TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Nikita Viira 185269IADB

Financial Data Aggregation from Multiple Sources

Bachelor's thesis

Supervisor: Mohammad Tariq Meeran Ph.D.

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Nikita Viira 185269IADB

Finantsandmete agregeerimine mitmest allikast

Bakalaureusetöö

Juhendaja: Mohammad Tariq Meeran Ph.D.

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: Nikita Viira

04.01.2023

Abstract

The thesis aims to create a mobile application which would allow users to connect their banks and crypto exchanges and see the overall picture of their portfolio.

The thesis is divided into 2 major parts, literature review, and implementation of the project. Throughout reviewing the literature, the author does an overview of similar apps, and decides which technology stack and API providers will be used. Literature review is followed by the creation of the methodology for the research and the creation of the requirements for the experiment.

The development process is described thoroughly and divided into 2 big parts, backend and frontend. The backend part describes the database structure, security measures, project structure, as well as testing process. The frontend part describes the creation of the single-page web application and its conversion into a mobile application.

The result of the thesis is a mobile application which follows all of the MVP criteria. Future development ideas and test results are also described at the end of the thesis.

This thesis is written in English and is 46 pages long, including 7 chapters, 27 figures and 3 tables.

Annotatsioon

Finantsandmete agregeerimine mitmest allikast

Lõputöö eesmärk on luua mobiilirakendus, mis võimaldaks kasutajatel ühendada oma pangakontod ja krüptobörsid ning saada võimalus näha portfelli üldpilti.

Lõputöö on jagatud kaheks suureks osaks, milleks on kirjanduse ülevaatus ja projekti loomine. Kirjanduse ülevaatuse käigus annab autor ülevaadet sarnastest äppidest ja otsustab, millised tehnoloogiad ja API pakkujad kasutatakse.

Arendusprotsess on põhjalikult kirjeldatud ja jagatud kaheks suureks osaks, taga- ja ees rakenduseks. Tagarakenduse osa kirjeldab andmebaasi struktuuri, turvameetmeid, projekti struktuuri ja testimisprotsessi. Eesrakenduse osa kirjeldab ühelehelise veebirakenduse loomist ja selle konverteerimist mobiilirakenduseks.

Lõputöö tulemuseks on mobiilirakendus, mis järgib kõiki MVP kriteeriume. Töö lõpus on kirjeldatud ka edasisi arendusideid ja testitulemusi.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 46 leheküljel, 7 peatükki, 27 joonist, 3 tabelit.

List of abbreviations and terms

Annotation (Java)	Metadata about the program embedded in the program itself. It can be parsed by the annotation parsing tool or by the compiler.
API	Application Programming Interface – the connection between different applications
Bean	A Java object managed by the Spring framework's context
Benchmark	Process of running software or hardware against a series of tests to understand how well it performs
Blockchain	Digital ledger of transactions that is duplicated and distributed across the entire network of computer systems
Cache	Software or hardware component, that stores data so that future requests for that data can be served faster
CPU	Central processing unit – computer processor
CSS	Cascading Style Sheets – style sheet language, which describes the presentation of the web page
Deep linking	Process of delivering users to a specific page inside the app even if they are outside of the app
Dependency injection	Implementation of the Inversion of Control (IoC) principle which uses a constructor to inject objects into other objects
DOM	Document Object Model – a tree of objects describing the web page
Fiat money	Government-issued currency that is not backed by a physical commodity, such as gold or silver
Framework	Abstraction, which provides default functionalities and can be selectively changed by the developers
HTML	Hypertext Markup Language – markup language, which defines the structure of the web page
HTTP	Hypertext Transfer Protocol
IDE	Integrated development environment – code editor with a lot of custom functions (database tools, version control, etc…)
JDBC	Java Database Connectivity – API for Java, which allows establishing database connection from code

JS	JavaScript – a high-level prototype-based programming language for web development that is one of the core web technologies
JSON	JavaScript Object Notation – data structure
JVM	Java virtual machine – a virtual machine that enables a computer to run Java bytecode
JWT	JSON Web Token – JSON payload signed with a private secret which is used for authentication
Library	A publicly shared code base which can be loaded into any code base and contains ready-made implementations
Locale	A set of parameters which defines the user's language
MVC	Model-View-Controller – development pattern, which divides the program logic into three interconnected layers
MVP	Minimal Viable Product – version of the product, which fulfils all early-stage requirements
OOP	Object-oriented programming – programming paradigm based on the concept of "objects"
PSD2	Revised Payment Services Directive – EU directive to regulate online payment services
REST	Representational state transfer – architectural style for distributed hypermedia systems
SDK	Software development kit – a collection of development tools
SQL	Structured Query Language – language to communicate with the database
Transaction (database)	A unit of work, performed inside the database, which represents any change in the database and can be rollbacked in case of a failure
UI	User interface
URL	Uniform Resource Locator – a reference to a web resource that specifies its location on a computer network
User flow	The path a user takes to reach a certain point inside the app
WWW	World Wide Web
XML	Extensible Markup Language – markup language, which uses schemas to define the data structure

Table of contents

List of figures

List of tables

1 Introduction

Accounting did not just appear out of nowhere. In fact, it took humans over 7000 years to transition from stone tablets to modern day computerized accounting. The earliest mentions of accounting date all the way back to Mesopotamian civilization, which was already using primitive accounting methods to keep records of livestock and crops. In the 1952, IBM released its first large computer and that's when the process of transitioning from paper accounting to computerized accounting began. [1]

Nowadays, the market is filled with different accounting apps. They can either be designed for big enterprises, or for private customers to keep track of their personal funds. Accounting apps for private customers usually deal with things like budgeting, expenses tracking, net worth tracking, or calculating taxes. Most of the time, a financial application is only dealing with one type of financial asset (bank account, stocks, crypto). This leads to an issue, that every bank, crypto exchange or stockbroker has its own app. This fills the user's phone with lots of financial apps as well as separates the financial assets.

In the scope of this thesis, the author is planning to solve these issues, by creating an all-in-one accounting app, which will function with multiple types of assets. Author has divided the thesis into two major parts: the collection and analysis of the literature and the development of the application. Throughout the analysis of the literature, the author analyses similar solutions, and finds the correct technologies and API providers for developing the app. In the development section, the author covers in detail the creation of the backend and frontend applications. The thesis is finished with the analysis of the created app and list of possible improvements followed by the overall conclusions.

1.1 Problem and aim

In the thesis the author is going to address two problems. The first problem being that the advancement of online banking and investments has led towards the creation of huge amount of different financial apps. Nowadays, people can easily have more than one bank account and a few investment accounts simultaneously. All this leads towards having

your device filled with different financial apps, each of which is dedicated to one exact source of funds and requires a long authentication process every time you enter it.

The second problem the author will address is the separation of cryptocurrencies, stocks, and traditional fiat currencies. Usually, bank apps and investment apps differ from each other in many ways. While bank apps are usually easy to understand and use, investment apps on the other hand are filled with tons of useless information and are overall hard to understand for people with no prior experience.

In the thesis, the author aims to solve the upper mentioned problems, by creating an all-in-one accounting app, which would allow users to get basic financial information such as their overall balance, the balance of each asset or transaction history in a matter of seconds without the need to constantly re-authenticate into their financial apps. The author will also combine both cryptocurrencies as well as traditional fiat currencies in one place.

1.2 Motivation

According to a survey made by “The World Bank" organization in 2021, the number of people who possess at least one online banking account is constantly growing even in the lower developed regions and has already reached 76% of the global population [2]. The COVID-19 pandemic made a huge boost towards the adoption of financial services with one-third of adults in developed economies making their first utility bill payments online [2]. Respectable news sources such as “The New York Times” have been pointing out in their articles, that paper money is losing to electronic payments in the long term and cash might disappear in the future [3].

The COVID-19 pandemic did not affect only the traditional banking sector. It also had a huge effect on the crypto and stock markets. During the pandemic, both stocks and cryptocurrencies saw a huge rise in their market prices which lead towards the rise in the popularity of different kinds of online investment providers. Based on the statistics from the “Global Crypto User Index 2021” survey made by the biggest crypto exchange Binance, the number of people using crypto has approximately increased from 5.8 million to 101 million users [4].

The rise of fintech has led towards the creation of thousands of different apps and it is becoming a common thing for people to have multiple banking and investing apps installed, meaning people will inevitably face the issue of having to authenticate into multiple apps to check basic information. This problem will only gain more and more relevancy with time, hence why the author has chosen this topic.

1.3 Research questions

Throughout the research the author will attempt to find answer to the following questions:

- What are the potential benefits of combining all financial applications into one?
- How can the cryptocurrencies and traditional fiat currencies be combined inside one application?

