of the application for tracking personal and/or family budgets.

As part of this dissertation, the author provides the vision of the concept of a budgeting application by compiling a list of requirements for an MVP, analyzing existing solutions, and speaking about the technologies chosen for implementing the application, complex aspects of implementation, and used solutions. The author completes the dissertation with a brief analysis of the final product and a description of the possible development of the application in the future.

1.1 Problem and aim

Today's market provides us with many programs designed to simplify home accounting. Unfortunately, the applications that exist currently, along with their advantages, have significant drawbacks. Most often, the disadvantages are the lack of automatic synchronization of user transactions made through electronic payments. Quite often, there is a lack of functionality that allows the customization of tools used for distributing the transactions, such as creating and/or changing spending categories and various tags. Some solutions provide the user with an obsolete and inconvenient interface, do not provide remote data synchronization, or do not allow joint accounting.

One way or another, the market has not yet provided a solution that would eliminate all of these shortcomings.
This thesis aims to create a prototype of a home accounting application that would have a simple and intuitive user interface, combine the functionality corresponding to the author's concept of a home budgeting program, and eliminate the most significant shortcomings of applications on the market today. One of the most challenging aspects of this thesis is a large amount of work that needs to be done in a relatively short period. Another goal of the project is to analyze existing programs and solutions for home accounting, then, based on the results obtained (and personal experience), formulate the application concept and implement both the server and client parts following the requirements.

1.2 Relevance

The primary task of the budget applications is an aggregation of all transactions from user bank accounts and manually entered income and expenses, as well as providing the ability to sort those using categories and automatic visual chart generation that allow users to identify the main categories of expenses and income.
The issue of accounting for family income and expenses became especially acute during the recent global crises. Accounting for users' income and expenses allows them save money significantly and simplifies the planning of future expenses.
Due to the digitalization of Estonian society, electronic payments have almost completely replaced physical currency. Nowadays, more people use more than one bank for day-to-day operations. The fact that each bank has own separate application creates difficulties in keeping records of the family budget. Most people who use two or more banks do not always know how much available funds they have at any moment, and this is where budgeting software comes in handy. Unfortunately, many existing applications have certain shortcomings, intended to be eliminated in the author's application created in this dissertation.

1.3 Methodology

In the theoretical part of the dissertation, the author analyzes existing applications. The analysis results and the author's personal experience are used to derive his concept of the application.

The author analyzed both the shortcomings and the advantages of the three existing solutions on the market and formulated the software requirements for the budget management application, which becomes the basis of the application's MVP, the creation of which is covered in more detail in the practical part of the dissertation.
The dissertation describes the key features of the home accounting application. The main aspects of the author's concept of the application for budgeting are formulated. The technologies used for the implementation of the practical part are also justified.
Finally, the application is implemented using the requirements in the practical part. Various difficulties that have arisen in the process of implementing various aspects of the application are described. The result of the dissertation's practical part is the application's server and client parts, implemented using the technologies chosen by the author. A description of the application's server and client workflow and features are provided, along with the source code of the various components included in the dissertation appendix.

2. Analysis of existing applications

An excellent starting point when creating software is to consider the analysis of solutions that already exist on the market. There are currently over 30 budget management applications available on the Google Play Market [39]. The results of the analysis of existing products are a good foundation for creating an application concept that would eliminate existing programs' main disadvantages and contain at least some of their main advantages. For analysis, the author defines the essential aspects for comparison and summarizes both advantages and disadvantages of each application.

2.1 Analysis scope

A comparison of the functionality of existing budgeting applications is made in two stages.

The first stage is the identification of key aspects to be assessed and the criteria for their assessment. As a method of evaluation, the author considers his personal experience in using these applications, the minimum period of one month in the last calendar year.

The second stage is the assessment of each application from the sample on a scale from 0 to 3, where:

0 (nothing to rate) - this functionality is not provided in the application.

1 (bad) - the functionality is present in a minimal form or does not perform its task properly.

2 (good) - the functionality performs its task, however, it has some restrictions on its use (for example, the maximum number of categories or the presence of prerequisites for fulfillment before use).

3 (excellent) - the functionality is presented in the application without any restrictions.

The results of the comparison are provided in table form.

2.1.1 Analysis criteria

As the main criteria for comparing the functionality, the author considers aspects that are most in demand for solving everyday budgeting tasks:

1. Automatic synchronization of bank transactions (**bank synchronization**)
2. Functionality that allows to create and edit user income/spending categories (**customizable categories**)

3. Possibility to manage the budget together with other users (**joint management**)
4. Remote data synchronization (**cloud synchronization** - user can access his data from different devices without the need for manual data transfer)
5. Automatic transaction amount conversion while using multiple currencies (**exchange rate conversion**)

2.2 Analysis

For comparison, the author selected three home budgeting programs, each of which is available for devices running on Android and iOS operating systems as of October 10, 2022.

2.2.1 Wallet [2]

This application is the leader among existing accounting programs. It provides connectivity to more than 15,000 banks worldwide and uses neural networks to categorize user's transactions automatically. The application has a pleasant and relatively intuitive user interface. It also offers additional useful features, such as spending planning or setting limits on spending categories. Also, Wallet has a desktop application.

Evaluation by criteria:

1. **Bank synchronization** - 2/3 (good)
   Although the application automatically synchronizes user transactions through connected banks, this functionality suffers from 2 minor limitations: automatic categorization works very mediocrely, forcing the user to check each time which category a particular transaction are assigned to since the algorithm very often determines it incorrectly. As a result, this leads to the fact that the user is still engaged in manual categorization. Sometimes the category is not recognized by the algorithm at all. The second disadvantage is that the application does not display booked bank transactions, without highlighting them in any way, as a result showing the user amount of the available balance that is higher than he has.
2. **Customizable categories** - 1/3 (bad)

Despite the acceptable number of built-in categories, the application does not allow users to create their own.

3. **Joint management** - 3/3 (excellent)

   The application provides functionality for joint budget management using a paid subscription.
4. **Cloud synchronization** - 3/3 (excellent)

   The application synchronizes user data through cloud storage providers.
5. **Exchange rate conversion** - 2/3 (good)

   The application supports the currency conversion function, however, it limits on the number of currency pairs - a maximum of 3.

Advantages:

- A large number of available banks to connect.
- A modern and clear user interface that supports the dark color UI theme is available in multiple languages.
- A large number of well-designed charts with visualization of user expenses and income.
- Many additional functionality - templates for creating transactions manually, the ability to track debts, set goals, and much more.
- A separate application for desktops with the same functionality.

Disadvantages:

- Almost all of the mentioned functionalities are only available with a paid subscription.
- Despite the overall good user interface, sometimes it looks overloaded and difficult to understand.
- Lack of error handling during synchronization with the bank. To eliminate errors, the user must connect to the bank again. The connection with Estonian banks is relatively unstable, and sometimes it takes 2 or 3 iterations to achieve the result.

2.2.2 Monefy [3]

The application, which is one of the most popular along with Wallet, offers the user a much smaller set of functions. Compared to the previous application, Monefy has a much less thoughtful and intuitive user interface, which may seem outdated.

Evaluation by criteria:

1. **Bank synchronization** - 0/3 (nothing to rate)
   Unfortunately, this application does not support automatic synchronization, and the users must enter transactions manually.
2. **Customizable categories** - 3/3 (excellent)
   The application allows users to create new and edit existing income and spending categories. This functionality is available with the paid subscription.
3. **Joint management** - 0/3 (nothing to rate)
   This application does not support joint budget management.
4. **Cloud synchronization** - 3/3 (excellent)
   The application allows users to synchronize their budget using remote storage and offers several options to choose from.
5. **Exchange rate conversion** - 3/3 (excellent)
   The application automatically converts transactions from other currencies to the main budget currency without restrictions if the user has a premium plan.

Advantages:

- Multilingual, not limited count of currency pairs, dark theme.
- Low price per subscription due to gift offers at the first launch, about 2.5 EUR per month at the time of writing the dissertation.

Disadvantages:

- Not always intuitive, obsolete user interface. Too many icons and a lack of text descriptions significantly affect usability.
- There is almost no visualization of user income and expenses statistics.
- The application is almost impossible to use without a paid subscription.

2.2.3 Bilance [4]

This application appeared on the market relatively recently, it is a young and actively developing Estonian startup. The main advantage of Bilance is the fully automatic categorization of user transactions and compatibility with all Estonian banks. Currently, the application is still actively developing and has yet to have time to acquire additional valuable features.

Evaluation by criteria:

1. **Bank synchronization** - 3/3 (excellent)
   This application automatically syncs and categorizes user transactions from over 2,000 European banks using premium Nordigen API solutions. Compared to Wallet, categorization is much more stable and categorizing works correctly for the biggest part of transactions. Moreover, the application accurately tracks the transactions booked by the bank and correctly displays the balance available to the user.
2. **Customizable categories** - 1/3 (bad)
   According to the author, one of the main shortcomings of the application is the inability to create new custom categories, which severely limits flexibility due to the small number of standard categories. Currently, the application only allows users to change existing categories, while the changes made will not affect automatic categorization in any way due to the tight binding of the application's categories with the categories provided by the Nordigen API.
3. **Joint management** - 0/3 (nothing to rate)
   This application does not support joint budget management.
4. **Cloud synchronization** - 3/3 (excellent)
   The application synchronizes user data using remote storage.
5. **Exchange rate conversion** - 3/3 (excellent)
   The application automatically converts transactions from other currencies to the main budget currency without restrictions.

Advantages:

- The application supports all banks available in Estonia, during the entire period of use, the author of the dissertation did not spot any problems during transaction synchronization or bank connection.
- The low subscription price, as of 10 Oct 2022 equals to 2 EUR per month.

Disadvantages:

- The application supports only English.
- The application cannot be used without a paid subscription.

- The user interface is not always intuitive, and the design feels a bit old.

2.3 Summary

The table below shows the final result of the application analysis. It is evident from Table 1, none of the applications fully meets the evaluation criteria chosen by the author:

Table 1: Applications analysis results

Application	Bank synchronization	Customizable categories	Joint management	Cloud synchronization	Exchange rate conversion	Average score
Wallet	2/3	1/3	3/3	3/3	2/3	2.2/3
Monefy	0/3	3/3	0/3	3/3	3/3	1.8/3
Bilance	3/3	1/3	0/3	3/3	3/3	2/3

3. Application concept and requirements for MVP

Before compiling a list of requirements for an application, the author outlines the fundamental principles for the proposed product.

3.1 The author's concept of budgeting application

In this chapter, the author briefly formulates his vision of the concept of an application for managing a personal or family budget:

The application should provide the user with the tools to solve his problems however, should not make decisions for the user.

An application is a set of tools for solving day-to-day tasks, which should make our life easier. However, the application should not make decisions for the user. Automatic categorization may be advisory, however, it should not be used without the user's consent.

Do not tie user data to his device.

Storing user data on their device may seem like a more secure solution, however, this is not always true and greatly affects usability if the user wants to access their data from another device. In addition, it cannot provide a user with joint budget management on multiple devices. The data should not be stored locally since this will also reduce the load from the user's device.

Give the user full control over their data.

The user should be able to change and delete all data, one way or another associated with him at any time and in the shortest possible time, especially when it comes to sensitive or personal information.

Keep the UI simple, the user's screen should not be overloaded with information.

The ease of use of the application and the time it takes the user to get used to the interface are directly affected by the amount of information presented on the screen. Too much information presented at the same time can scare or confuse the user, directly affecting the application's usability.

Do not limit users in the customization of tools.
The ability to customize distribution tools such as budgets, categories and tags significantly impacts the users and helps them handle their day-to-day routine.

Do not forget about secondary features.
When developing an application, it is challenging to consider all the nuances and lay the foundation in time for expanding the functionality. However, the foundation for functionality that would serve as a nice addition to the main tools of the application can be laid already at the initial stage. According to the author, this includes support of multiple user interface languages, a dark color theme, and the ability to synchronize with fiat banks and user wallets on crypto exchanges as cryptocurrency has become increasingly widespread and popular.

3.2 Application MVP requirements

Based on the data obtained during the analysis and the formulated principles, it is necessary to draw up the minimum requirements for the project. A clear definition of the minimum expected functionality gives an understanding of the technologies and solutions most appropriate and effective in creating the final product.

3.2.1 Base functionality

The practical part of the thesis should provide users with the minimum functionality for home budget accounting. The functionality is described from the point of view of the end user and is not divided into server and client parts:

1. Seamless process of account creation and authorization using a Google account.
2. Automatic detection of the user's language, time zone, and currency.
3. Creation, modification, and deletion of entities used to group transactions (internal usage: budgets).
4. Creation, modification, and deletion of entities used to categorize transactions (internal usage: categories and tags).
5. Viewing available information and connecting to the budget entities used to determine whether certain transactions belong to a particular banking institution or currency (internal usage: wallets).

6. Creation, modification, and deletion of entities containing transaction details, such as amount, currency, type (income, expense, transfer between accounts, correction), date of the transaction, category, list of tags, whether the transaction belongs to the budget and wallet, as well as any available transaction details.
7. Ability to analyze income and spending using app-generated pie charts with selectable display types (sorted by categories, tags, expenses, income, or wallets).
8. The dependence of the application on a stable Internet connection to avoid desynchronization and possible problems.
9. Support of Google Android and Apple iOS mobile platforms.
10. Three currencies support (Euro, United States Dollar, and Great Britain Pounds) for displaying and converting balances and transaction amounts.

The key features of the application (which combine the advantages of existing solutions on the market) are listed in the following chapters.

3.2.2 Automatic synchronization of bank transactions

In addition to the implementation of the main functionality, the author assumes that the problem of manual data filling will be solved by automating the process of obtaining user transactions through a banking data aggregator.

In the expected scenario, the user uses multiple banks (two or more) to make his daily payments. Accordingly, the user must grant all necessary permissions to the data aggregator to start using automatic bank data synchronization. While connecting the user's bank with the application, the user will be redirected to the website of a banking institution for subsequent authorization and data processing commitment. All redirects must be done using a secure data transfer protocol - HTTPS.

Automatic synchronization of transactions with the bank is carried out every time the user enters the application, after an hour of active use of the program, or by manually pressing the synchronization button.

The maximum number of requests for transaction synchronization can be limited by both the data aggregator and the banking institution itself. If the limit for data synchronization is reached, the user will be notified about this by an error pop-up notification.

The banking institution may also limit the amount of information available about a banking transaction. The list of mandatory information to provide includes the amount

of the transaction, the currency, and the date of the transaction (see chapter 5.2 of the Nordigen API for more details on the amount of provided data).
The application must support at least five largest banks in Estonia, including Swedbank, Revolut, SEB, LHV, and Luminor.

3.2.3 Customizable transaction categories and tags

To provide the most flexible personalization, the user should not be limited to a basic set of categories and tags. Users should be able to create, modify and delete both pre-generated and personally created categories and tags.
The list of possible changes includes category name, type (income only/expenses only/all), icon, icon color, and icon background color. A tag can be configured with both the scope of use (revenues only/expenses only/all) and a list of categories for which this tag will be available.
Storage and interaction with categories and tags should be carried out at two levels:

- On the user level - a list of user-created categories and tags, the removal of which will not affect their availability at the budget level. However, any changes to the category or tag at the user level will be reflected at the budget level.
- On the budget level - a list of categories and tags is used to categorize transactions within a given budget. Removing a category or tag at the budget level will not affect it at the user level, however, changes made by the user will be treated by the system as changes at the user level and will affect their display at the budget level.

3.2.4 Remote data storage

To store user data, remote data storage is used, which can only be accessed by the application’s backend server. Remote data storage is a more secure alternative to client-side data storage considering security requirements since the level of digital hygiene differs from user to user and cannot guarantee secure storage.
Another advantage of remote storage is the ability for a user to access his data from several devices. By logging in, the user can access the same content on different platforms.

3.2.5 Joint budget managing

The application should provide functionality for joint budget management. To do this, it is necessary to implement the functionality for inviting other users within a single budget, configure their roles, and, if necessary, revoke access granted earlier.

The author assumes that in most cases, users invite an average of one participant to access a joint budget, however, the maximum number of invitations issued should not have any restrictions.

Searching for users should be done by entering another user's email address. Next, the system will try to find a record in the database with the specified email address and inform the user if such an email is not in the system.

The roles that reflect the level of user access must provide the right to perform a certain list of operations, presented below:

1. Administrator - has a full set of budget management features: inviting and deleting users within his budget, changing their roles, connecting and disconnecting previously created or connected wallets, categories and tags. Right to create, modify, categorize and delete transactions.
2. Editor - functionality is limited by the ability to categorize transactions and connect wallets, categories, and tags.
3. Read-only - a user with this role does not have the right to perform any actions however, it can view the information available within a specific budget without any restrictions.

3.2.6 Multiple currencies support within one budget

One of the important requirements for most budgeting applications is multicurrency support and automatic currency conversion. With the current pace of globalization in our world, more and more people open accounts in different currencies, creating a need for correctly displaying information in budgeting applications. Although this statement mainly applies to cryptocurrencies, multicurrency support is becoming an increasingly popular functionality in the modern world.

The user should be able to set the default currency to display all balances and transaction amounts within a specific budget. The server part of the application must correctly and timely update the exchange rates and convert all the necessary balances and amounts.

Currency pair rates should not be received later than one day before the amount is converted. Within the framework of this dissertation, the author has limited the range of supported currencies to the three most popular and stable currencies - the Euro, the United States Dollar, and the Great Britain Pound.

3.2.7 Other requirements

In addition to the listed functionality, the application must meet the following requirements:

1. Any user, banking, or other sensitive data must be stored in encrypted form, the encryption key must be changed at least once a month to avoid major data leaks when third parties hypothetically receive access to the database. Data encryption is discussed in chapter 6.2.1 Data encryption.
2. The user should be able at any time to delete all data associated with him and his account as soon as possible.
3. In case of a user access token compromise, the compromised token must be blacklisted, and the session immediately terminated. The token is considered compromised if the initiated requests to the server do not match the user's access rights or action scope, which can be performed through the application. This concept also includes an attempt to receive data from endpoints that are not directly used by the application.
4. The client part of the application must provide the ability to change the application language. Within the framework of this dissertation, it is enough to implement English language support.
5. The client part of the application should provide the ability to select a color theme from two options: dark (for use in low-light conditions) and light (standard). Within the scope of this thesis, only the light theme is implemented, however, the necessary functionality for further expansion are laid down at the architectural level.

4. Technology stack

The modern IT industry offers developers many different tools and technologies for solving problems of absolutely any type. In projects like this dissertation, well-chosen technologies can critically impact performance and scalability. The description of the stack of selected technologies is divided into two parts due to fundamental conceptual differences - server (backend) and client (frontend).

4.1 Backend technologies

Back-end development technologies play a paramount role in the development of any software product. Choosing the right technology for back-end development can make development both easier and faster. Indeed, well-chosen technologies for writing a backend can guarantee good scalability and speed.

Within the framework of this thesis, the author, first of all, chooses in favor of the technologies which he is most proficient with to achieve the greatest productivity. One of the main favorites for the role of the language for the backend is a fairly new, however extremely promising functional programming language - Elixir.

4.1.1 Elixir

Elixir is a dynamic functional language that is great for building highly scalable applications. One of the main advantages that influenced the choice of this language for writing the application server part is the ability to create distributed and fault-tolerant systems [5].

Elixir is chosen as the language for writing the backend due to its main features that allow developers to create reliable and high-performance applications.

Among its positive properties, the author would especially like to highlight the following advantages, which influenced the final choice of the language:

1. Stability - Elixir is currently one of the most fault-tolerant systems. Thanks to an internal system of supervisors who, in the event of a crash in one of the isolated processes, come to the rescue and help describe how to take actions that will ensure a full recovery. Thanks to the united group of supervisors, which form the so-called "supervision trees", Elixir allows you to build an uninterrupted and fault-tolerant architecture, which is extremely important for any product [6].

2. Data immutability - the concept of data immutability follows the rule: once data has been created in a memory location, it cannot be changed and must be preserved. The only disadvantage of this approach is leveled due to a significant reduction in the cost of memory in our time [7], which allows you to enjoy a wide range of advantages - thread safety, a constant state of an object, better encapsulation, more transparent and readable code [8].
3. Scalability - today, scalability is a critical requirement when building products. Although scalability is not a design requirement for this dissertation, the author has strived to create an application that meets this requirement as closely as possible. With parallel processes, supervised trees, and excellent fault tolerance, Elixir makes it easier to scale services than in many object-oriented languages [9]. Also, if we look at the practice of using Elixir by large services (like Discord [10] or Pinterest [11]) as the main backend language for working with millions of users, we can conclude that this language, like no other, allows you to seriously scale the services created on it.
4. Built-in caching solutions - Elixir has four different standard methods for temporarily storing and accessing data [12], which eliminates the need to add another dependency to the application (author means third-party data caching solutions such as Redis), making it more difficult to maintain.
5. Pattern matching - is one of the key features when writing code in Elixir. It is difficult to fully emphasize the power of this feature. With pattern matching, a programmer can use different signatures to describe how the code should behave in different cases. This helps to write code more concisely and cleanly. Figure 1 shows an example of pattern matching in Elixir.

```
{:status_atom, %YourStruct{param: value}} = your_function(arg)
```

Figure 1: Pattern matching in Elixir.

4.1.2 Phoenix Framework

According to the survey.stackoverflow.co [13], Phoenix is the most loved web framework among developers in May 2022. Phoenix is a web development framework written in Elixir and widely used for REST API implementation. Phoenix combines high developer productivity with high application performance and has many advantages, including reactiveness, a perfect balance between abstraction and explicitness, an easy router mechanism, various testing tools, and good documentation [15]. The reason why the author made his choice in favor of this framework, in addition to its features, is its great popularity and the large number of training materials associated with it. Moreover, the author of the dissertation already had some experience with this framework, which became the decisive argument in favor of this technology.

4.1.3 PostgreSQL

The choice of PostgreSQL for the implementation of the database is facilitated by many different advantages that make this solution the most promising:

1. PostgreSQL manages concurrent read/write efficiently using multi-version concurrency control (MVCC). This means that any read-in-progress does not block writes and vice versa [16].
2. With pgSQL, a developer can group a calculation block and a series of queries inside the database server, thus significantly saving server resources for communication between the client and the server [17].
3. It is a cross-platform solution that makes it easy to host and set up a database on a Linux or Windows server in the future [18].
4. PostgreSQL is an open-source solution that allows it to be used in projects of any type. Moreover, it provides good scalability for any projects in the future [18].
5. This solution is an extensible database, which means that in case of a lack of any functions, the developer can write them himself or install a third-party extension. This feature also has a positive effect on scalability [19].

Thanks to its strong community support and many advantages, this open-source relational database is an excellent fit for the project created within this dissertation.

4.2 Frontend technologies

The author primarily considered solutions focused on multi-platform development when choosing technologies for the client (front-end) part of the application. Even though that native application development significantly outperforms cross-platform application development in terms of performance and the ability to use the full functionality of the user device, at the same time, it requires much more development time and reduces code reusability.

4.2.1 JavaScript

One of the most important advantages of this programming language is a large number of different cross-platform frameworks for developing mobile applications.
Cross-platform development significantly reduces development time, which, within the framework of this thesis, is one of the key factors in choosing a technology for the implementation of the client side.
A large number of well-documented, tested frameworks and various libraries for any need make this language a favorite tool for mobile front-end development.

4.2.2 TypeScript

TypeScript is a strongly typed, object-oriented, compiled language. TypeScript is both a language and a set of tools. It is a typed superset of JavaScript compiled into JavaScript. TypeScript supports JavaScript libraries, and compiled TypeScript can be used by JavaScript code. TypeScript-generated JavaScript can be reused by all the existing JavaScript frameworks, tools, and libraries.

4.2.3 React Native framework

React Native is a framework for building native rendering mobile apps using ReactJS, a JavaScript code library developed and maintained by Meta Platforms (formerly Facebook) [20].
The key features for which this framework is chosen as the primary tool for implementing the front-end part of the application include the following:

1. Using JavaScript makes it possible to develop a universal application using this framework without diving into the ecosystem and language features of each operating system [21].
2. React Native significantly speeds up the development process. Initially, this saves developers from recompiling with every change since the application is immediately reloaded during the development phase [21].
3. A dynamic toolbox with many productivity-enhancing features, such as integrated components providing built-in solutions for the most common tasks.

These features not only greatly increase the speed of product development however, also provide a pleasant and productive development experience [21].

4. React Native uses the Flow framework by default. This framework promotes static typing in JavaScript however, has fewer commitments than TypeScript since it can be used within default .js files. While React Native is built in Flow, it supports both TypeScript and Flow by default [38].

Summing up, it is worth mentioning that React Native is a powerful tool that allows the developer to create a mobile application in a short time without the need to dive into the ecosystem of each platform.

All the existing shortcomings of the cross-platform development approach, such as the inability to use all available functionality of each platform, lower performance, bigger size of the final application, debugging difficulties, and other less significant shortcomings [21] are the cornerstones of the chosen approach however, are not sufficient within the framework of this dissertation and could not be a reason for considering other tools and solutions due to the lack of time to solve such a large amount of tasks.

5. Third-party services

Nowadays, various third-party services can greatly simplify and speed up application development, providing various services ranging from simple authentication to processing huge amounts of data using machine learning technologies or online payment solutions. Most modern applications use third-party services in one way or another because they can significantly reduce both the overhead costs for the development and maintenance of a particular functionality and provide a much higher quality of services due to many years of development, a large array of accumulated data, more advanced architecture, and so on.

However, it is crucial to choose third-party services, especially when it comes to the user data, because the service provider's reliability can affect the security of client data, which can cause reputational or other losses in case of third-party service data leakage.

As part of this dissertation, the author intends to use third-party services for user authentication, application image assets storing, obtaining up-to-date exchange rate data, and obtaining user banking data.

5.1 Google Account API

The Google Account APIs use the OAuth 2.0 protocol for authentication and authorization. Google also supports common OAuth 2.0 scenarios for the web server and client side [22].

For the least resource-intensive solution for user authorization, it is decided to use a Google Account service that uses the OAuth 2.0 protocol. By providing single sign-on functionality, the user can use their existing credentials to open or create an account in the application. This convenient way allows developers to significantly simplify the implementation of authorization and speed up user registration/login to the application without having to spend time creating an account with another username and password.

After clicking on the register button (see Figure 2), the user will be redirected to the Google service authorization page, after which a pop-up window will request permission to transfer information about the user's Google account to the application (see Figure 3).

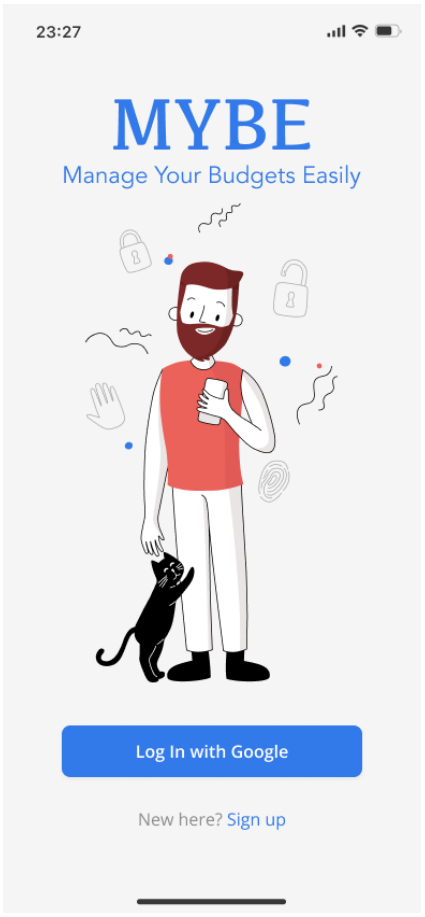
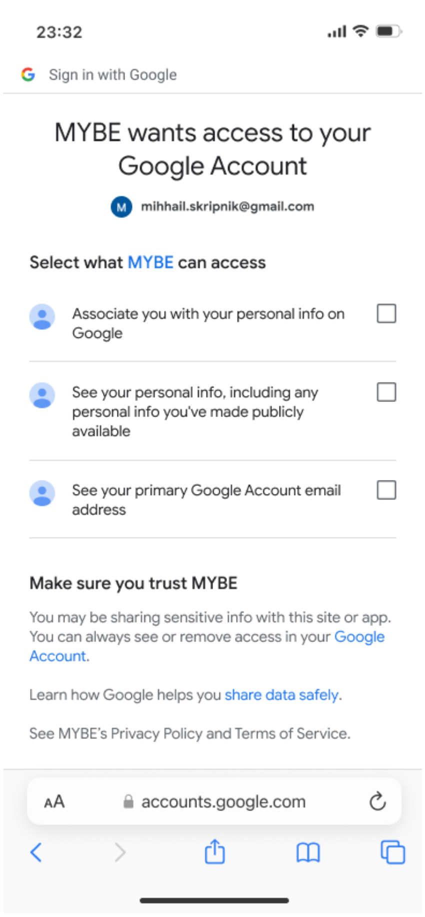


Figure 2: MYBE login screen.


23:32

Sign in with Google

MYBE wants access to your Google Account

mihhail.skripnik@gmail.com

Select what MYBE can access

Associate you with your personal info on Google ☐

See your personal info, including any personal info you've made publicly available ☐

See your primary Google Account email address ☐

Make sure you trust MYBE

You may be sharing sensitive info with this site or app. You can always see or remove access in your Google Account.

Learn how Google helps you share data safely.

See MYBE's Privacy Policy and Terms of Service.

AA accounts.google.com

Figure 3: Sign in with Google screen.

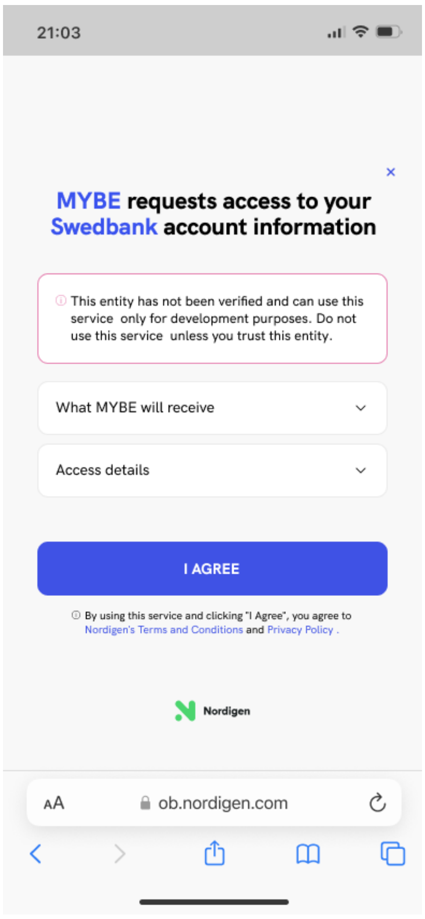
After the user is allowed to share his information with a third-party site, he will be redirected to a specially created deep link, where the user will be redirected back to the application.