2 Literature review

Before deciding on the methodology for the research, a review of literature must be conducted. Throughout the literature review, similar solutions will be overviewed and best tools and technologies for solving the problem will be found. Various articles, people's opinions, surveys, and statistics will be used as sources of the literature in this thesis.

2.1 Overview of similar solutions

An overview of similar solutions helps to determine the must-have features as well as find possible shortcomings which can be avoided in the app. Before looking for similar solutions the search criteria must be decided upon. Based on the 2 problems which were stated by the author as well as some of the aspects which are considered by the author to be important in a mobile finance app, solutions which follow the next set of criteria must be found:

- Solution supports Estonian banks
- Solution supports crypto accounts
- Solution supports multiple fiat conversion currencies
- Solution has a dedicated mobile app
- Solution is completely free to use

Two similar solutions, which partly match the upper-mentioned criteria were found. Toshl Finance is a personal finance app which advertises itself as a place where you can track your cards and cash in one place. The app supports 14587 different bank and crypto account connections as well as all Estonian banks. Crypto accounts are limited to balances only. The app supports 302 different fiat conversion currencies as well as has Android, iOS and web apps available. The app offers a 30-day free trial after which you will have to pay 40 dollars per year to access the full functionality of the app. The app also offers management of expenses, monthly budget statistics as well as tools for budget planning. [5]

Kubera is a net worth tracker, which allows you to connect banks, crypto exchanges as well as stock brokerage accounts. App offers over 20000 different financial institutions around the world. App also offers the ability to connect crypto wallets as well as the ability to track the price of your home, cars or precious metals to keep track of your net worth. App supports all Estonian banks and has support for almost every fiat currency in the world. App does not have a mobile version and offers a 14-day trial for 1$ after which you will have to pay 150$ per year to access all features. [6]

Having analysed the existing solutions similar to the author's vision, Kubera is highlighted by the author as one of the best finance tracking apps even though the app is built mostly for the American market. Ideas which the author would like to borrow from Kubera would be the ability to add crypto wallets and stock brokerage accounts. Toshl Finance is highlighted by the author for the analytical tools it provides as well as good budget management. Both solutions offer a big amount of functionalities, but both come with a paid subscription, unlike the author’s solution which will be completely free. A comparison between similar solutions can be seen in Table 1.

Table 1. Similar solutions comparison.

Solution	**Estonian banks**	**Crypto support**	**Stocks support**	**Currency conversion**	**Mobile app**	**Free**
Toshl Finance	Yes	Yes (limited)	No	Yes	Yes	No
Kubera	Yes	Yes	Yes	Yes	No	No

2.2 Choice of technologies and tools

The choice of the correct technologies and tools is one of the most important steps in developing any kind of project. Incorrect choice of technologies can slow down the whole development process and cause disturbances in the work of the app. The app consists of two essential parts, the frontend, and the backend. Apps, which consists out of client and server apps are called full-stack apps.

According to an article from cleveroad.com [7], to choose the right technology stack for a full-stack app, the following factors must be accounted for:

- Current expertise – to maximize the quality of the end product, it makes more sense to use the technologies which the developer is familiar with. This also makes the development process a lot faster.
- Scalability – developers should always think in advance when choosing the technology stack. The technology might be fine at handling 1000 users, but this number might always increase and that's when the scalability of the technology starts to matter.
- Development time – to ship the product in the shortest possible time, it might be worth to consider technologies which already have all the needed features and integrations implemented.
- Maintenance – after the product is finished, a lot of unexpected issues might pop up. Technology must allow dealing with these issues without any problems.

2.2.1 Choice of main programming language

Language choice is really tied to framework choice. Nowadays, most of the companies choose a specific language just because the framework was written in it. Alongside frameworks, the 4 aspects mentioned earlier must also be considered. According to latest survey, peoples' choice of languages has not changed by a lot compared to previous years and languages such as Java, C#, Python and JS remain on top [8].

Java is an object-oriented, strongly typed language which comes from the family of C languages. Java is often the choice of companies who develop big and scalable systems which can withstand huge pressure, such as banks. Java's documentation is far from the best, which is compensated for by its huge online community. It has consistently been in the top 5 most popular languages according to many surveys. Regarding web development, Java offers a lot of different frameworks such as Spring, Play, or GWT (Google Web Toolkit). Java's biggest problem is its high memory and CPU (Central Processing Unit) consumption but considering the power of today's computers this can be negated. [9]

C# is always compared to Java because both are descendants of C language and share the same syntax. C# offers a lot of syntax sugar which is not available in Java, which helps shorten the code. C# provides great documentation as well as a big community. C# has a popular backend framework ASP.NET developed by Microsoft. Main disadvantages of

C# would be its dependency on the .NET platform and its relatively higher learning curve compared to Java. [10]

Python is a high-level, dynamically typed language with a huge variety of built-in libraries. It is by far the easiest language out of the 4 and is considered a great option for beginners. Due to its easy syntax and absence of typing, Python takes a lot less time to write code. Python is also a great choice for the backend as it offers frameworks such as Flask or Django. Python has great documentation and a huge online community. Its main disadvantages are bad concurrency and slow speed. [11]

JS (JavaScript) is a high-level, dynamically typed language with an option of OOP (Object-oriented programming). It is often used as a tool to control the behaviour of the webpage. It is relatively easy to learn and has straightforward syntax. It has constantly been the most popular language and has a giant community as well as great documentation written by Mozilla. For backend development JS offers server-side runtime Node.js which allows to run JS outside the browser. Node.js has quite a lot of different backend frameworks, but the most popular are Next.js and Express.js. Biggest problem with Node.js would be its lack of multi-threading due to all processes running in one thread. [12]

Author has a proper level of expertise in all of the above-mentioned languages. All 4 languages have been compared based on the 4 criteria mentioned earlier. Java was selected as the main programming language, because of its great scalability, multi-threading and automatic memory management, as well as its big online community [13]. Java provides developers with a big number of implementations out of the box, making development time much smaller. Even though Java is relatively old, a lot of big companies, such as Netflix, LinkedIn and PayPal build their products using Java [14].

2.2.2 Choice of framework

Choice of framework is even more important than language choice. Typical web frameworks provide users with a web server, database access layer, session management, and thousands of pre-made constructs which allow users to quickly build and deploy their apps. For web development, Java has quite a lot of frameworks such as Play, GWT, Struts, or the built-in Java Servlet functionality, but nothing compares to the Spring framework [15].

Spring framework provides users with an out-of-the-box Security and Authentication systems included in Spring Security module, easy project configuration via property files, convenient abstraction layers for data access, database transaction management, dependency injection, Model-View-Controller pattern implementation and a lot of support functionalities for writing unit and integration tests [16]. Spring framework also provides users with a Spring Boot module, which includes an embedded web server, easy dependency management, automatic configuration, and other features which simplify the development process [17]. Spring framework has great documentation with code examples and a thorough description of every feature.

Spring framework provides a huge number of high-level abstractions for third-party integrations. Whether you need to connect your key-value store like Redis or a messaging broker like Kafka, Spring always has a ready-made integration solution available for everything. The same goes for relational SQL (Structured Query Language) databases like PostgreSQL. Spring combined with JDBC (Java Database Connectivity) drivers allows users to connect to the database just through a few lines in the configuration file.

2.2.3 Choice of data storages

Any app depends largely on data storages as they allow the app to memorize important data such as users' logins and passwords. Data storages are divided into 2 categories: non-relational NoSQL storages and relational SQL storages. SQL databases are usually used for big amounts of data which needs to be filtered or sorted, and NoSQL databases are used for unstructured and large data objects such as chat logs or for quick-access data.

Based on the latest developer survey, among SQL databases PostgreSQL, MySQL and SQLite were the most common choices for developers [8]. Among NoSQL databases MongoDB and Redis came on top. Both persistent quick access storage as well as relational database are needed for the project meaning that suitable SQL and NoSQL databases have to be found.

All SQL databases share the same syntax which follows the ISO/IEC 9075 standard. Some of them provide users with custom functions and data types such as JSON (JavaScript Object Notation) or Array.

- MySQL is a free open-source database, which is often used in PHP apps. It is highlighted for its ease of use, low price for enterprise edition and relatively higher speed compared to other databases. Compared to others, it does not offer custom data types, has worse indexing, and does not have a dedicated programming language for procedures like PL/SQL or PL/pgSQL. [18]
- PostgreSQL is a free open-source database. It is often chosen by big companies for its ability to work under huge loads. It provides users with custom data types such as array or JSON, as well as a lot of custom functions. It also offers great indexing. Its weak sides compared to MySQL would be slower speed of reading and a higher learning curve. [18]
- SQLite is a free open-source database. SQLite differs from others by not having a server. SQLite reads and writes data straight to the disk, which simplifies the configuration and removes the need for configuring the server. The absence of a server also comes with its disadvantages, such as a lack of configurable user roles and permissions. [18]

NoSQL databases can be divided into two categories: key-value stores and document stores. For the current project, the author requires a key-value store.