After going through these quick steps, the user has successfully logged in with Google, and the application has access to the user's profile information that can be used to create an account or log in.

5.2 Nordigen API

One of the key services required for the application implementation is the aggregator of user banking data. In this dissertation, the primary criterion for choosing a data aggregator is the price of using such a service. Currently, the only free solution is the Nordigen API, which, despite this, even outperforms its competitors in some aspects - the paid aggregation platforms Tink [24], Plaid [25], and Truelayer [26].

This data aggregator is used within the framework of this dissertation to obtain user bank accounts (as well as their balances) and transactions through an open API that operates according to the PSD2 directive. Banking institutions transfer user data for processing only if this permission is issued by the user on the banking portal page, where the user will be redirected (see Figure 4) upon initial conclusion of an agreement to provide data to both the aggregator (represented by Nordigen) and a third party (in this case, the author's application - MYBE).


21:03

MYBE requests access to your Swedbank account information

This entity has not been verified and can use this service only for development purposes. Do not use this service unless you trust this entity.

By using this service and clicking "I Agree", you agree to Nordigen's Terms and Conditions and Privacy Policy .

Figure 4: Nordigen data processing request.

Within the framework of this dissertation, any information and/or user data can be used only to provide budgeting services directly related to the user account.

This API, due to the specifics of its application, has many different restrictions, primarily imposed by the banks themselves, among such restrictions are response formats that differ from bank to bank, restrictions on the number of synchronizations per day, and restrictions on the number of calls to the banking institution API. Despite the technical complexity, all the listed limitations should be considered when writing the server part of the application.

Also, this service tries in every possible way to facilitate the implementation of the solution for unifying the provided banking data and offers detailed information [28] on each of the supported banks and the amount of information provided, which significantly simplifies development. Based on the author's personal experience, Nordigen provides high-quality and prompt technical support to help with the solution for both technical and legal issues.

5.3 exchangerate.host API

The last of the third-party services used is exchangerate.host, which provides current and historical rates for fiat and cryptocurrencies. When choosing a service for receiving exchange rates, the author also primarily relied on the pricing policy of the service, as a result of which the choice is made in favor of this solution. Moreover, it is worth noting that in most cases, all similar services offer approximately the same functionality, which is why the author considered it irrational to check other paid alternatives.

All API requests are subject to rate limits. Real-time rate limit usage statistics are described in headers that are included with most API responses once enough calls have been made to a service endpoint [29].

As part of this dissertation, the application supports three major currencies - Euro, the United States Dollar, and Great Britain Pound with currency pair rates updated every hour to keep the displayed balances and amounts up to date.

According to the service website, currency data delivered are sourced from financial data providers and banks, including the European Central Bank [29].

One of the main advantages of this service is the almost constant uptime [29], a large range of supported fiat currencies and cryptocurrencies free of charge, and the absence of strict limits on the number of requests.

5.4 Google Cloud Storage

To store various visual assets, such as logos, icons in vector format, illustrations, and other design elements used in the client side of the application, it is decided to use Google Cloud Storage. The author of this thesis already has a paid subscription, which significantly saved time from looking for free visual assets hosting.

6. Application backend implementation

This chapter describes the approaches used to write the application backend, the structure of the project, encountered and/or potential challenges, as well as their solutions. The main server-side language of the application is Elixir, with the Phoenix framework for REST API implementation and the Ecto library for working with the PostgreSQL 14.5 database.

6.1 Architecture and project structure

The server part is divided into three layers - presentation, business, and data layer. Each of the layers performs strictly defined functions, described below:

1. Presentation layer - this layer is responsible for translating the data received in the request into the appropriate format for further processing at the next level. Since Elixir doesn't have objects, we can conduct the presentation layer operating with data transfer structs.
2. Business logic layer - this layer contains all the logic of the server application, represented by various services. It is responsible for interacting with all third-party services, processing, creating, modifying, caching, and validating user data.
3. Data layer - this layer is responsible for storing entities in the database. The data layer works with the transformation of data transfer structs into database entities and directly with the execution of SQL queries.

The general schematic separation of the server part into three layers is shown in Appendix 3, which provides a complete architectural diagram of the entire project. The chosen approach, dividing applications into three layers, is also reflected in the application structure itself, shown in Figure 5. The file structure has also been greatly simplified on the schematic image of the project structure to make visualization clearer and more understandable.

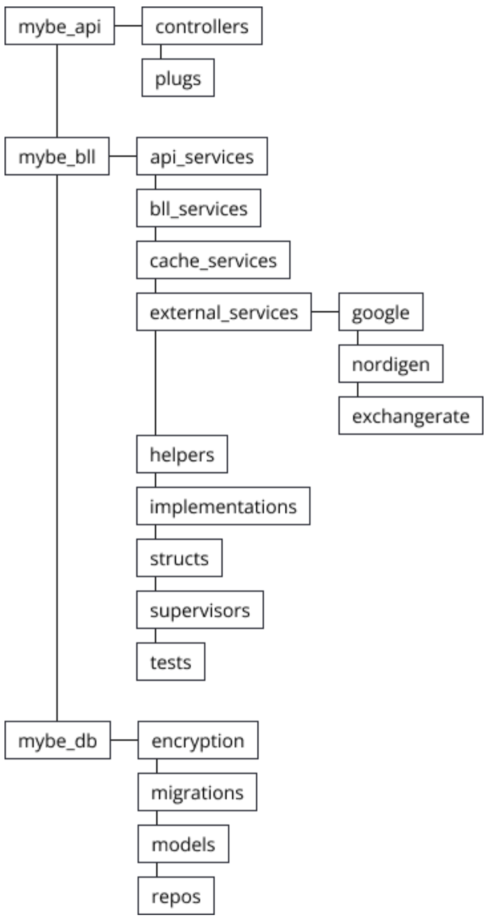


Figure 5: Backend project structure.

Package *mybe_api* contains two directories - controllers and plugs. The folder with controllers contains all the controllers used in the API, each of which is responsible for its endpoint namespace (the list of API endpoints is presented in Appendix 4). The plugs folder contains modules, which are a set of functions that process an incoming request before passing it directly to the handler controller. An example of use is a custom token validation.

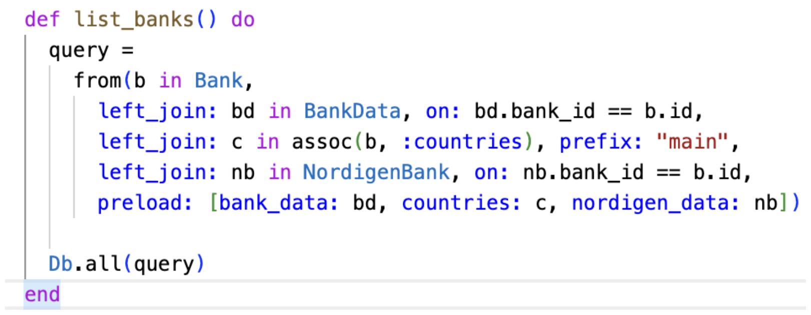
Package *mybe_bll* contains four directories with business logic services implementation, the *helpers* directory (contains a set of macros to simplify code writing), the implementations directory (contains a set of author's protocol implementations for working with some data types and serialization), the structs directory (data transfer structs), supervisors (containing supervisors for groups of services, united by the so-called “global” supervisors that create supervision trees) and the tests directory for testing specific cases or entire services.

Package *mybe_db* is represented by four directories - encryption (where modules for data encryption and decryption are stored), migrations (where all migrations performed with the database are stored), models (describe database entities and algorithms for converting them from/to data transfer structs) and *repos*, which contain repositories responsible for executing SQL queries inside the database.

6.2 Database

To work with the database, the author used one of the most popular libraries - Ecto. This library contains four main modules used for working with the database:

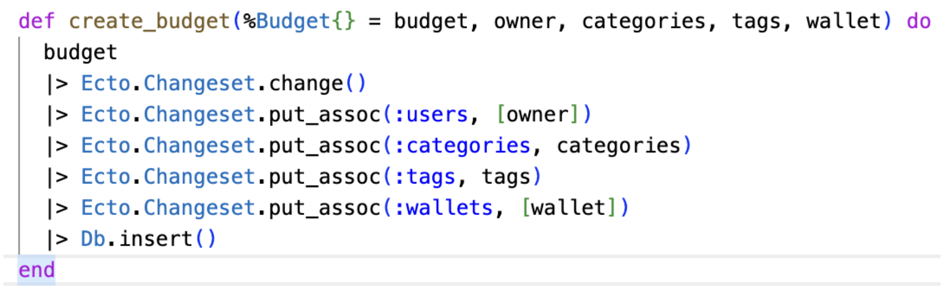
1. Ecto.Repo - repository modules are wrappers around the data store. Interaction with repositories allows developers to create, update, delete and query entities stored in the database. To set up work with PostgreSQL, this library requires an appropriate driver, or, as they are called in Elixir, an adapter. Credentials for connecting to the database are stored in the project's configuration file.
2. Ecto.Schema - a module that converts entities stored in the database into Elixir structures.
3. Ecto.Query - written using the Elixir syntax, a simple, understandable, and extremely powerful module that allows developers to get the information using repositories. It is also worth mentioning that queries written using this module are protected from SQL injections by default. Figure 6 shows examples of using this module in a project.



```
def list_banks() do
  query =
    from(b in Bank,
      left_join: bd in BankData, on: bd.bank_id == b.id,
      left_join: c in assoc(b, :countries), prefix: "main",
      left_join: nb in NordigenBank, on: nb.bank_id == b.id,
      preload: [bank_data: bd, countries: c, nordigen_data: nb])

  Db.all(query)
end
```

Figure 6: Querying list of available banks using Ecto.Query module.

4. Ecto.Changeset - so-called changeset allows filtering, casting, validation, and definition of constraints when manipulating structs. An example of usage within this application is shown in Figure 7.



```
def create_budget(%Budget{} = budget, owner, categories, tags, wallet) do
  budget
  |> Ecto.Changeset.change()
  |> Ecto.Changeset.put_assoc(:users, [owner])
  |> Ecto.Changeset.put_assoc(:categories, categories)
  |> Ecto.Changeset.put_assoc(:tags, tags)
  |> Ecto.Changeset.put_assoc(:wallets, [wallet])
  |> Db.insert()
end
```

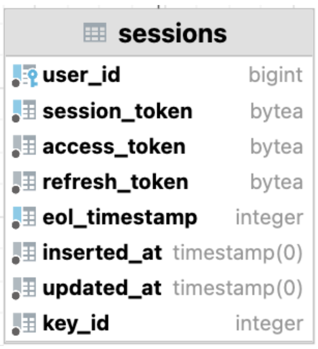
Figure 7: An example of using the Esto.Changeset module.

Working with the database using the Ecto library is greatly simplified, allowing a developer to write easy-to-read and understandable code without the need to interact with SQL directly.

In order to ensure the security of user data, all fields containing sensitive data about the user (transaction balances, account numbers, personal emails, access tokens, etc.) are encrypted when writing and decrypted when reading from the database. This aspect is discussed in more detail in the next paragraph.

The application database consists of 26 tables (including eight many-to-many tables), the most basic ones are listed below:

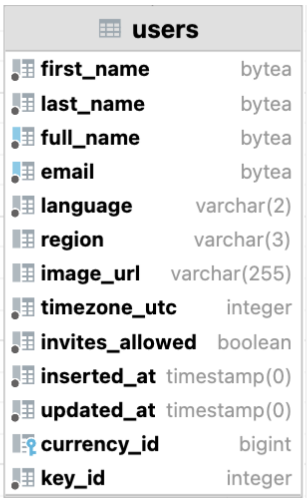
1. Sessions - used to store user sessions, access tokens, and refresh tokens.



sessions	
user_id	bigint
session_token	bytea
access_token	bytea
refresh_token	bytea
eol_timestamp	integer
inserted_at	timestamp(0)
updated_at	timestamp(0)
key_id	integer

Figure 8: Database “sessions” table.

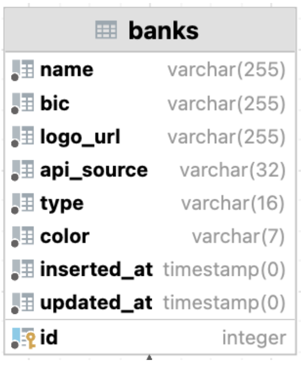
2. Users - stores information about the user, including first name, last name, full name, personal email, device language, time zone, and default currency.



users	
first_name	bytea
last_name	bytea
full_name	bytea
email	bytea
language	varchar(2)
region	varchar(3)
image_url	varchar(255)
timezone_utc	integer
invites_allowed	boolean
inserted_at	timestamp(0)
updated_at	timestamp(0)
currency_id	bigint
key_id	integer

Figure 9: Database “users” table.

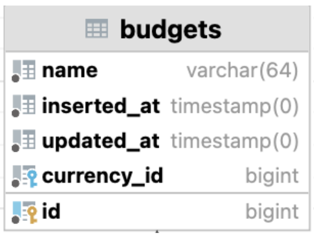
3. Banks - contains information about a banking institution, including the type of institution (can be used to expand further the functionality for working with crypto exchanges) used to obtain API data, a link to the logo, and BIC.



banks	
name	varchar(255)
bic	varchar(255)
logo_url	varchar(255)
api_source	varchar(32)
type	varchar(16)
color	varchar(7)
inserted_at	timestamp(0)
updated_at	timestamp(0)
id	integer

Figure 10: Database “banks” table.

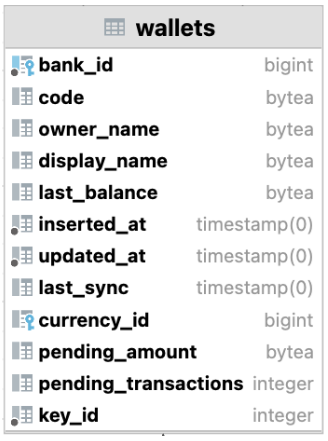
4. Budgets - a table for storing user budgets contains the name of the budget and its currency.



budgets	
name	varchar(64)
inserted_at	timestamp(0)
updated_at	timestamp(0)
currency_id	bigint
id	bigint

Figure 11: Database “budgets” table.

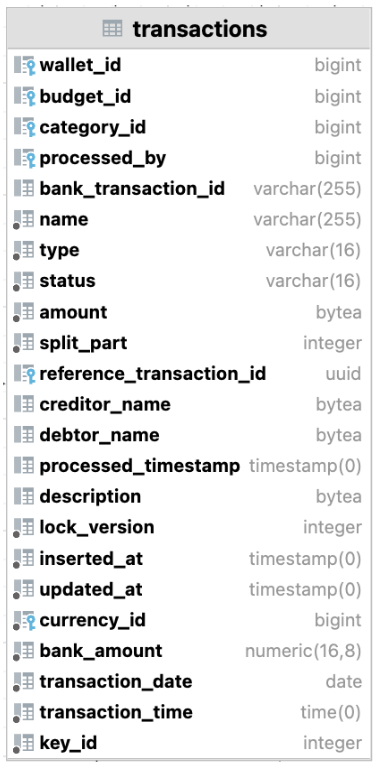
5. Wallets - stores all available and necessary information about the user's bank account, including the number of booked transactions and their amount, for the correct display of the available balance in the application.



wallets	
bank_id	bigint
code	bytea
owner_name	bytea
display_name	bytea
last_balance	bytea
inserted_at	timestamp(0)
updated_at	timestamp(0)
last_sync	timestamp(0)
currency_id	bigint
pending_amount	bytea
pending_transactions	integer
key_id	integer

Figure 12: Database “wallets” table.

6. Transactions - the main and the biggest table contains information about each entity associated with a real user transaction in his bank. This table contains a lot of fields with various metadata required to implement the internal logic for correct interaction with the Nordigen aggregator.



transactions	
wallet_id	bigint
budget_id	bigint
category_id	bigint
processed_by	bigint
bank_transaction_id	varchar(255)
name	varchar(255)
type	varchar(16)
status	varchar(16)
amount	bytea
split_part	integer
reference_transaction_id	uuid
creditor_name	bytea
debtor_name	bytea
processed_timestamp	timestamp(0)
description	bytea
lock_version	integer
inserted_at	timestamp(0)
updated_at	timestamp(0)
currency_id	bigint
bank_amount	numeric(16,8)
transaction_date	date
transaction_time	time(0)
key_id	integer

Figure 13: Database “transactions” table.

The complete database schema is provided in Appendix 2, the image is made using the built-in visualization tools of the DataGrip program from JetBrains due to its compactness compared to the ERD schema.

6.2.1 Data encryption

Since the application deals with user personal and banking data, there are a set of security requirements that, one way or another, should be considered. The author, not being an expert in the cybersecurity domain, cannot guarantee full compliance with all requirements regarding the storage and processing of data, therefore, all found or potential vulnerabilities should be considered acceptable within the framework of this dissertation.

Since in our time, there is a whole criminal industry [30] that targets personal client data, it cannot be guaranteed that unencrypted data is stored safely.

To encrypt user data, two different approaches are used, depending on the type of data to be encrypted:

1. For sensitive data such as bank account codes, wallet balances, transaction amounts, access tokens, and usernames, the Advanced Encryption Standard (AES) is used. This algorithm uses Galois/Counter Mode for symmetric key cryptographic block ciphers, which is often recommended as a more efficient encryption algorithm compared to RSA [31, 32]. To implement this encryption method, the author used a native Rlang cryptographic library with 256-bit keys.
2. For data not intended for direct retrieval (user email, access token, session token, and refresh token), the author used the Argon2 key derivation hashing function (this algorithm is the winner of the Password Hashing Competition [33]). This approach guarantees if third parties access the database, access tokens cannot be compromised. Hashing function is used from the argon2_elixir library.

In addition to encryption, the author also implemented encryption key rotation. Every 24 hours, the algorithm generates and adds a new key to the secure encryption key store. This practice significantly limits the amount of data that a potential attacker can decrypt if the database has been compromised in any way [34].

The implementation of the encryption and decryption algorithm are performed following the manual “Data encryption in a Phoenix (Elixir) App using Ecto Types” [35], the final source code is provided in Appendix 5.

6.3 Peculiarities of the business logic implementation, encountered problems and their solutions

During the server part of the business logic implementation, the author faced different peculiarities of banking data processing, various service restrictions, controversial points, and other challenges, described in this chapter.

6.3.1 Repetitive code

To reduce the amount of repetitive code, the author has implemented a separate service for simple operations such as adding, updating, deleting and querying entities. The code for this service can be found in Appendix 6. Although this solution may complicate the debugging of errors that occur, nevertheless, since it is used for the simplest operations, this significantly increases the readability of the code for individual services and methods that do not contain any complex logic.

6.3.2 Third-party services error handling

The server part of the application for its work actively uses third-party services with certain limitations. The most suitable example is the Nordigen API, which has more than eight error variants when requesting user transactions. To solve this problem, the author has implemented a separate module for each third-party service, which deals with error processing, logging (if required), taking the necessary measures, and compiling error messages for the user. The code for one of these services is presented in Appendix 7.

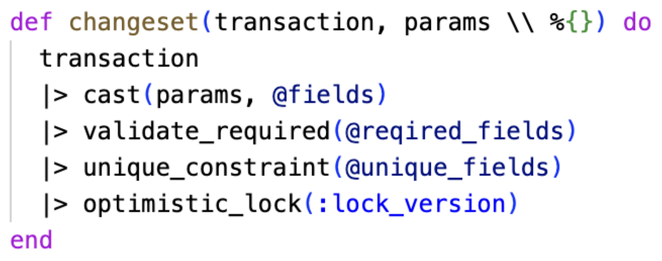
6.3.3 Race conditions

Since the application, among its advantages, offers user functionality for joint budget management, it becomes necessary to consider the scenario when the same transaction is processed (processing means defining its category, adding tags, and so on) by users almost simultaneously. As a result of this scenario, a so-called “race condition” may occur due to multi-threaded request processing. Despite the unlikely occurrence of this scenario, the author wished to consider this and implement the solution.

The least resource-intensive solution is chosen to solve this problem - so-called "optimistic blocking". This solution is most often used in systems with low data contention. Since, in theory, a race condition and subsequent undesirable consequences can occur during the distribution of transactions, it is decided to use "optimistic locking" only for this table.

The principle of operation of this solution is as simple as possible - a counter of "versions" is stored for each entry. This counter is incremented each time a record is modified. The library for working with the Ecto database already contains a method for working with optimistic locking, including a macro function for the database entity model that points to the "version counter". Ecto will throw an exception and rollback changes if a record which is previously retrieved is updated and the same record is modified at the time it is retrieved.

An example of using “optimistic locking” is shown in Figure 14.



```
def changeset(transaction, params \\ %{}) do
  transaction
  |> cast(params, @fields)
  |> validate_required(@reqired_fields)
  |> unique_constraint(@unique_fields)
  |> optimistic_lock(:lock_version)
end
```

Figure 14: Setting up transaction changeset using built-in Ecto “optimistic lock” function.

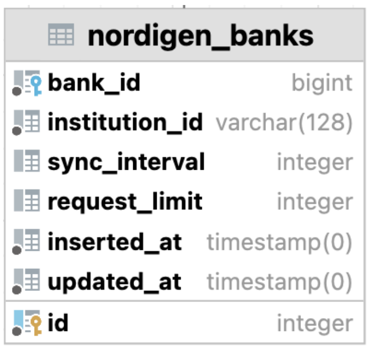
6.3.4 Nordigen service restrictions

One of the drawbacks of Nordigen API is the need for more transparent information about the limits on requesting user information. According to Nordigen's official website, the service guarantees at least four daily requests for any client data (account details or list of transactions) [36].

To consider this feature of the service, the author implemented a cache service to store information about the number of requests for user data for each bank on the current day. This service is implemented using the built-in caching mechanism - ETS. More details about this mechanism can be found in chapter 6.3.7 Data caching.

Data on the number of requests is stored in random-access memory in the following format: user, requisition_id (identifier used by Nordigen for the user-bank connection), the number of requests, and the time stamp of the last change. Every day at 00:00 UTC+0, the system automatically sets the number of requests to 0 and removes records that are requested more than a week ago.

In addition, the database for each bank (see Figure 15) stores a list of available data and limit values (determined empirically during Nordigen integration testing). As the result, the system can track the number of available requests and determine when the user has reached the limit by notifying him with the appropriate message.

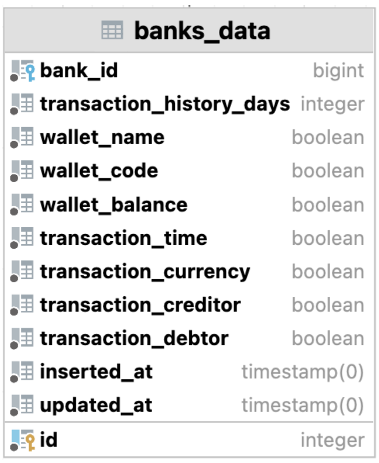


nordigen_banks	
bank_id	bigint
institution_id	varchar(128)
sync_interval	integer
request_limit	integer
inserted_at	timestamp(0)
updated_at	timestamp(0)
id	integer

Figure 15: A table that stores request limits for banks accessed through the Nordigen API.

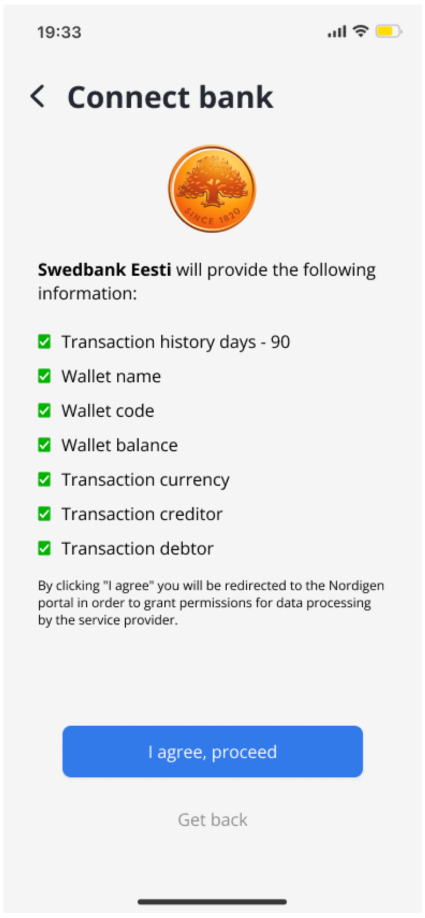
6.3.5 Different bank data formats

As mentioned earlier in chapter 5.2 Nordigen API, this service cannot guarantee the provision of the same amount of data for all banks. This leads to a situation where user account or transaction data can look completely different, as banking institutions provide a limited amount of data at their discretion. To solve this problem, the application database stores a table containing information about data scopes which will be provided by banking institutions (see Figure 16). Also, during the user's first synchronization with the bank, he will be redirected to a screen with information about bank data that can be retrieved and displayed by the application (see Figure 17).



banks_data	
bank_id	bigint
transaction_history_days	integer
wallet_name	boolean
wallet_code	boolean
wallet_balance	boolean
transaction_time	boolean
transaction_currency	boolean
transaction_creditor	boolean
transaction_debtor	boolean
inserted_at	timestamp(0)
updated_at	timestamp(0)
id	integer

Figure 16: A table with the information about data fields that the bank provides via Nordigen.



19:33

Connect bank

Swedbank Eesti will provide the following information:

- ✓ Transaction history days - 90
- ✓ Wallet name
- ✓ Wallet code
- ✓ Wallet balance
- ✓ Transaction currency
- ✓ Transaction creditor
- ✓ Transaction debtor

By clicking "I agree" you will be redirected to the Nordigen portal in order to grant permissions for data processing by the service provider.

Figure 17: A screen displays the information stored in the table from Figure 16.

6.3.6 Parallel wallet use by multiple budgets

Another interesting scenario that required a pragmatic solution is the parallel use of the same bank account in two or more budgets. By default, the system updates all wallets attached to the budget when the user enters the application if the last update is made more than an hour ago.

Before the discovery of this scenario, the user, after updating the list of transactions on budget A, did not receive new transactions for budget B since each transaction is tied to a specific budget (see Figure 13). This happens although the user worked with the same wallet on both budgets. As a result, this leads to a bad user experience and the need to synchronize the transaction list again, increasing the number of requests to Nordigen and lowering the request quota for a given user.

After implementing changes in the transaction synchronization algorithm, this problem is fixed. Now, after transaction synchronization, all budgets that are tied to the previously updated wallet will receive new transactions. Although this method multiplies the number of insert operations into the database (for example, with ten new transactions and two budgets, 20 entities will be added to the database instead of 10), this improves user interaction with the application and eliminates the need to make unnecessary requests, spending the bank request quota.

6.3.7 Data caching

Data caching allows an application to increase performance by using previously stored data. Using cached data API can serve requests many times faster. There are datasets in the application database that change infrequently, such as tables with countries and banks, as well as sets of standard categories, tags (these records have a true "default" parameter), and exchange rates that are updated once an hour. All this data in the process of using the application is requested much more often than it is updated, and therefore it makes sense to cache it to increase the API performance.

As a caching tool, the choice is made in favor of ETS - Erlang Term Storage, a part of the Erlang VM, the use of which eliminates the need to add extra dependencies. According to Elvio Viçosa Jr in the article "Caching with Elixir and ETS", the data structures used to implement the ETS tables are optimized to provide the best possible access time [37]. One of the simplest, however, efficient examples of using ETS for cache service implementation can be found in Appendix 8, while the cache service for the database contains more logic for periodic cache refreshing and various interfaces for obtaining data, as well as initiating a force cache update.

6.3.8 Testing

To test the server part, due to time constraints, the author used only unit tests for the most critical methods and modules (Appendix 9 provides unit tests for the session service). The most outstanding contribution to identifying bugs and errors is made by direct testing of the application through the front end. During business logic implementation, the author also initiated a series of requests to the server using Postman, which also made it possible to test the basic functionality and identify some errors.

Although the testing falls outside the scope of this thesis, the author believes that proper testing of the server part is a mandatory requirement for further development of the application and is omitted from the thesis scope only due to lack of time.

7 Application frontend implementation

This chapter describes the main challenges during the implementation of the mobile application front-end. To implement the cross-platform application is used JavaScript React Native.

7.1 Design

Before designing the front-end application, the author made a few mockups of the main application sections in Figma, which simplified the implementation of the visual part since there is no need to think over the product design during code writing. The author is inspired by Revolut's online bank with its simple and intuitive user interface. Two of the most frequently used screens are shown in Figures 18 and 19.

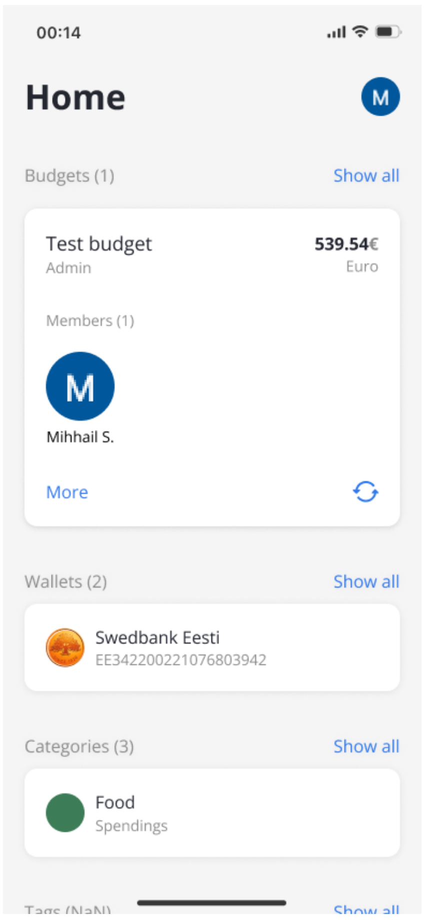


Figure 18: MYBE user home screen.