- MongoDB is a source-available document-oriented storage with options of free community edition as well as paid cloud solution. MongoDB offers schemeless data storing without predefined columns. MongoDB is a great option for people who don't care much about the structure of the data. For caching and quick access of data, MongoDB is not suitable. [19]
- Redis is an in-memory key-value storage which is often used as a cache. Unlike local in-memory caches, Redis is persistent, which means that if the app suddenly restarts or dies, cache data will not be erased. Redis supports various data types such as strings, lists, maps and sets and can also be used as a messaging broker. [20]

Among the relational SQL databases, the choice fell on PostgreSQL, due to its great performance as well as the fact that it has custom data types and functions. For NoSQL database, the choice is pretty much straightforward. As far as persistent key-value storage is concerned, Redis is the best option available on the market offering developers great

low-latency caching. Companies like Twitter and Pinterest are using Redis as their main cache provider to speed up their apps [21].

To easily create tables and rollback changes as well as have a full history of database changes, a database version control library for Java has to be selected. The choice lies between Liquibase and Flyway, as these are the two biggest and most popular projects, and both are free. Liquibase schemas are written in the form of XML (Extensible Markup Language) file entries and Flyway schemas are written in Java and SQL similarly to Entity Framework in C#. Liquibase was chosen because database migrations are easier to create using Liquibase compared to Flyway.

2.2.4 Frontend technologies

According to the scope, a mobile app which can be run on iOS and Android devices must be built. Nowadays, there are multiple ways of developing a mobile app.

One possible option is to do a native app for a specific device (iOS or Android) using a platform-specific language such as Swift. This method is really time consuming and only allows you to create an app for one device, so all the code has to be written twice for both platforms. On the positive side, it gives more flexibility and apps usually have better performance compared to hybrid apps.

Another way of creating a mobile app is by using hybrid app development frameworks. It allows to simultaneously create iOS, Android and even web apps. Most notable hybrid app frameworks are React Native, Flutter and Ionic.

- Flutter is a cross-platform UI (user interface) SDK (software development kit) created by Google. Development in Flutter requires knowledge of the Dart. Flutter is the fastest performance wise compared to Ionic or React Native. Flutter is relatively new, so the community size is small, but its popularity is constantly rising. [22]
- React Native is a cross-platform UI framework created by Facebook. React Native is similar to the JS framework React, except it does not manipulate the DOM. With React Native, default CSS (Cascading Style Sheets), HTML (Hypertext Markup Language) and JS techniques cannot be used, so additional learning is

required. Since React Native uses mostly standard iOS and Android controls, it offers great performance. [22]

- Ionic is an SDK for hybrid mobile app development, which was originally built on top of Angular and Cordova. Ionic offers traditional frontend app development experience mixed with native functionalities. According to benchmarks [23], Ionic has similar performance as React Native in almost all the default aspects such as boot time, smooth scroll, and so on. CPU consumption and energy impact, on the other hand, is much lower for Ionic compared to React Native. [22]

Since the author does not have any prior knowledge of mobile development and React Native and Flutter both require learning new techniques, it was decided to choose Ionic as it allows to transform a simple JS frontend app into a mobile app as well as easily utilize the native mobile functionality. Capacitor runtime, which is used as the base for Ionic can be used on its own and since custom UI components provided by Ionic are not needed, Capacitor will be used together with a JS framework for the creation of mobile app.

Easiest way to develop an app together with Capacitor is by using one of the top 3 JS frameworks which are Vue.js, React, and Angular. With the help of additional libraries, all the frameworks offer all the default functionalities such as routing, data store, two-way data binding, reusable components, lifecycle methods, and so on.

- React is a free open-source JavaScript framework for building user interfaces created by Facebook. It is used to develop single-page or server-rendered apps. It is lightweight, highly performant and provides great documentation. Out of all JS frameworks, React is undoubtedly the most popular one with the biggest online community. [24]
- Angular is one of the oldest JS frameworks, originally developed by Google. Angular is based on TypeScript. Angular is the hardest out of 3 and has quite poor documentation. [24]
- Vue.js is the youngest out of 3 and was developed by a single developer Evan You. Vue.js is driven largely by the community, which creates a lot of libraries which help other developers speed up the development. In comparison with others, Vue is relatively easier to learn since HTML, CSS and JS are all part of one component. [24]

All the JS frameworks share the same functionalities required for development of a single-page web app. Vue was chosen by the author due to its great documentation, huge amount of tools and additional libraries, easily readable code, high performance, as well as a decent level of previous experience using this framework [25].

Easiest way to develop a web app is by using the customizable ready-made components, as they allow developers to save a lot of time and offer out-of-the-box mobile responsiveness, which is usually hard to achieve when done manually. Vue.js offers quite a lot of component libraries such as Vuetify, Quasar, or Ionic [26]. Vuetify was chosen due to the big number of different components, as well as good design, and ease of integration with a Vue project.

2.2.5 Development tools

Every developer has a set of tools, which help speed up the development process. Development tools include such things as IDE (Integrated development environment), version control systems, and so on. For developing the project, an IDE with Java and JS support, a version control system, code repository and container virtualization software have to be found.

According to the latest survey, Visual Studio Code and IntelliJ IDEA are the 2 most popular IDEs among developers [8]. Both have support for custom plugins, code autocompletion, debugging, integrated version control, database administration, and many more, but since Java was chosen as the primary language of development, the choice here is obvious, because IntelliJ was originally developed specifically with Java in mind and almost 75% of Java developers use it according to a survey [27]. IntelliJ also allows development of JS apps with the help of additional plugins.

Version control systems are usually needed for enterprises to share code between team members, but they are also useful for single developers as it allows them to watch history of code changes, do rollbacks to previous versions and many other things. There are many different version control systems and all of them share the same set of functionalities, so it was decided to choose Git because of the author's previous experience with the system.

All code repositories provide default functionalities, such as merge request reviews, CI/CD pipelines, and others. In the scope of the thesis project, an online repository is needed only to share the code between devices. All of the author's non-commercial projects have already been stashed on GitHub, so it was decided to put the project there.

Finally, is container virtualization software, which allows developers to greatly increase their productivity. Essentially, this kind of software separates the app from the infrastructure (databases, messaging brokers, microservices, and others). Instead of running the software like a database manually, users can just run the container which already has the necessary software on it and connect to it through a shared port. There are several alternatives, but Docker is free and has a huge community as well as good documentation.

2.3 Choice of API providers

To develop the project, online banks and crypto exchanges have to be researched. Communication with third-party services can be accomplished by using the API. An API is a mechanism, which enables the communication between two applications through a specific protocol. Every API provides a contract, which describes what requests can be made and what responses will be received. There are multiple types of APIs such as REST, SOAP, gRPC, or WebSocket. [28]

According to "2022 State of the API Report" [29] made by the Postman, REST (Representational state transfer) is by far the most used API type with 89% of people claiming they have used it. APIs can either be private, public, or available only to partners. Public APIs are used by companies to provide certain functionality to the public, while private and partner APIs are only available for authorized users [30].

2.3.1 Banks

As far as banks are concerned, Estonian banks which follow the open banking concept have to be found. Open banking is a relatively new term among banking services which is aimed towards providing verified companies with an API, which would allow them to do default banking operations such as payment initiation, account information querying or balance check [31]. Open banking APIs in Europe follow the Berlin Group standard, which is based on the requirements of PSD2 (Payment Services Directive). PSD2 sets

specific technical requirements for the API interfaces [32]. To be able to use the Open Banking API from one of the Estonian banks, a valid license from the Estonian Finance Inspection must be obtained [33]. This makes it impossible to implement the OpenAPI manually as the license can only be obtained by a financial institution.

The only way to receive the required banking data is by using a third-party banking aggregator, which already has the required PSD2 license and can provide secure connections to the banks without any specific licenses. According to openbankingtracker.com [34], Estonia has 9 bank providers with OpenAPI connection. Based on that list as well as some of the other banks which are popular in Estonia, the following banks will be added into the app: Swedbank, Revolut, SEB, N26, Luminor, COOP, LHV, Wise and Citadele.