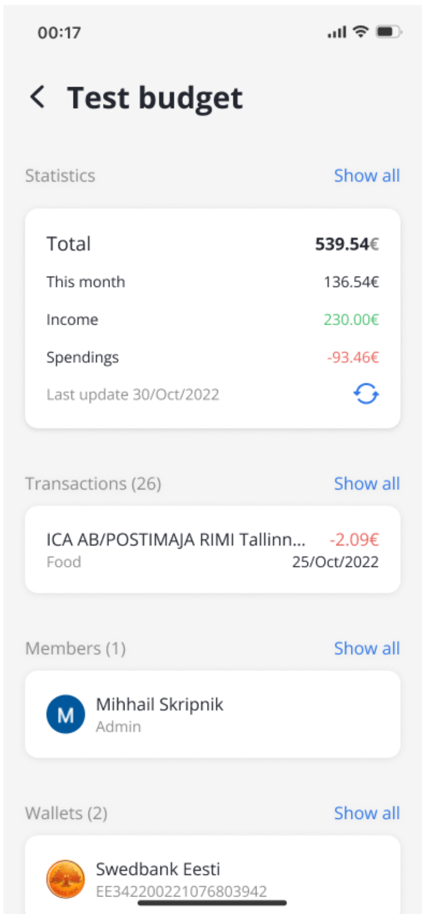
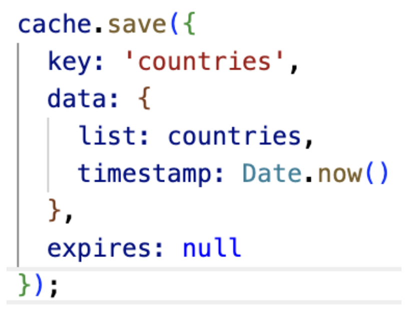


Figure 19: MYBE user budget screen.

7.2 Client-side data caching

To reduce the number of requests to the server and increase application performance, some of the least frequently changed data is cached. The user profile, user-created tags, categories, connected banks, wallets, and lists of countries and banks are cached in the application's memory to reduce loading time. To implement caching *react-native-storage* library is used. This library implements key-value data storage without any encryption (which would be redundant for such data), which means this library can be used only for non-sensitive data storing.



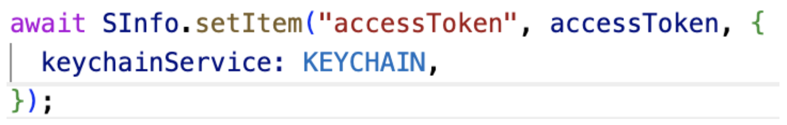
```
cache.save({
  key: 'countries',
  data: {
    list: countries,
    timestamp: Date.now()
  },
  expires: null
});
```

Figure 20: Example of *react-native-storage* library usage.

Using data caching, the loading time of some screens has greatly decreased, as a result, user info, tags, categories, and wallets screens loading becomes almost instantaneous.

7.3 Sensitive data storage

The application uses the *react-native-sensitive-info* library to implement secure storage for sensitive data. This library allows the application to securely store small chunks of data. Within the framework of this application, this library is used to store an access token, a refresh token, and a session token for calling the application backend.



```
await SInfo.setItem("accessToken", accessToken, {
  keychainService: KEYCHAIN,
});
```

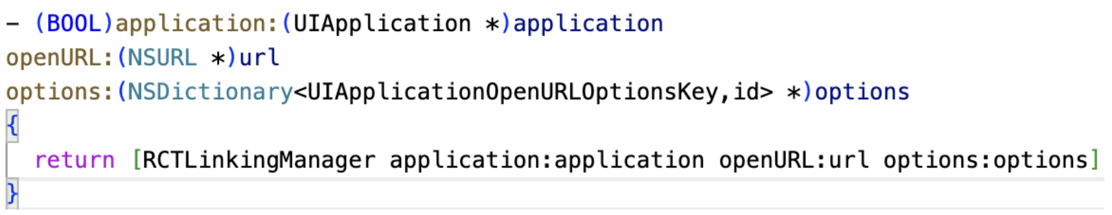
Figure 21: Example of *react-native-sensitive-info* library usage.

On the iOS platform, encrypted values are stored using system-native keychain services. On the Android platform, values are stored in application-shared memory and encrypted with the Android Keystore system. The stored value is limited to a maximum size of

2048 bytes for each key, however, this amount is more than enough for the stated purposes.

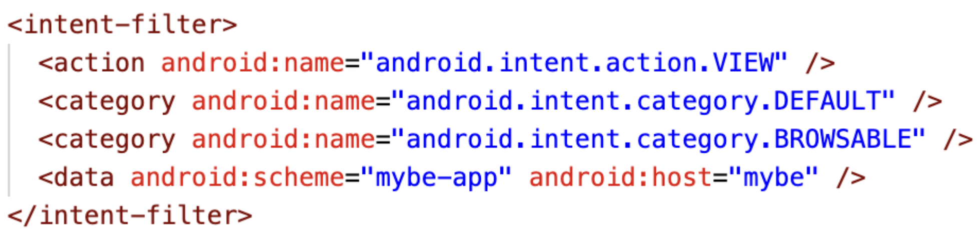
7.4 Redirections and deep links

Deep links allow to redirect the user from a web page to an application to display a specific screen with the requested content. In this project, deep links are used when authenticating a user through Google services (redirecting the user to the main screen) and connecting user wallets through the Nordigen API (redirecting to the selection of connected wallets).
The implementation of deep links requires additional configuration to work on both platforms (Figures 22 and 23).



```
- (BOOL)application:(UIApplication *)application
openURL:(NSURL *)url
options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options
{
  return [RCTLinkingManager application:application openURL:url options:options]
}
```

Figure 22: Deeplink configuration for iOS devices.



```
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data android:scheme="mybe-app" android:host="mybe" />
</intent-filter>
```

Figure 23: Deeplink configuration for Android devices.

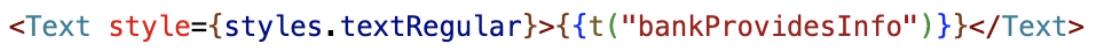
As part of this project, the author used custom URI schemes. Using this approach is like creating a "private" protocol for an application, which is then used for routing within the application. The schema URI format looks as follows - application://path/to/content.
The advantage of custom URI schemes is that they are easy to configure, while the main disadvantage is the lack of functionality for handling scenarios where the application is not installed on the user's device. Since the application is not planned to be exposed publicly (at least at this stage), the disadvantage of this approach does not affect the result in any way.

7.5 Multiple language support

Although the support of more than one language by the application is not included in the list of requirements for the project MVP, the author considered that it is necessary to lay the foundation for further expansion of the product's capabilities and the implementation of multilingualism.

To solve this problem, the author used the *react-i18next* library, which provides several extremely simple and effective components and functions for the implementation of multilingual interfaces.

The implementation of multilingualism is quite straightforward - each language has its JSON file with a list of text field keys and their values corresponding to their language. Each text component uses a translator function, which searches for the key passed as an argument in the localization file of the currently used language, returning the value for the text component. Using this transparent and straightforward approach, the translation of the interface becomes extremely simple, with almost no effect on the amount of code. Among the useful functions of this library, it is worth noting the ability to set not only a standard language, however, a fallback language as well, which will be applied if the selected language pack is not found for some reason.



```
<Text style={styles.textRegular}>{{t("bankProvidesInfo")}}</Text>
```

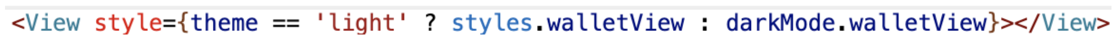
Figure 24: *react-i18next* library usage.

Within the framework of this thesis, the author has implemented only an English language pack.

7.6 Color themes

The implementation of the functionality for changing color themes is also not a requirement for the MVP of the project, however, according to the author, this functionality should be included at the initial stage, which will facilitate its implementation later.

The implementation of color styles from the code's point of view is quite simple - each UI component has 2 CSS styles (for light and dark themes, respectively), which are applied following the color scheme used on the user device. The application stores this information in local storage using the Appearance module and tracking the color scheme change event. When the style is rendered, the component receives the stored value and selects the appropriate schema for the CSS style (see Figure 25).



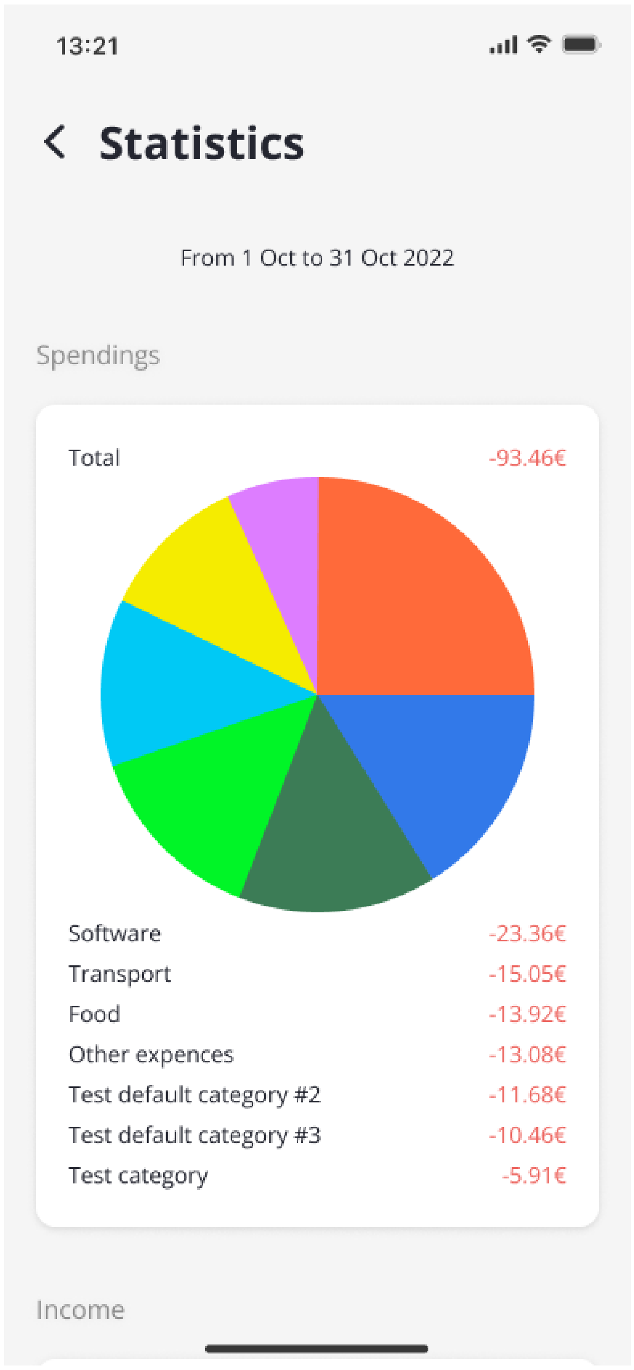
```
<View style={theme == 'light' ? styles.walletView : darkMode.walletView}></View>
```

Figure 25: Component color scheme definition.

Although for this application with a ready-made codebase, the implementation of a dark theme requires only a set of styles, the author considered the implementation too time-consuming and therefore limited himself to using the standard light theme.

7.7 Data charts

Working with data visualization, the author used the *react-native-pie* library for a simpler and more intuitive presentation of user statistics. The library usage is extremely simple - the main component takes several different parameters, such as an array of data (it is an array of objects with two parameters - filling percentage and fill color), the chart radius, the size of the inner radius, the background of the chart, and the display type (there are two views - chart and percentage display). As shown in Figure 26, the library generates a simple and easy-to-understand graph used for the application's statistics module.


13:21

Statistics

From 1 Oct to 31 Oct 2022

Spendings

Total	-93.46€
Software	-23.36€
Transport	-15.05€
Food	-13.92€
Other expences	-13.08€
Test default category #2	-11.68€
Test default category #3	-10.46€
Test category	-5.91€

Income

Figure 26: Generated pie chart using *react-native-pie* library.

8 Application analysis

As part of this dissertation, the author created an MVP for an application for accounting for a personal or household budget. During the implementation and testing of the mobile application, the author received enough user experience to identify all the main advantages and disadvantages of the application.

From a user point of view, among the positive sides of the application, the author would like to note the following:

1. Full compliance with all the requirements described in Chapter 3. Application concept and requirements for MVP.
2. Ability to manage multiple budgets at the same time using joint bank accounts does not require synchronizing with the bank more than once.
3. The number of synchronizations of transactions with the bank is dynamic and depends on bank restrictions, while other applications usually use constant values.
4. Simple and fast registration and authorization processes using Google service.
5. All data is dynamically stored on the application backend side and tied to the user's Google account.

Among the tangible shortcomings of the application, the author notes the following:

1. Although the mobile part of the application complies with the requirements of the MVP, the application needs additional testing and technical improvement due to intermittent errors on the client side.
2. High battery consumption for an unspecified reason.
3. Unstable operation of the application on the version of Android 12 on Xiaomi devices (this problem is not discovered on Samsung devices).
4. Lack of support for cryptocurrencies and crypto exchanges.

Even though the application created within the framework of this dissertation is still far from a complete product, the author would like to note that already at this stage, there is a visible superiority over some existing analogues due to the automatic synchronization of transactions, joint budget management, remote data storage and the ability to configure or create new user tags and categories.

9 Possible improvements and further development

At this stage, the client part of the application requires great improvements since the current project structure can be excessively overloaded and contains, in some places, the re-implementation of existing functionality. Among the factors that negatively affected the quality of the client side, one of the most important is the lack of experience and time.

The server part of the application undoubtedly requires writing additional integration and unit tests that could detect previously undetected or missed errors and bugs. For the further development of the application, the components responsible for the secure storage and modification of user data should certainly be reviewed by more experienced specialists. Legal advice is also needed regarding the requirements for the processing and storage of user personal and banking data. There is also a need to conduct additional stress testing to identify performance-reducing factors for further code optimizations.

It should also be noted that the author abandoned the original idea of positioning this project as an open-source solution due to the potential danger of providing public access to the application code that works with extremely sensitive user data (bank accounts and transactions).

This application could be a significant competitor to existing solutions after eliminating all existing technical imperfections, meeting all requirements for working with client banking data, and adding a more extensive choice of available synchronization banks.

10 Summary

The main goal of the bachelor's work is to create an application for managing personal and family budgets based on the analysis of solutions existing on the market, as well as subsequently derived the author's concept. Analysis of existing applications reveals a big scope for possible improvements of home budgeting applications. The application created within the framework of this dissertation allows the user to automatically receive banking transactions for further categorization. The data aggregated by the application allows users to view simple and understandable statistics of all user's income and expenses. Extensive customization options for custom categories and tags make the generated charts more personalized, increasing the efficiency of budget analysis. Inviting other users to manage created budgets provides a convenient way to manage a joint budget. Storing user data on the application's server-side allows the client to access data from different devices. Cross-platform implementation of the client application provides support for the biggest part of devices used in the world. Encrypted storage and periodic change of encryption keys for client data provide a high level of security.

The implemented functionality meets the requirements stated for the MVP, however, it is worth considering that the application still requires significant improvement and legal consultation for mass use. The final application and its concept combine all the advantages of existing solutions, partially solving their shortcomings. In conclusion, it can be noted that this project is well suited for further development, as it requires a lot of different kinds of minor improvements, however, it already has a ready-to-use main functionality core.

References

[1] cloudflare.com, "What is HTTPS?" [Online]
Available: https://www.cloudflare.com/learning/ssl/what-is-https/
Accessed 10 Oct 2022

[2] budgetbakers.com, Wallet app by Budgetbakers. [Online]
Available: https://budgetbakers.com/
Accessed 11 Oct 2022

[3] monefy.me, Monefy app. [Online]
Available: https://monefy.me/
Accessed 11 Oct 2022

[4] bilanceapp.com, Bilance app. [Online]
Available: https://www.bilanceapp.com/
Accessed 11 Oct 2022

[5] G. Dreimanis, "Introduction to Elixir", serokell.io, 15 04 2020. [Online]
Available: https://serokell.io/blog/introduction-to-elixir
Accessed 14 Oct 2022

[6] R. Lazzara, "How to build fault-tolerant software systems", Computer Science Blog, 13 10 2019. [Online]
Available:
https://blog.mi.hdm-stuttgart.de/index.php/2019/10/13/how-to-build-fault-tolerant-software-systems/
Accessed 14 Oct 2022

[7] "Memory prices 1957+". [Online]
Available: https://jcmit.net/memoryprice.htm
Accessed 14 Oct 2022

[8] R. Gin, "5 Benefits of Immutable Objects Worth Considering for Your Next Project", hackernoon.com, 18 07 2017. [Online]
Available:
https://hackernoon.com/5-benefits-of-immutable-objects-worth-considering-for-your-next-project-f98e7e85b6ac
Accessed 15 Oct 2022

[9] R. Hryniewicz, "What is App Scaling and why Elixir is Perfect for Scalable Applications?", curiosum.com, 04 10 2022. [Online]
Available:
https://curiosum.com/blog/what-is-app-scaling-why-elixir-perfect-scalable-app
Accessed 15 Oct 2022

[10] S.Vishnevskiy, "How Discord scaled Elixir to 5,000,000 concurrent users", discord.com, 06 07 2017. [Online]
Available:
https://discord.com/blog/how-discord-scaled-elixir-to-5-000-000-concurrent-users
Accessed 16 Oct 2022

[11] Pinterest Engineering Blog, "Introducing new open-source tools for the Elixir community", medium.com, 18 12 2015. [Online]
Available:
https://medium.com/pinterest-engineering/introducing-new-open-source-tools-for-the-elixir-community-2f7bb0bb7d8c
Accessed 16 Oct 2022

[12] "Caching Options in an Elixir Application", prying.io, 01 09 2019. [Online]
Available:
https://prying.io/technical/2019/09/01/caching-options-in-an-elixir-application.html
Accessed 16 Oct 2022

[13] "2022 Developer Survey", survey.stackoverflow.co. [Online]
Available:
https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-web-frameworks-and-technologies
Accessed 17 Oct 2022

[14] L. Gupta, "What is REST", restfulapi.net, 07 04 2022. [Online]
Available: https://restfulapi.net/
Accessed 17 Oct 2022

[15] "Things Elixir's Phoenix framework does right", scorpio.com, 25 09 2020. [Online]
Available: https://scorpil.com/post/things-elixirs-phoenix-framework-does-right/
Accessed 17 Oct 2022

[16] PostgreSQL 14 Documentation - MVCC Introduction, postgresql.org. [Online]
Available: https://www.postgresql.org/docs/14/mvcc-intro.html
Accessed 18 Oct 2022

[17] PostgreSQL 14 Documentation - Overview, postgresql.org. [Online]
Available: https://www.postgresql.org/docs/14/plpgsql-overview.html
Accessed 18 Oct 2022

[18] "About", postgresql.org. [Online]
Available: https://www.postgresql.org/about/
Accessed 18 Oct 2022

[19] PostgreSQL 14 Documentation - Extensions, postgresql.org. [Online]
Available: https://www.postgresql.org/docs/14/external-extensions.html
Accessed 18 Oct 2022

[20] "Chapter 1. What Is React Native?", oreilly.com. [Online]
Available:

https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html
Accessed 19 Oct 2022

[21] K. Shah, "Advantages and Disadvantages of React Native Development in 2022", thirdrockechkno.com, 20 05 2021. [Online]
Available:
https://www.thirdrocktechkno.com/blog/pros-and-cons-of-react-native-development-in-2021/
Accessed 19 Oct 2022

[22] "Using OAuth 2.0 to Access Google APIs", developers.google.com. [Online]
Available: https://developers.google.com/identity/protocols/oauth2
Accessed 20 Oct 2022

[23] D. Hardt, "The OAuth 2.0 Authorization Framework", rfc-editor.org, 10 2012. [Online]
Available: https://www.rfc-editor.org/rfc/rfc6749
Accessed 20 Oct 2022

[24] "Free alternative to Tink", nordigen.com. [Online]
Available: https://nordigen.com/en/free-alternative-to-tink/
Accessed 23 Oct 2022

[25] "Free alternative to Plaid", nordigen.com. [Online]
Available: https://nordigen.com/en/free-alternative-to-plaid/
Accessed 23 Oct 2022

[26] "Free alternative to Truelayer", nordigen.com. [Online]
Available: https://nordigen.com/en/free-alternative-to-truelayer/
Accessed 23 Oct 2022

[27] "What is PSD2?", openbankproject.com. [Online]
Available: https://www.openbankproject.com/psd2-2/
Accessed 23 Oct 2022

[28] "Nordigen data points". [Online]
Available:
https://docs.google.com/spreadsheets/d/11tAD5cfrlaOZ4HXI6jPpL5hMf8ZuRYc6TUXTxZE84A8/edit#gid=0
Accessed 23 Oct 2022

[29] "exchangerate.host FAQ". [Online]
Available: https://exchangerate.host/#/#faq
Accessed 24 Oct 2022

[30] D. Endler, "How Much Data Was Leaked To Cybercriminals In 2020 — And What They're Doing With It", forbes.com, 20 04 2021. [Online]
Available:
https://www.forbes.com/sites/forbestechcouncil/2021/04/20/how-much-data-was-leaked-to-cybercriminals-in-2020---and-what-theyre-doing-with-it/?sh=1b948cd51f03
Accessed 26 Oct 2022

[31] M. Green, "A Few Thoughts on Cryptographic Engineering", cryptographyengineering.com. [Online]
Available: https://blog.cryptographyengineering.com/
Accessed 27 Oct 2022

[32] R. Franklin, "AES vs. RSA Encryption: What Are the Differences?", precisely.com, 13 03 2021. [Online]
Available:
https://www.precisely.com/blog/data-security/aes-vs-rsa-encryption-differences#:~:text=Because%20there%20is%20no%20known,only%20small%20amounts%

20of%20data.
Accessed 27 Oct 2022

[33] "Password Hashing Competition". [Online]
Available: https://www.password-hashing.net/
Accessed 27 Oct 2022

[34] "Key rotation", Google Cloud documentation. [Online]
Available: https://cloud.google.com/kms/docs/key-rotation
Accessed 27 Oct 2022

[35] "Phoenix Ecto Encryption Example". [Online]
Available:
https://github.com/dwyl/phoenix-ecto-encryption-example#31-encrypt
Accessed 27 Oct 2022

[36] "Bank API rate limits", Nordigen. [Online]
Available:
https://nordigen.zendesk.com/hc/en-gb/articles/6761006738717-Bank-API-Rate-Limits
Accessed 28 Oct 2022

[37] E. Viçosa Jr, "Caching with Elixir and ETS", blog.appsignal.com, 12 11 2019. [Online]
Available:
https://blog.appsignal.com/2019/11/12/caching-with-elixir-and-ets.html
Accessed 28 Oct 2022

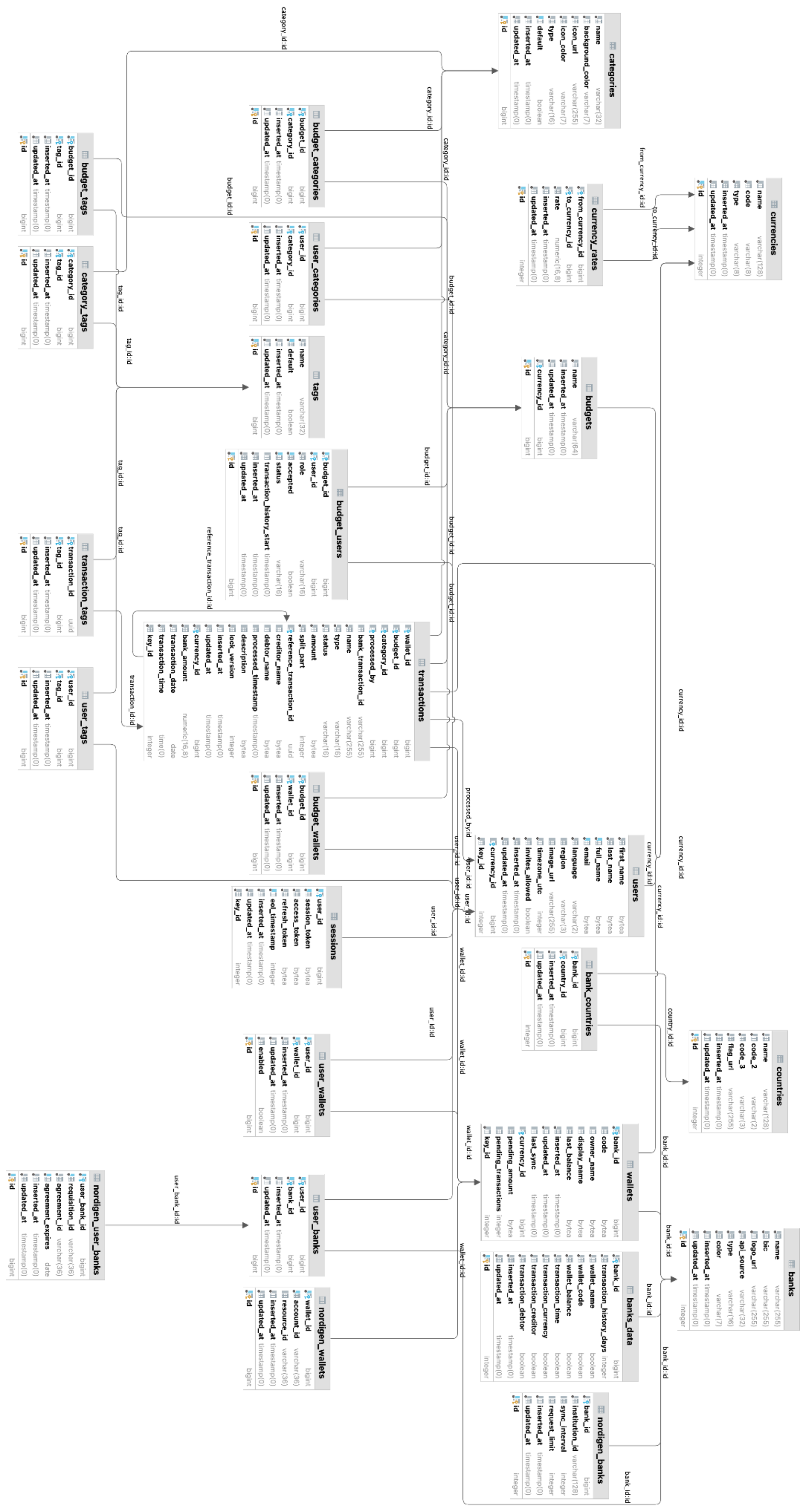
[38] React Native official documentation. [Online]
Available: https://reactnative.dev/docs/typescript
Accessed 30 Oct 2022

[39] Google Play Market. [Online]
Available:
https://play.google.com/store/search?q=budget+management&c=apps&hl=ee&gl=US
Accessed 15 Dec 2022

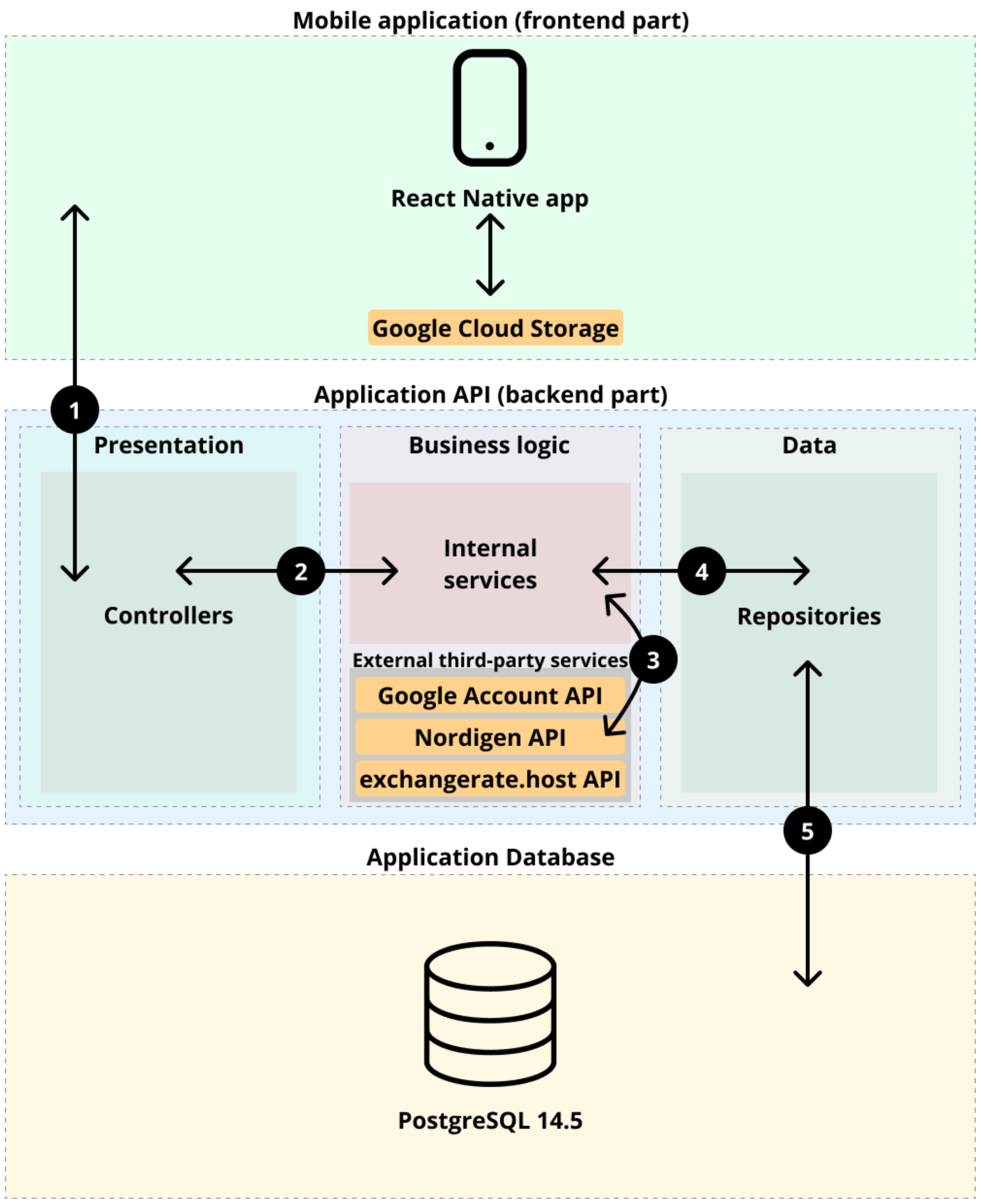
Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Application database schema



Appendix 3 - Application overall architecture schema



Appendix 4 - API endpoints

Namespace	Endpoint	Request method	Description
/auth	/identify	POST	Used to identify the user, it takes as an argument the Google authorization code, which will be "exchanged" for an access token. Using an access token application receives information about the user from Google. If this user exists in the system, the application returns a session token, an access token, and a refresh token. The application also returns the user's profile if the user has not been registered before.
/users	/info	GET	Returns the user's profile.
	/update	PATCH	Used to change the user's base currency or time zone.
	/remove	DELETE	Deletes a user and all associated data.
/sessions	/refresh	POST	It takes a session token and a refresh token as an argument, returning a new refresh token and an access token.
	/logout	DELETE	Deletes the current user session.
/budgets	/create	POST	Creates a new budget and automatically adds standard tags and categories, as well as the base currency selected when creating the budget.
	/connect	POST	Used to connect tags, categories, or wallets