By using the openbankingtracker.com, the aggregators which support the above-mentioned banks were found. The following 3 aggregators have been brought out by the author:

- Salt Edge is a UK aggregator which supports 5,147 connections from 53 different countries. Salt Edge implements all the default OpenAPI functionality, such as payment initiation, account balance, and transaction querying. To receive unlimited access to all features, a client agreement must be signed with the aggregator and a technical review must be passed. [35]
- Enable Banking is a Finnish aggregator which supports 2515 connections from 22 countries. Enable Banking implements all the OpenAPI functionalities. By default, only the sandbox is available and to receive access to full functionality a contract must be signed. [38]
- Nordigen is a Latvian aggregator which supports 2392 banks. Unlike its counterparts, it does not support payment initiation. Nordigen is Europe's first free PSD2 data API provider. [41]

Out of all the options, Salt Edge stands out as the most finessed out of 3, but since it requires contract signing, it was decided to select Nordigen since it's the only free option. Nordigen lacks payment initiation, but for the current project, only balance and transactions data are required, so this issue can be ignored. Comparison of bank aggregators can be seen in Table 2.

Table 2. Bank aggregators comparison.

Aggregator	Payment initiation	Balance querying	Transactions querying	Has all Estonian banks	Free
Salt Edge	Yes [35]	Yes [35]	Yes [35]	Yes [36]	No [37]
Enable Banking	Yes [38]	Yes [38]	Yes [38]	Yes [39]	No [40]
Nordigen	No [41]	Yes [41]	Yes [41]	Yes [42]	Yes [43]

2.3.2 Crypto exchanges

Nowadays, fiat currencies such as Euro or Dollar are not the only possible stores of value or payment options. With the advancement of technology, new fully digital currencies driven by blockchain technology have appeared under the name of crypto currencies. Cryptocurrency differ from classic fiat currency in many things. Unlike fiat currency, cryptocurrency is not backed by the government or any physical value and are created by computing. Crypto transactions are done between two devices, meaning no intermediary such as a bank is needed. [44]

Cryptocurrency can either be received from someone else through a blockchain transaction on your personal wallet or bought on one of the crypto exchanges with fiat money, which is much easier. Crypto exchanges work similarly to a stockbroker, giving you the option to obtain a certain asset, as well as trade it. Unlike banks, crypto exchanges are a relatively new concept which first appeared in the beginning of the 2010s. Nowadays, there are over 200 different exchanges available and most of them have a free API connection. [45]

APIs of crypto exchanges do not follow any specific protocols such as PSD2, like in case with banks, so the APIs can vary by a lot. They also have custom rate limits between them, meaning some exchanges can allow infinite requests while others may allow you to only do 1000 requests per minute. Another bad thing about crypto exchanges is that most of them do not provide OAUTH2 secure authentication. Instead, they provide insecure authentication based on an API key and an API secret combination, meaning this information will have to be accepted from the user and stored somewhere, leading to the risks of this information being leaked.

In the scope of this thesis only the 3 biggest crypto exchanges will be implemented. Based on information from the biggest crypto market analysis provider coinmarketcap.com, the top 3 crypto exchanges currently are Binance, Kraken, and Coinbase [46]. They are ranked based on the amount of traffic, liquidity, and trading volumes as well as their secureness. Comparison of the top 3 crypto exchanges can be seen in Table 3.

Table 3. Top 3 crypto exchanges comparison.

Crypto exchange	**Balance querying**	**Transactions querying**	**OAUTH2 authentication**	**API rate limits**
Binance	Yes (limited) [47]	Yes [47]	No [47]	1200/12000 per minute (IP limit) 180000 per minute (User limit) [47]
Kraken	Yes [49]	Yes [49]	No [49]	Every API user has a "call counter" which increases by 1 or 2 on each call [49]
Coinbase	Yes [48]	Yes [48]	Yes [48]	10000 per hour [48]

3 Methodology

The methodology for this research will be divided into three parts. In the first part, the requirements for the experiment will be gathered. In the second part, the experiment will be conducted according to the requirements. In the end, the results of the experiment will be analyzed, and a summary of the research results will be provided.

3.1 Method of collecting data

Research data is divided into two categories: primary and secondary data. Primary data is new information collected by the author, whilst secondary data is taken from already existing sources. There are also different techniques of gathering and analyzing the data.

Qualitative data collection method helps to gain the human perspective on the problem. It is based on nonmaterial factors such as experiences, opinions, and thoughts. Qualitative data includes things such as user interviews, case studies, and observations.

Quantitative data collection method uses numerical data and graphs to receive precise results with less bias. It is usually expressed in the form of digits, percentages, and ratios. Typical examples of quantitative data are closed ended surveys, experiments, and statistical data reviews. [50]

In the scope of this thesis, a mixed approach has been used. Throughout the process of reviewing the literature for the research, a comparative analysis has been made based on the articles, people's opinions, surveys, and statistics which were gathered by the author. User surveys and interviews will be conducted by the author to gather opinions on the potential MVP requirements as well as to get feedback on the results of the experiment.

3.2 Experiment

To solve the problem described in the thesis, an experiment must be conducted. The aim of the experiment is to create a mobile app. This process will consist of creating frontend and backend apps. The technology stack for the development was decided throughout the process of reviewing the literature. To understand which features the app must implement, a user survey will be conducted. The development process of both apps will be described

in detail throughout the thesis. After finishing the experiment, the solution will be tested to prove that the author has succeeded in solving the problems. In the end, test results description and research summary will be made.

4 Experiment requirements gathering

Requirements describe obligatory features, which must be implemented throughout the development process, and they are divided into two categories, functional and non-functional requirements. In agile development, requirements are created in the form of user stories. A user story usually follows the format of "persona + need + purpose". [51]

To formulate the requirements, the potential users were given a questionnaire to find out what are the most important features in their opinion. The target group for this questionnaire was limited to people from Estonia who have multiple financial applications (both crypto and bank) on their mobile device installed. 10 people were presented with a Google Forms questionnaire, which consisted of a list of potential functionalities with three options of importance: "must-have", "nice to have" and "not important". Results were automatically analysed by Google Forms and presented in a form of bar chart. An overview of similar solutions which was done by the author earlier is also useful when formulating the requirements as some of the ideas can be borrowed from the existing solutions.

4.1.1 Functional requirements

Functional requirements describe what a product must do, and what features and functions it must implement. Without the fulfilment of these requirements, the app will not be able to function properly. [52]

Based on the results of the questionnaire as well as the analysis of similar solutions, the following set of requirements has been created by the author:

- As a user, I want to log in or register, so I can enter the app
- As a user, I want to logout, so I can leave the app and be sure no one gets access to my financial data
- As a user, I want to connect my Estonian bank account, so I can track my funds and transactions
- As a user, I want to connect my crypto exchange, so I can track my funds and transactions
- As a user, I want to see my overall balance summed from all accounts, so I know how much money I have on all accounts

- As a user, I want to see the balance of each asset separately, so I know how much of every asset I have
- As a user, I want to see balances in the conversion currency I selected, so I don't need to convert them manually
- As a user, I want to see the list of my latest transactions, so I can track my fund movements throughout all accounts at the same time

4.1.2 Non-functional requirements

Non-functional requirements are tied to functional requirements. They describe the overall usability of the app, which is affected by things like performance, user experience and so on. The app can function without the fulfilment of these requirements. [52]

To formulate the non-functional requirements, results of a questionnaire as well as an article from indeed.com [53] about the most popular non-functional requirements will be used. Based on that information, the following set of requirements has been created by the author:

- User balances should not take more than 2-3 seconds to be aggregated (less than a second if cached)
- App should force the user to create a secure password
- Crypto exchange credentials and user passwords should be encrypted
- User should be able to switch language and currency
- User's language and currency choice should be saved permanently
- User should get notified if an error has happened on the server
- App should work the same way on both Android and iOS platforms

4.1.3 Limitations

Since the author's time to write the thesis is limited to a certain time window, some of the requirements described earlier will come with the following limitations:

- App will only support English, Estonian and Russian languages
- App will only provide 2 conversion currencies (Euro and Dollar)
- Due to a huge number of crypto exchanges only the 3 biggest will be implemented
- Since the app is focusing on Estonian customers, only Estonian banks will be supported

5 Development of the application

In this section, the process of creating the app will be described in depth. This section is divided into two parts: the backend and the frontend. Backend part describes the process of working with the data storages, methods of securing the app, cron jobs, project structure as well as the testing process. Frontend part describes the process of creating a single-page web app (components, router, store, communication with the backend), converting the web app into a mobile app using the Capacitor.js framework and creation of views.

5.1 Backend application

The backend part of the app is responsible for the whole logic behind the frontend app. Spring backend apps can be built either by using the MVC (Model-View-Controller) pattern or by using the REST pattern. Since it was decided to make a mobile app, the REST pattern will be used to create the HTTP (Hypertext Transfer Protocol) endpoints for the communication with the mobile app.