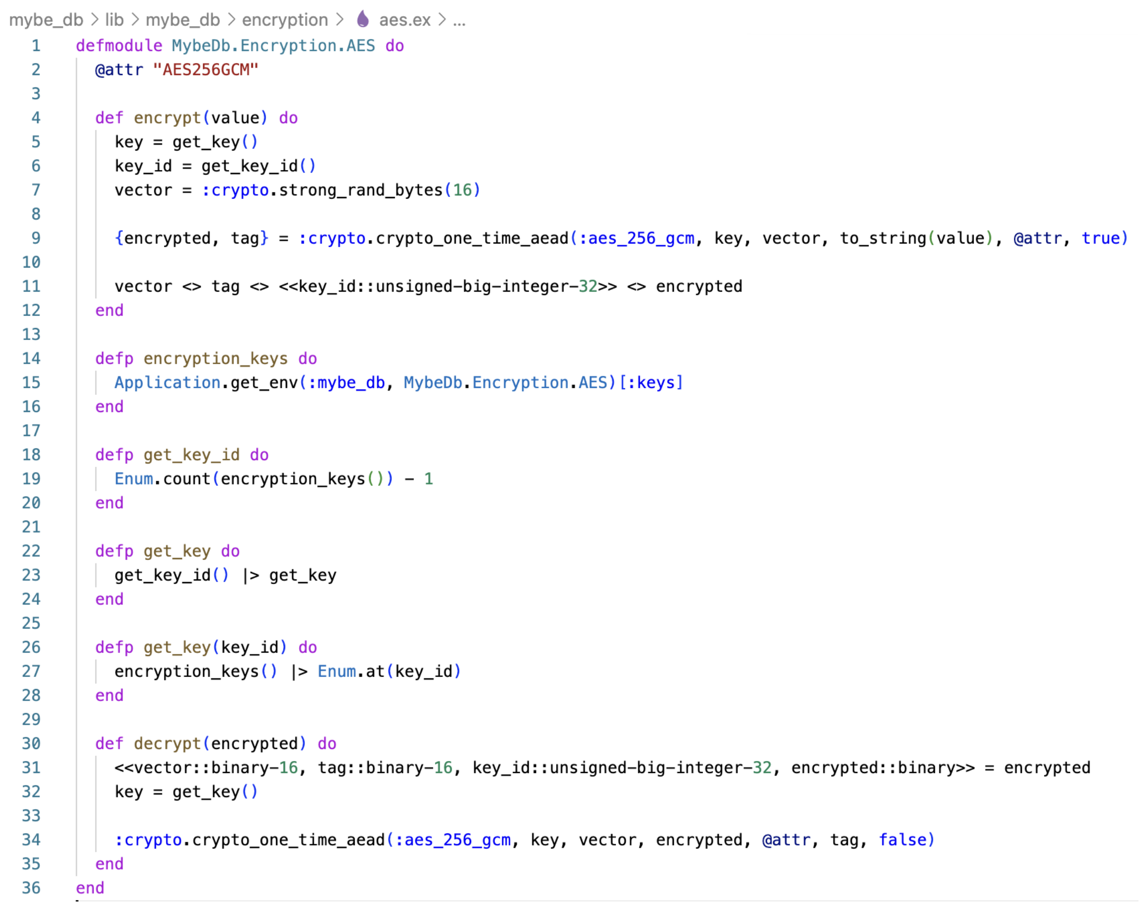
			to the budget. It takes as an argument the type of the connected entity and its identifier.
	/list	GET	Returns a list of user budgets with the user's access level.
	/info	GET	Returns the details of the specified user budget (connected categories, tags, wallets, and transactions for the current month).
	/update	PATCH	Used to change the name or base currency of the budget.
	/remove	DELETE	Deletes the budget and all related data.
/banks	/connect	POST	Used to connect a user bank account through the Nordigen service.
	/list	GET	Returns a list of all available banks.
/countries	/list	GET	Returns a list of countries with a list of supported banks for each country.
/wallets	/connect	POST	Connects selected by user wallets after synchronization via Nordigen.
	/list	GET	Returns a list of connected user wallets.
	/remove	DELETE	Deletes selected user wallets.
/categories	/create	POST	Creates a user category with the selected parameters (icon, color, title, category type).
	/list	GET	Returns a list of user categories.
	/update	PATCH	Updates a user category.

	/remove	DELETE	Deletes a user category.
/tags	/create	POST	Creates a user tag with the selected options (tag type, related categories).
	/list	GET	Returns a list of user tags.
	/update	PATCH	Updates a user tag.
	/remove	DELETE	Deletes a user tag.
/transactions	/sync	POST	Returns a list of transactions from the Nordigen service for the budget specified in the request (maximum one budget).
	/create	POST	Used to create a transaction manually (for tracking cash spendings).
	/split	POST	Splits the specified transaction into two different ones with the specified amount.
	/list	GET	Returns a list of transactions for the specified budget. It takes a time range as an argument.
	/process	PATCH	Used to edit or categorize budget transactions.
	/remove	DELETE	Deletes the specified transaction (works only for manually created transactions).
/invites	/create	POST	Used to create an invitation for joint budget management.
	/list	GET	Returns a list of active user invites.
	/update	PATCH	Used to set the status of the specified invitation (accepted/rejected).

	/remove	DELETE	Revokes the specified invitation.
/currencies	/list	GET	Returns a list of available currencies.
	/rates	GET	Returns the exchange rate for the specified currency pair.

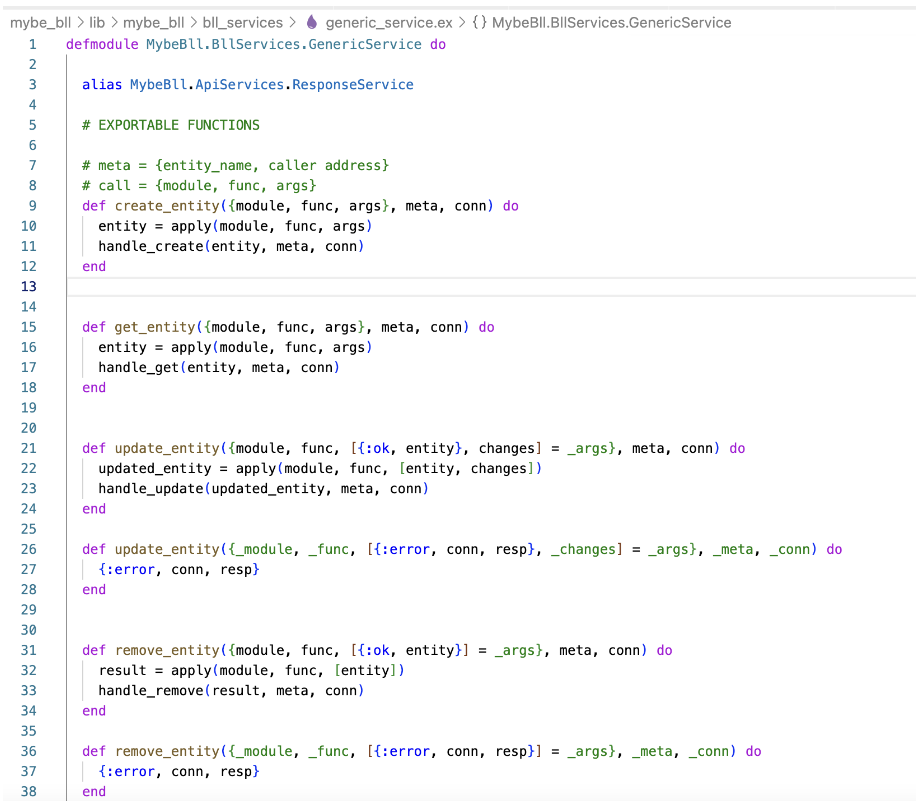
/statistics	/calculate	GET	Returns the sum of all expenses and income separated according to the specified filter (category, tag, or wallet) for the specified period. Example:
			{
			  "from": "12/10/2022",
			  "to": "25/10/2022",
			  "category #1": 250.34,
			  …
			}

Appendix 5 - Data encryption & decryption module using AES



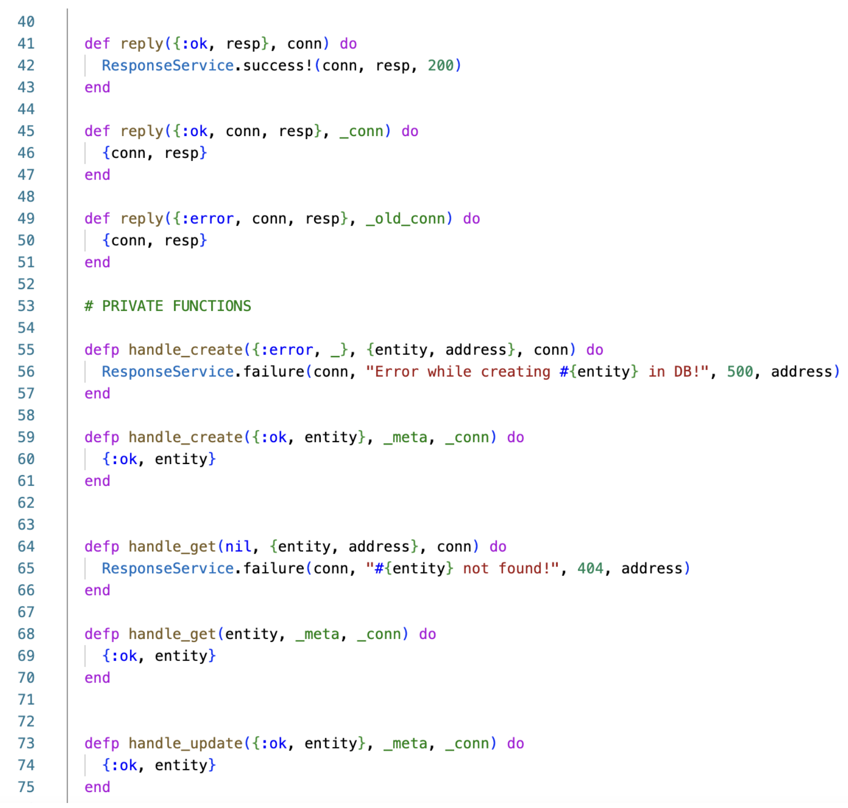
```elixir
defmodule MybeDb.Encryption.AES do
  @attr "AES256GCM"

  def encrypt(value) do
    key = get_key()
    key_id = get_key_id()
    vector = :crypto.strong_rand_bytes(16)

    {encrypted, tag} = :crypto.crypto_one_time_aead(:aes_256_gcm, key, vector, to_string(value), @attr, true)

    vector <> tag <> <<key_id::unsigned-big-integer-32>> <> encrypted
  end

  defp encryption_keys do
    Application.get_env(:mybe_db, MybeDb.Encryption.AES)[:keys]
  end

  defp get_key_id do
    Enum.count(encryption_keys()) - 1
  end

  defp get_key do
    get_key_id() |> get_key
  end

  defp get_key(key_id) do
    encryption_keys() |> Enum.at(key_id)
  end

  def decrypt(encrypted) do
    <<vector::binary-16, tag::binary-16, key_id::unsigned-big-integer-32, encrypted::binary>> = encrypted
    key = get_key()

    :crypto.crypto_one_time_aead(:aes_256_gcm, key, vector, encrypted, @attr, tag, false)
  end
end
```

Appendix 6 - Business logic layer services - generic service



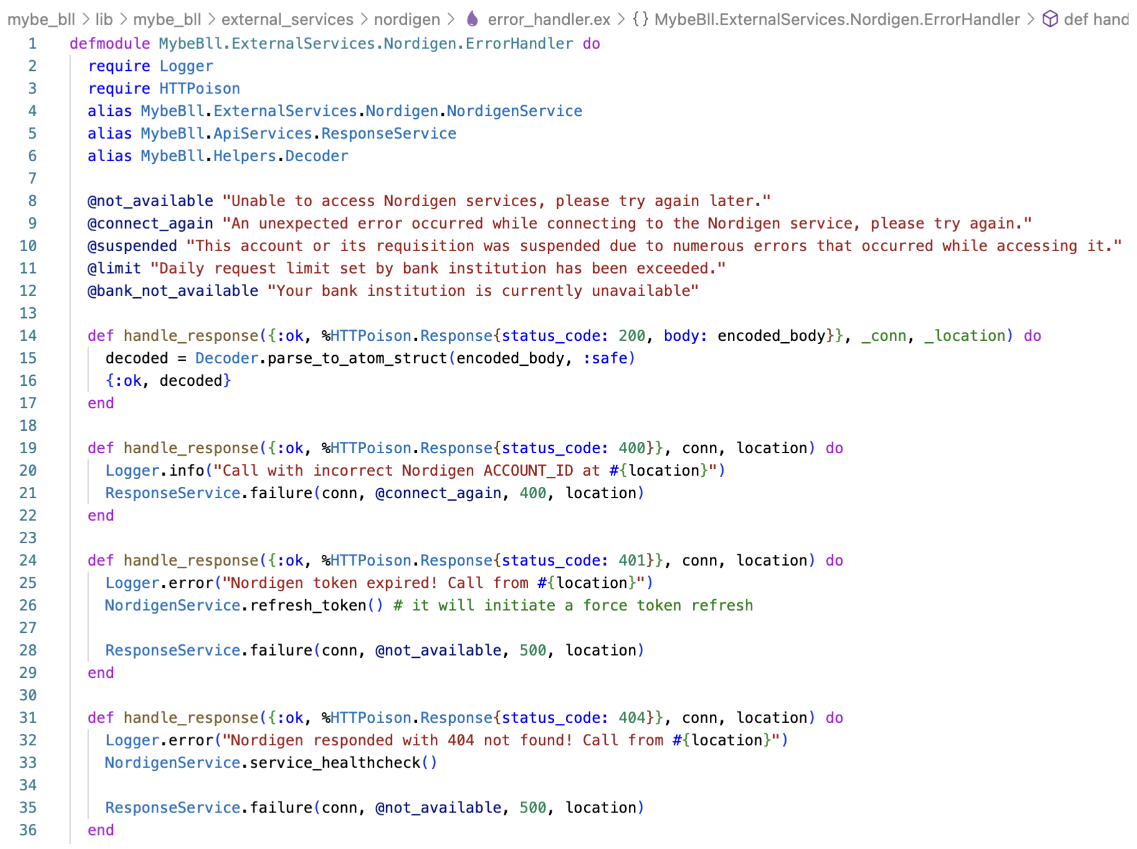
mybe_bll > lib > mybe_bll > bll_services > generic_service.ex > { } MybeBll.BllServices.GenericService

```elixir
defmodule MybeBll.BllServices.GenericService do

  alias MybeBll.ApiServices.ResponseService

  # EXPORTABLE FUNCTIONS

  # meta = {entity_name, caller address}
  # call = {module, func, args}
  def create_entity({module, func, args}, meta, conn) do
    entity = apply(module, func, args)
    handle_create(entity, meta, conn)
  end


  def get_entity({module, func, args}, meta, conn) do
    entity = apply(module, func, args)
    handle_get(entity, meta, conn)
  end


  def update_entity({module, func, [{:ok, entity}, changes] = _args}, meta, conn) do
    updated_entity = apply(module, func, [entity, changes])
    handle_update(updated_entity, meta, conn)
  end

  def update_entity({_module, _func, [{:error, conn, resp}, _changes] = _args}, _meta, _conn) do
    {:error, conn, resp}
  end


  def remove_entity({module, func, [{:ok, entity}] = _args}, meta, conn) do
    result = apply(module, func, [entity])
    handle_remove(result, meta, conn)
  end

  def remove_entity({_module, _func, [{:error, conn, resp}] = _args}, _meta, _conn) do
    {:error, conn, resp}
  end
```

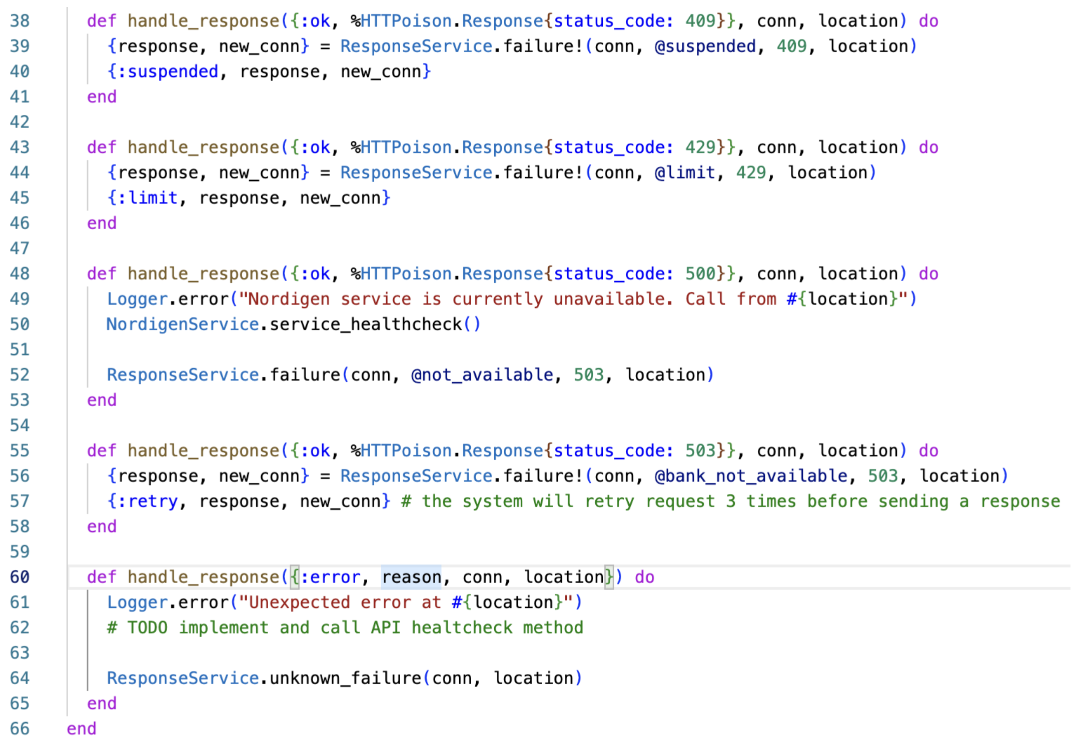


```elixir

  def reply({:ok, resp}, conn) do
    ResponseService.success!(conn, resp, 200)
  end

  def reply({:ok, conn, resp}, _conn) do
    {conn, resp}
  end

  def reply({:error, conn, resp}, _old_conn) do
    {conn, resp}
  end

  # PRIVATE FUNCTIONS

  defp handle_create({:error, _}, {entity, address}, conn) do
    ResponseService.failure(conn, "Error while creating #{entity} in DB!", 500, address)
  end

  defp handle_create({:ok, entity}, _meta, _conn) do
    {:ok, entity}
  end


  defp handle_get(nil, {entity, address}, conn) do
    ResponseService.failure(conn, "#{entity} not found!", 404, address)
  end

  defp handle_get(entity, _meta, _conn) do
    {:ok, entity}
  end


  defp handle_update({:ok, entity}, _meta, _conn) do
    {:ok, entity}
  end

  defp handle_update({:error, _}, {entity, address}, conn) do
    ResponseService.failure(conn, "Database error while updating #{entity}!", 500, address)
  end

  defp handle_update({:error, conn, resp}, _meta, _conn) do
    {:error, conn, resp}
  end


  defp handle_remove({:ok, _result}, {entity, _address}, _conn) do
    {:ok, %{"Message" => "#{entity} has been successfully removed!"}}
  end

  defp handle_remove({:error, _}, {entity, address}, conn) do
    ResponseService.failure(conn, "Database error while removing #{entity}!", 500, address)
  end

  defp handle_remove({:error, conn, resp}, _meta, _conn) do
    {:error, conn, resp}
  end
end

```

Appendix 7 - Business logic layer services - Nordigen service error handler

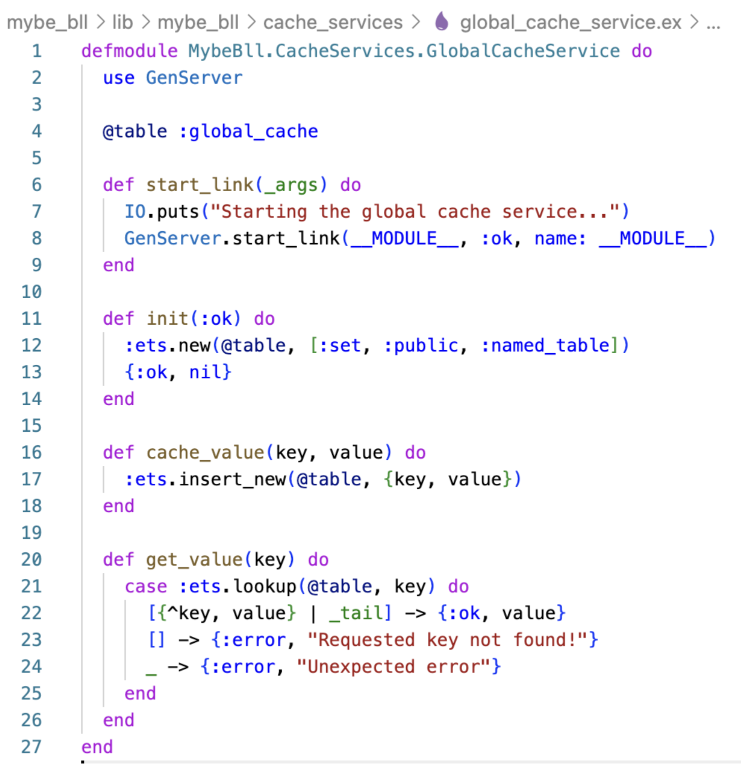

mybe_bll > lib > mybe_bll > external_services > nordigen > error_handler.ex > {} MybeBll.ExternalServices.Nordigen.ErrorHandler > def hand

```elixir
defmodule MybeBll.ExternalServices.Nordigen.ErrorHandler do
  require Logger
  require HTTPoison
  alias MybeBll.ExternalServices.Nordigen.NordigenService
  alias MybeBll.ApiServices.ResponseService
  alias MybeBll.Helpers.Decoder

  @not_available "Unable to access Nordigen services, please try again later."
  @connect_again "An unexpected error occurred while connecting to the Nordigen service, please try again."
  @suspended "This account or its requisition was suspended due to numerous errors that occurred while accessing it."
  @limit "Daily request limit set by bank institution has been exceeded."
  @bank_not_available "Your bank institution is currently unavailable"

  def handle_response({:ok, %HTTPoison.Response{status_code: 200, body: encoded_body}}, _conn, _location) do
    decoded = Decoder.parse_to_atom_struct(encoded_body, :safe)
    {:ok, decoded}
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 400}}, conn, location) do
    Logger.info("Call with incorrect Nordigen ACCOUNT_ID at #{location}")
    ResponseService.failure(conn, @connect_again, 400, location)
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 401}}, conn, location) do
    Logger.error("Nordigen token expired! Call from #{location}")
    NordigenService.refresh_token() # it will initiate a force token refresh

    ResponseService.failure(conn, @not_available, 500, location)
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 404}}, conn, location) do
    Logger.error("Nordigen responded with 404 not found! Call from #{location}")
    NordigenService.service_healthcheck()

    ResponseService.failure(conn, @not_available, 500, location)
  end
```



```elixir
  def handle_response({:ok, %HTTPoison.Response{status_code: 409}}, conn, location) do
    {response, new_conn} = ResponseService.failure!(conn, @suspended, 409, location)
    {:suspended, response, new_conn}
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 429}}, conn, location) do
    {response, new_conn} = ResponseService.failure!(conn, @limit, 429, location)
    {:limit, response, new_conn}
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 500}}, conn, location) do
    Logger.error("Nordigen service is currently unavailable. Call from #{location}")
    NordigenService.service_healthcheck()

    ResponseService.failure(conn, @not_available, 503, location)
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 503}}, conn, location) do
    {response, new_conn} = ResponseService.failure!(conn, @bank_not_available, 503, location)
    {:retry, response, new_conn} # the system will retry request 3 times before sending a response
  end

  def handle_response({:error, reason, conn, location}) do
    Logger.error("Unexpected error at #{location}")
    # TODO implement and call API healtcheck method

    ResponseService.unknown_failure(conn, location)
  end
end
```

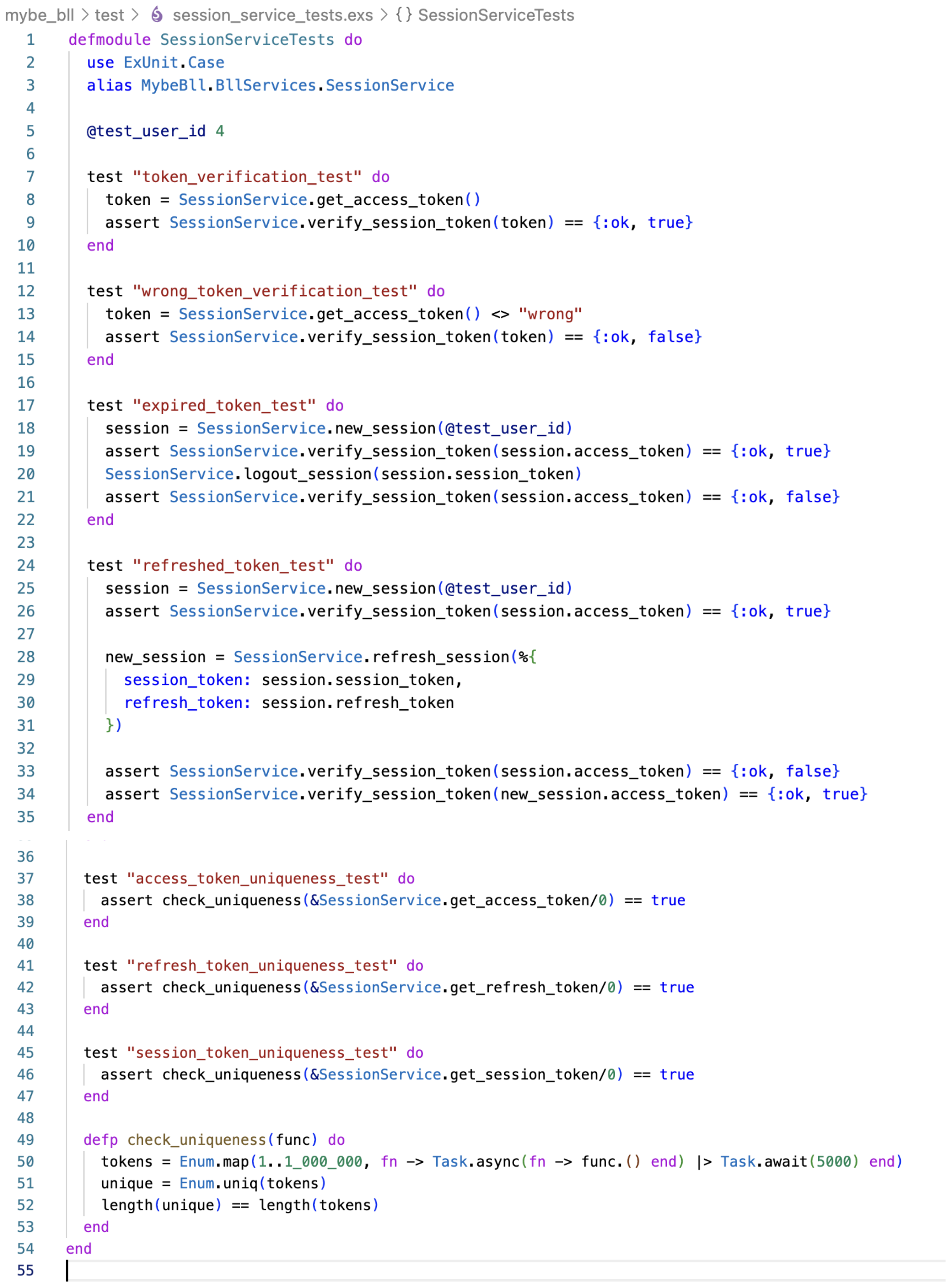
Appendix 8 - Simple cache service using ETS.



mybe_bll > lib > mybe_bll > cache_services > global_cache_service.ex > ...

```elixir
defmodule MybeBll.CacheServices.GlobalCacheService do
  use GenServer

  @table :global_cache

  def start_link(_args) do
    IO.puts("Starting the global cache service...")
    GenServer.start_link(__MODULE__, :ok, name: __MODULE__)
  end

  def init(:ok) do
    :ets.new(@table, [:set, :public, :named_table])
    {:ok, nil}
  end

  def cache_value(key, value) do
    :ets.insert_new(@table, {key, value})
  end

  def get_value(key) do
    case :ets.lookup(@table, key) do
      [{^key, value} | _tail] -> {:ok, value}
      [] -> {:error, "Requested key not found!"}
      _ -> {:error, "Unexpected error"}
    end
  end
end
```

Appendix 9 - Session service unit tests



mybe_bll > test > session_service_tests.exs > { } SessionServiceTests

```elixir
defmodule SessionServiceTests do
  use ExUnit.Case
  alias MybeBll.BllServices.SessionService

  @test_user_id 4

  test "token_verification_test" do
    token = SessionService.get_access_token()
    assert SessionService.verify_session_token(token) == {:ok, true}
  end

  test "wrong_token_verification_test" do
    token = SessionService.get_access_token() <> "wrong"
    assert SessionService.verify_session_token(token) == {:ok, false}
  end

  test "expired_token_test" do
    session = SessionService.new_session(@test_user_id)
    assert SessionService.verify_session_token(session.access_token) == {:ok, true}
    SessionService.logout_session(session.session_token)
    assert SessionService.verify_session_token(session.access_token) == {:ok, false}
  end

  test "refreshed_token_test" do
    session = SessionService.new_session(@test_user_id)
    assert SessionService.verify_session_token(session.access_token) == {:ok, true}

    new_session = SessionService.refresh_session(%{
      session_token: session.session_token,
      refresh_token: session.refresh_token
    })

    assert SessionService.verify_session_token(session.access_token) == {:ok, false}
    assert SessionService.verify_session_token(new_session.access_token) == {:ok, true}
  end

  test "access_token_uniqueness_test" do
    assert check_uniqueness(&SessionService.get_access_token/0) == true
  end

  test "refresh_token_uniqueness_test" do
    assert check_uniqueness(&SessionService.get_refresh_token/0) == true
  end

  test "session_token_uniqueness_test" do
    assert check_uniqueness(&SessionService.get_session_token/0) == true
  end

  defp check_uniqueness(func) do
    tokens = Enum.map(1..1_000_000, fn -> Task.async(fn -> func.() end) |> Task.await(5000) end)
    unique = Enum.uniq(tokens)
    length(unique) == length(tokens)
  end
end

```