5.1.1 Database and cache

Development of backend apps usually begins with creation of the entity relation diagram. Thoroughly thought-out database models can simplify the coding process in the future, and badly made ones can cause a lot of issues later. Database creation usually begins with a user table (“users”), which is accountable for holding user information such as login and password. User banks (“user_bank_accounts”) and crypto exchanges (“user_crypto_exchanges”) which the users have connected have to be tracked, so two separate tables are needed for that. Both tables are connected to a table (“funds_source”), which holds information about financial institutions used in the app.

Users can add a specific financial institution only once. Bank accounts can have multiple sub-accounts, so an additional table (“user_bank_sub_accounts”) is needed to track them. To have a full user transaction history as well as to be able to fetch transactions faster, a separate table (“user_transactions”) is created. Both crypto and bank transactions are held in one table and specific description details are held in the form of a JSONB object in the “description” field. One table (“asset_metadata”) is separated from the others as it is not

related to the overall logic of the app and is only needed to fetch asset rates as well as their metadata such as their name or picture URL (Uniform Resource Locator) in case the cache does not have these values already. Entity relation diagram is shown in Figure 1 and was created with the help of IntelliJ IDEA diagram visualization feature available in database tools. Tables “databasechangelog” and “databasechangeloglock” were excluded from the schema as they were not related to the logic.

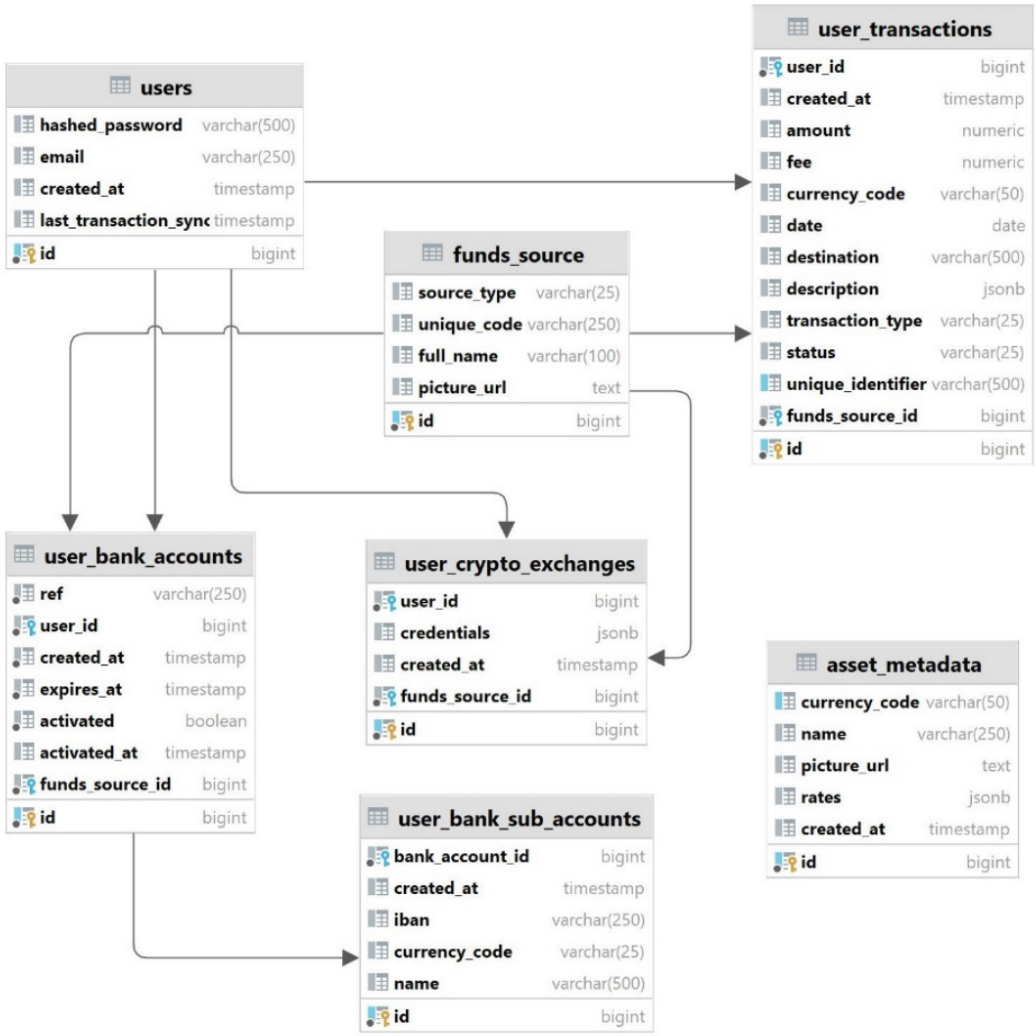

Figure 1. Entity relationship diagram (ERD)

Liquibase was previously chosen as the database version control library for Java. This allows creating the whole database structure with a few XML entries. XML is later translated into the SQL language of the developer’s choice, which is PostgreSQL in the author’s case. History of changes is held in the “databasechangelog” table, which is automatically generated by the Liquibase. Figure 2 displays creation of a “users” table through an XML entry.

```
<changeSet id="202202241737" author="Nikita Viira">
    <createTable tableName="users">
        <column name="id" type="bigserial" autoIncrement="true">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="hashed_password" type="varchar(500)"/>
        <column name="email" type="varchar(250)"/>
        <column name="created_at" type="timestamp"/>
    </createTable>
</changeSet>
```

Figure 2. Liquibase table creation

Since the app will be developed using the reactive programming pattern with the help of the Project Reactor, a special reactive database driver for PostgreSQL is needed. Regular JDBC drivers are not suitable for this because they are blocking, meaning each connection to the database needs a dedicated thread. Thankfully, a fully reactive database driver was created for Java under the name of R2DBC (Reactive Relational Database Connectivity), which provides a reactive API which works together well with Project Reactor. Spring provides a way to easily integrate R2DBC into the app by using the "spring-boot-starter-data-r2dbc" library. R2DBC is enabled by using the *@EnableR2dbcRepositories* annotation inside the Spring main class and can later be accessed in the repository class by injecting the *R2dbcEntityTemplate* bean.

To speed up some of the processes in the app and prevent constant API and database querying, it was decided to use Redis cache. Things such as user balances, currency rates or API access tokens can be held in the cache because they are needed all the time but querying them from the database or third-party API may take too much time, and this affects user experience in a bad way. To use Redis in a reactive Spring app, the non-blocking Lettuce driver must be used. Spring provides users with a "spring-boot-starter-data-redis-reactive" library, which allows users to easily integrate reactive Redis into the app.

Usage of R2DBC and its configuration can be found in Appendix 2. Usage of reactive Redis and its configuration can be found in Appendix 3.

5.1.2 Security

To authenticate users into the app, the combination of the login and password combined with a JWT (JSON Web Token) will be used. It is important to encrypt a user password before inserting it into the database to prevent anyone from ever using it. One-way

hashing should be used for passwords because it makes sure no one will be able to decrypt them. Password hashes are generated by the author using PBKDF2 function in combination with HMAC SHA1 from the "javax.crypto" Java library [54].

JWT is an open standard which was created to securely transmit JSON data in the form of a long string without spaces. JWT can be trusted because it is signed by a secret combination which is hidden from everyone. JWT is made of 3 important parts [55]:

- Header, which is made of 2 parts: type of token and the algorithm used to sign the token (SHA256, RSA). [55]
- Payload, which holds the claims. Payload usually contains information such as expiration time of the token, token issuer name and so on. Developers can also add their own information such as a user identifier or email inside the payload. [55]
- Signature, which is used to verify the message was not changed throughout the request. [55]

After the user has entered his login and password, JWT is generated and passed to the frontend app and held in the local storage. This allows users to stay logged into the app until the JWT eventually expires or a user logs out manually erasing the JWT from storage. It is the responsibility of the developer to verify JWT on each request to an authorized endpoint. Backend must make sure that JWT was signed with the correct secret combination and algorithm as well as that it has not expired yet.

JWT is passed to the backend from the frontend through the "Authorization" HTTP header during a web request. To verify JWT, the app must intercept it at the beginning of the request. To do this in a Spring app, a method argument handler can be used. Spring provides *HandlerMethodArgumentResolver* interface, which can be manually implemented for a specific controller argument so that every time an endpoint with this specific argument is called, the app can intercept the request and all its data, such as headers, and do the verification of the JWT.

Implementation of the authorization can be found in Appendix 4.

Due to the issue with crypto exchanges still using API key and API secret authentication, these credentials must be saved in the database to be able to do requests in the future. This

is very sensitive information meaning it must be encrypted before putting it into the database. One-way hashing is not suitable for this, because encrypted credentials have to be decrypted later. Safe way to encrypt a string is to use the combination of a secret key and salt. Both salt and secret key are held as system properties and hidden from code. For the encryption, the AES256 algorithm from the "javax.crypto" Java library is used by the author [56].

5.1.3 Cron jobs

Cron job is a task scheduling utility for UNIX-based systems (Linux). Tasks are run periodically at a specified date and time. This is a very useful utility for frequent tasks, which must be executed every day. In the scope of the thesis, this functionality is needed to do periodic updates of currency rates, daily transaction syncs as well as deleting the expired bank accounts which were not activated. A cron job consists of the cron expression specifying the date and time it must run at and the command they must execute. Cron expression consists of 6 positions: second, minute, hour, day of month, month and day of week.

Spring framework provides built-in scheduling functionality, which can be found in the "spring-context" module. Scheduling can be enabled in the Spring app by using the *@EnableScheduling* annotation inside the main class. To create a cron job, a method which will run periodically has to be annotated with *@Scheduled* annotation which takes cron expression string as its parameter. Figure 3 displays an example of a Spring cron job which runs every 20 minutes.

```
@Scheduled(cron = "0 0/20 * 1/1 * ?")
public void loadLatestCurrencyRates() {
  if (cronTasksEnabled) {
    metadataService.loadLatestAssetMetadata().subscribe();
  }
}
```

Figure 3. Spring cron job

5.1.4 Project structure

Apps must be designed so that if you open them later to add a new feature or fix a bug, you will not run into the issue of having to spend hours to understand the logic behind some of the functionality. Most developers who use Spring framework use the three-layer

architecture to build their apps. According to this architecture, the whole code is divided into 3 layers:

- The web layer, which is the topmost layer of the app. It accepts and validates user input, returns data to the client in a readable JSON format, handles exceptions thrown by the lower levels, as well as handles authentication of the requests.
- The service layer, which acts as the brains of the app. It implements business logic, meaning all the calculations, evaluations and data processing is done inside this layer. It is also responsible for processing the data between the data access layer and the web layer.
- The data access layer, also known as the persistence layer, is responsible only for the interactions with the database, such as saving or querying the data.

All the layers will be built using the reactive programming paradigm. In this paradigm developers use declarative code (like functional programming) to create asynchronous processing pipelines. It is based on publishers and subscribers, where a subscriber subscribes to the publisher and gets the data as soon as it is available. This allows apps to save on resources and withstand huge pressure. Reactive programming allows to write readable fully asynchronous code and comes as a substitution for the Java concurrency utilities such as Futures, which are quite hard to read and understand. Project Reactor is the best Java reactive library available. It provides users with two reactive APIs, Flux, and Mono. Flux is used to describe a collection of reactive elements and Mono describes one or zero reactive elements. In combination with Spring WebFlux module, users are provided with an asynchronous event-loop web server Netty, and a reactive WebClient which is an interface for performing web requests and comes as a substitution for the Java's built-in HttpClient and Spring's RestTemplate.

Appendix 5 displays implementation of WebClient.

Implementation of the three-layer architecture usually begins with the data access layer. With the help of the database table schema which was created earlier, all the required entities can be created. A database entity in Java is usually a class marked with @Table annotation, which describes all the fields and their data types inside the table by marking fields with the @Column annotation. These entities are later used by the Spring Data

module to automatically map all the database rows to an entity. Figure 4 displays an example of a Java database entity.

```
@Table("users")
@NoArgsConstructor
@Data
public class User {
  @Id
  public Long id;
  @Column("hashed_password")
  public String hashedPassword;
  @Column("email")
  public String email;
  @Column("created_at")
  public Instant createdAt;
  @Column("last_transaction_sync")
  public Instant lastTransactionSync;
}
```

Figure 4. Java database entity

Entities by themselves are unable to do any actual database operations. Repository pattern will be used to implement all the required database operations. In the repository pattern, developers must create a specific class which will implement all the operations (such as read, write, or delete) on a particular entity. Easiest way to do this with R2DBC is to inject the *R2dbcEntityTemplate* bean inside the repository class, which has all the default database CRUD (Create, Read, Update, Delete) operations available and automatically maps database rows into the needed reactive collection (Flux or Mono) based on the entity class provided.

Repository implementation can be found in Appendix 2.

The second and most important layer of the app is the service layer. It implements the business logic of the app. Services inject all the needed dependencies and do the required calculations. Service layer is responsible for opening of the database transaction during the beginning of an operation and interrupting it in case an error in business logic has happened. Thanks to the "spring-tx" module, which comes together with Spring Boot, transaction management is done automatically. Method has to be annotated with *@Transactional* annotation to initiate a transaction. Data transfer between the data access layer and service layer is done through the entity objects. To transfer data to the web layer, separate DTOs (Data Transfer Object) must be created. Separate “mapper” classes are created to do the assignment of all the required fields for the DTOs.

At the top of the project architecture sits the web layer (also known as the presentation layer). This layer is responsible for performing the authentication of the web requests made to the server, conversion of JSON data into Java objects, and vice versa, as well as validation of the request body fields. The Web layer in Spring apps is expressed through controller classes. Nowadays, controllers are created using the REST architecture, meaning other systems can request the required resource through one of the predefined URLs by doing an HTTP call. REST uses HTTP methods such as GET, POST, DELETE and PUT to perform operations. REST APIs are built using endpoints, which are resources with a predefined URL and HTTP method.

REST controllers in Spring app can be implemented using the Spring web module. To create a controller, a class must be annotated with the *@RestController* annotation and filled with the needed endpoints. Endpoint in Spring is a method annotated with *@RequestMapping* annotation with a specific URL and HTTP method. Spring also provides automatic conversion of JSON into Java objects with the help of Jackson library. Jackson library in Spring is configured through the global *ObjectMapper* bean in web configuration. To perform validation of the user data, the Hibernate Validator library is used, which implements the JSR-380 (Java Specification Requests) validation specification. Endpoints can be either authorized or not. Endpoint authentication is described in the authorization section. Figure 5 displays an example of a Spring REST controller.

```
@RestController
@RequiredArgsConstructor
public class UserController extends BaseController {
  private final UserService userService;

  @RequestMapping(value = "/user/login", method = POST)
  public Mono<LoginResponseDto> login(@RequestBody LoginRequestDto
loginRequestDto) {
    return userService.login(loginRequestDto);
  }
}
```

Figure 5. Spring REST controller

Web layer is also responsible for error handling in the app. Web layer needs to be able to intercept an exception thrown by the app and display it to the client in a readable way. Since the app is multilingual errors have to be translated as well. Spring Web provides *@RestControllerAdvice* annotation, which can be used to mark class as the main

exception handler of the app. Translation texts are held inside special Java ".properties" files. To fetch translations from the properties, the *ResourceBundleMessageSource* bean must be implemented, which can automatically fetch the needed error message based on the locale. To understand which language the user has chosen in the app, the "Accept-Language" header will be sent to the backend with each request, so the error handler can then fetch the header from the request and respond with the translated error.

Exception handler implementation can be found in Appendix 6. Figure 6 shows an approximate visualization of the application's architecture and was created with the help of edrawsoft.com.

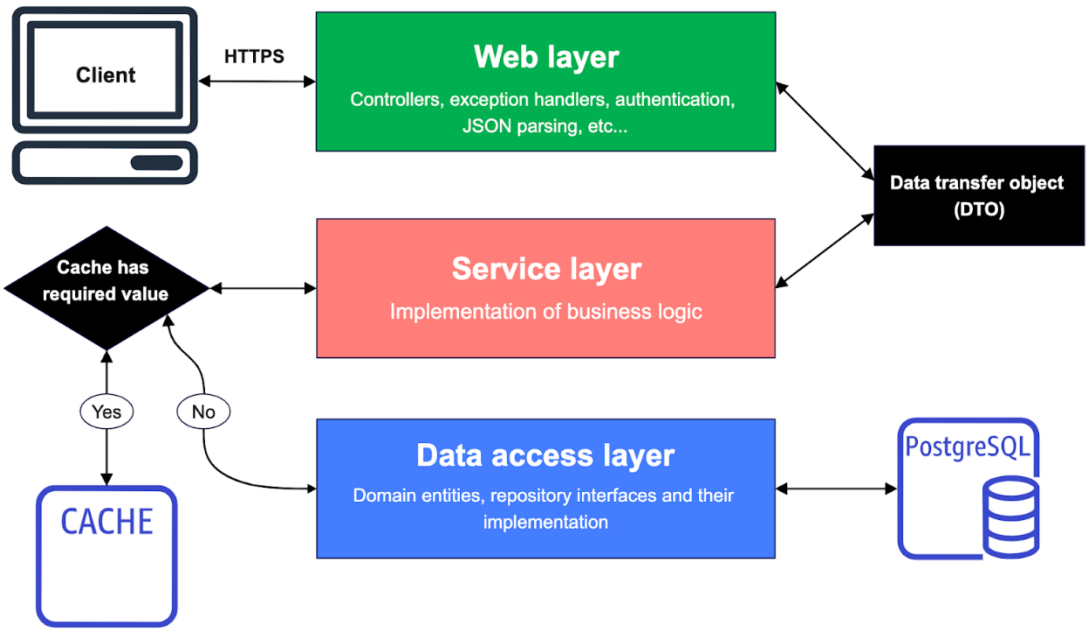

Figure 6. Backend application architecture diagram

5.1.5 Additional project tools

To make development easier, a few additional tools can be used. For the dependency management and building of the project, Gradle will be used. It is a build automation tool based on the JVM (Java Virtual Machine) language Groovy. It allows developers to configure things such as Java version, check style, tests and dependencies. Gradle has great support in IntelliJ IDEA and allows running Gradle commands straight from the IDE instead of running them manually from the console.

Java is disliked by many because of all the boilerplate code developers must write repeatedly. Therefore, Lombok is a "must have" tool. It is an annotation-based library created to minimize the amount of boilerplate code. By putting a *@Data* annotation on top of the class, Lombok automatically generates *ToString* methods, getters and setters, constructors and *equals* and *hashCode* methods. For easier dependency injection inside the services, a *@RequiredArgsConstructor* annotation can be used, which automatically creates a constructor for all the fields which are marked with the "private final" Java keywords. Adding *@Slf4j* on top of the class adds a logger. IntelliJ IDEA has great support for Lombok and adds code autocompletion for the generated methods.

5.1.6 Testing

It is very important to test the backend side of the app. This allows to spot issues and corner cases at the early stages of development and quickly fix them. Tests also show the original intention behind the logic, so that if the logic changes later the test will fail, indicating that the logic has changed. Tests are usually divided into two categories: unit and integration. Unit tests are used to test a small piece of code to identify issues at the code level. Unit tests are fast and easy to maintain, hence they are usually done in huge quantities. Integration tests are used to test the overall logic of the app. Integration tests must be as close to the real app as possible, meaning actual database calls and API requests must be done.

To test a reactive Spring app, the following tools are used:

- *Spring Boot Starter Test* library provided by Spring Boot for automatic configuration of tests
- *Reactor Test* library provided by Project Reactor to test reactive pipelines
- *MockServer* library, which runs a separate testing server and helps emulate actual requests to a third-party APIs with predefined responses
- *Mockito* library to mock database and service calls in unit tests
- *AssertJ* library, which provides a huge number of custom assertions to simplify the testing

The test class is annotated with a *@SpringBootTest* annotation from the starter library, which automatically injects the whole Spring context allowing to inject dependencies into the test class. In unit tests, calls to an actual service must be avoided, so Mockito is used

to return a predefined answer in case of a call. Figure 7 displays an example of using Mockito.

```
Mockito.when(accessTokenCache.getAccessToken())
  .thenReturn(Mono.just(new CachedAccessToken("token", "refreshToken",
Instant.now())));
```

Figure 7. Example of mocking with Mockito

To test reactive pipelines, the *StepVerifier* interface from the *Reactor Test* library is used, which allows to transform a reactive collection like Flux into a series of steps, each of which can be asserted. For integration tests, Docker containers are used to create the whole development environment. To initialize the testing environment, the "docker-compose.yml" file was created which allows to start the whole environment with one console command. A separate Spring configuration file ("application.properties") was created for integration tests to separate configuration between the testing and production environments. To test requests to third-party APIs, a Docker container with *MockServer* and the "mockserver-netty" Java library are used to emulate API responses. JSON response and request files for tests must be provided manually, so they were saved in the test resources folder.

"docker-compose.yml" file configuration can be found in Appendix 7.

5.2 Frontend application

Frontend displays backend information in a form of visual representation. Throughout the literature review, it was decided to first create a single-page web app using Vue.js and then attach the Capacitor cross-platform runtime to transform it into a mobile app.

5.2.1 Single-page web application (SPA)

Single-page apps (SPAs), as the name suggests, only show users one page which is then dynamically changed based on the user interactions. This means that the user does not need to constantly reload the page whenever a redirect is needed. Compared to traditional multi-page apps, SPAs are a lot faster, because resources are only loaded once. These kinds of apps are also a lot easier to develop and they are the ideal choice for native mobile development. There are a few disadvantages to this developing method, such as bad SEO (Search Engine Optimization), meaning it is harder for people to find this website in search engines like Google. Another big disadvantage of such websites is that

they take a lot of time for the initial load, because the frameworks must load all their code into the user's device. [57]

Development of JS apps depends a lot on the usage of libraries because they allow developers to avoid "reinventing the wheel". To connect a library to a JS app, a line which specifies the name and version of the library must be added to the "package.json" file, which is generated by the JS package manager NPM (Node Package Manager). For developing the frontend app, the following libraries (and some other less important ones) are used:

- "vue" - the core library of the Vue.js framework. Provides all the framework's key functionalities. Even though the new improved Vue version 3 exists, the author will use Vue version 2, because he is more familiar with it.
- "vue-router" - library, which provides easy-to-configure routing functionalities. This allows developers to configure the navigation on the website.
- "vuex" - library for state management, which is now deprecated in favour of the "pinia" library. State is used as a global storage to pass data between components.
- "vue-i18n" - library for internationalization of the app. Provides an easy way to translate static text with separate JSON message files for each language.
- "axios" - HTTP client for Node.js. Allows to make API calls to the backend.
- "typescript" - library, which adds an ability to write typed JS code using TS (TypeScript) language. TS provides default data types like "string" or "number" as well as the ability to create custom data types.
- "vuetify" - UI library for Vue, which provides handcrafted customizable components based on the "Material Design" design language made by Google.
- "vue-class-component" - library, which allows the creation of class-based Vue components.

5.2.2 Components

JS frameworks are usually called "component-based" frameworks. Components can be seen as classes in the OOP languages. Components represent a tree-like structure where "parent" components can have multiple "child" components. They allow developers to split UI and logic into smaller isolated pieces, which provides greater code readability and shortens the code base. In Vue, data can be passed between components using the "props". Since it was decided to use Vuetify, all of the typical web app components such

as navigation bars, footers, icons, images, loaders, form inputs, buttons, alerts and many others are already provided by the library.

Each Vue component is created inside a dedicated file with ".vue" extension using the SFC (Single-File Component) pattern, where HTML, CSS and JS (or TS) are all located inside one file. Components in Vue can be written in both JS as well as TS. TS is the direct improvement of the JS, which adds type safety and increased readability to the project, so it was decided to write all the components using TS. Vue also provides multiple ways of creating the components. The original way of creating them was to export JS object with predefined Vue-specific fields ("data", "props") inside the "script" HTML tag, which is very inconvenient to use in case you are using TS. A more convenient way to create components by using the OOP classes was created later. This is especially good in the case of TS, because TS is considered an OOP language, whereas JS is a prototype-based language. To create a class-based component, the "vue-class-component" library must be loaded to the project. After this, the components can be created by creating classes which extend global *Vue* class using the inheritance feature of the OOP languages. This class is then exported inside the HTML "script" block, where the "lang" HTML attribute must be set to "ts".

5.2.3 Router, store and HTTP client

Routing in SPAs is different compared to traditional ways of routing. In server-side web apps, whenever a redirect happens, the user's browser is requesting the page resource from a server. In SPAs, all the pages are contained on the frontend side, meaning nothing has to be loaded from the backend. The process of redirecting in SPAs is easy and consists of changing the URL hash pattern and displaying the correct view, which corresponds to this hash. This way the content is reloaded dynamically, meaning a page refresh is not needed. To implement routing in Vue, the Vue Router library must be used. Routing in Vue is done by globally exporting the router class instance which is filled with routes. Route is a JSON object with the URL path and the view, which is shown when someone enters this URL. All routes are also assigned a name which allows redirects without a URL. Route URLs also support path parameters by setting the "props" field of the route object to "true" and adding the parameter name inside the URL with a colon before the name. Path parameters can be set to optional by putting a question mark after the

parameter name. Route object also has a special “meta” field, which can be used to pass custom parameters. Figure 8 displays an example of a route object.

```
{
  path: 'sources/:ref?',
  name: 'sources',
  props: true,
  component: FundsSources,
  meta: { authRequired: true }
}
```

Figure 8. Vue Route object example

Unauthorized users must be restricted from accessing the pages. Vue router provides a *beforeEach* hook, which is invoked before every redirect. To separate routes which need authorization from the ones which don’t, a “meta” field of the route object can be used. Author passes *authRequired* field into the “meta” field of the route object and if the field is set to *false*, users are allowed to proceed. Otherwise, the JWT and user data are queried from the Vuex store. If user data is missing and the route requires authentication, then the user gets sent back to the signup page. In case user data is present, the Axios HTTP client is imported, and “Authorization” header for every request is set to the JWT acquired from the store.

To manage authentication data around the whole app, the Vuex library must be used. Vuex enables “store”, which acts as the single source of truth inside the app. To make changes to the state of the store, special “mutations” must be created, which are synchronous operations of changing the state. Mutations are used in combination with actions. Actions are allowed to do asynchronous calculations and the results of these calculations are then committed through the mutation. Data from the store can only be accessed through “getter” methods. Store is shared throughout the whole app by exporting an instance of Vuex store specifying all the mutations, actions, getters, and fields inside the options. Vuex supports division into modules and that’s why it was decided to create a separate authentication module.

To communicate with the backend, the Axios HTTP client is used. Axios is shared throughout the whole app by exporting an instance of Axios with a custom configuration. To be able to communicate with the backend, the CORS (Cross-Origin Resource Sharing) policy must be specified. This can be done by setting the “Access-Control-Allow-Origin” as the default header for every request with a value equal to the backend server IP and

port. To pass a user's locale to the backend, the "Accepted-Language" header is used. Axios allows adding special "interceptors" to catch every response. This is used in the app to log out the user in case an error with status code 401 is received in response, because HTTP code 401 means that the JWT has likely expired or is not valid anymore. Interface data structure from TS is used to map request and response fields.

beforeEach hook configuration and Vuex store authentication module implementation can be found in Appendix 8.

5.2.4 Conversion to mobile app

To convert web app into mobile app Capacitor framework will be used. To add Capacitor into the existing JS project, the *@capacitor/ios* and *@capacitor/android* libraries were added into the project. Afterwards, the Capacitor is initialized by running the needed console commands to create projects for both platforms [58]. During initialization, Capacitor generates a *capacitor.config.ts* file, which acts as the main configuration for the mobile apps. In this config file, the IP and port on which the frontend app is running must be specified so that the mobile apps can connect to it. This enables the "live reload" feature of the Capacitor, which means that whenever a change is made to the web app, it is immediately seen on the mobile app as well [59]. Platform-specific config parameters are also available, such as disabling the link preview feature of the iOS platform.

Best way to run an Android app is to install the Android Studio IDE and then select the emulation device and run the app. For iOS, things are a bit harder, because development of iOS apps can only be done on the macOS platform inside the XCode IDE, but luckily the author is using the macOS platform so both iOS and Android apps can be developed easily. For iOS development, Xcode Command Line Tools, Homebrew and CocoaPods must be installed alongside XCode for the app to work. To run the iOS app, the required emulation iOS device (iPhone, iPad) has to be selected in XCode. After selecting the device, the app can be built and then ran. [60]

Additional Capacitor plugins can be installed to obtain control over functionalities such as phone storage, camera, in-app browser, and many others. To save things such as JWT inside the web app, the built-in JS *localStorage* is typically used, but this would not work in the case of mobile devices. That's why it was decided to use the *@capacitor/storage* plugin which adds a simple key/value persistent store for lightweight data. Things like

users' preferred locale, preferred conversion currency, user data and JWT are saved inside this storage. Phone vibration can also be controlled by using the *@capacitor/haptics* plugin. This is used to notify the user in case a success or error message pops up on the screen by making a short vibration.

During the bank authentication the user needs to be redirected to the bank authentication URL. To open a link on the mobile phone, the *@capacitor/browser* plugin is used, which allows to open the browser inside the app as well as close it whenever it's needed. When the authentication is finished, the user needs to be sent back from the mobile browser to the specific app page. To do this the "deep links" must be configured for both platforms. For the Android, an RSA certificate has to be generated and then the SHA256 fingerprint needs to be extracted. This fingerprint is then used in Google's Asset Links tool to create the Site Association file. This file is then put inside the ".well-known" folder inside the app. To finalize Android deep links, the "Intent filter" was added inside the *AndroidManifest.xml* file. To configure this on the iOS, the Apple Developer Program subscription must be bought. From the Apple Developer website, the App ID and Bundle ID have to be saved and the "Associated Domains" have to be enabled. Inside the project, an *apple-app-site-association* file must be created, which specifies the Bundle ID and App ID for the deep links. Finally, inside the XCode the link to the website inside the "Associated Domains" section is specified. To intercept the deep links inside the Vue app, a special "listener" inside the "created" Vue lifecycle hook is created, which intercepts every deep link and then the router redirects users to the required page. [61]

5.2.5 Views

Views are used to separate web app flow into different stages. User flow typically begins from the authorization view, where the user needs to register or login. Authorization view includes the name of the app; two text fields for password and email and two buttons for login and register. Text fields are made using the "<v-text-field>" tag from Vuetify, which includes custom validation based on the regular expressions provided by the author. Email and password are checked for validity. For the password, it is checked that the password is at least 8 chars long and that it has at least one capital letter. Password field has an option to show or hide the password. In case an error has happened on the backend side, a pop-up is shown on top of the screen followed by a vibration of the device. The same error handling logic is used for all the views. Figure 9 displays signup view.

All of the screenshots in this section were done using the iPhone 11 emulation from XCode.

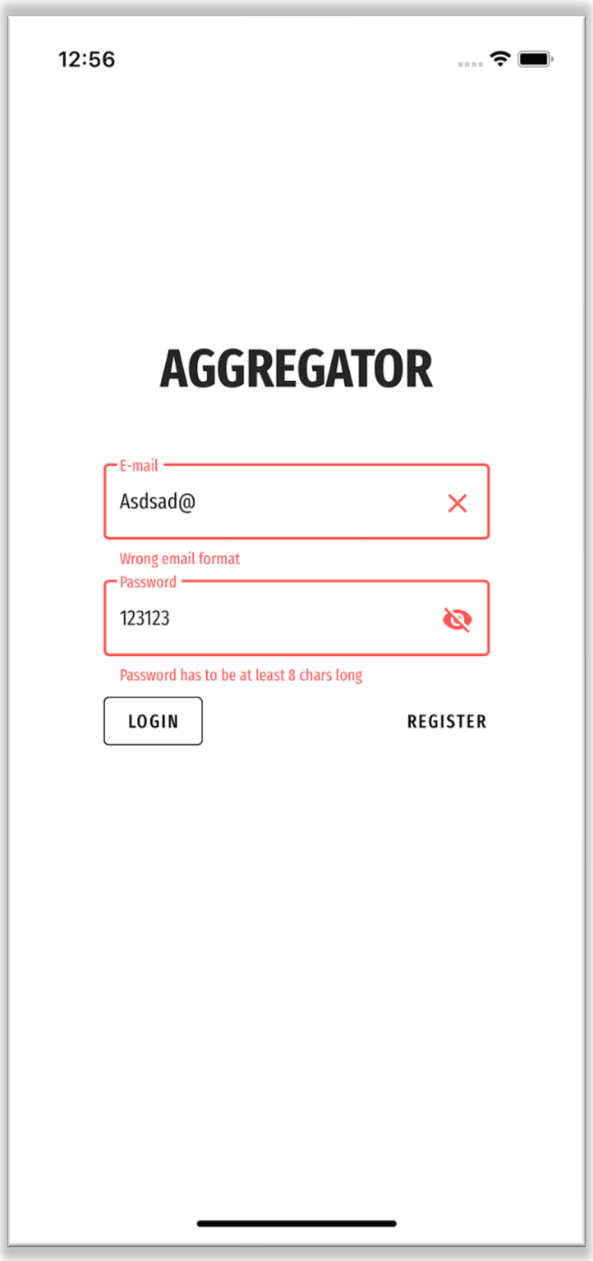

Figure 9. Signup view

After the authorization, the user is then redirected towards the balances view. An API call is made to the backend to get user balances. If there are no balances, a default placeholder which has a text telling the user to add a funds source is shown as well as a button which redirects the user to the "Add bank/exchange" view. If there is at least one balance available, a donut chart which shows the distribution of the user's balances across different currencies is displayed. Inside the chart, the user's overall balance in the currency he has selected is shown. Below the chart the balances of each asset separately (both in

original currency as well as in the currency of user's choice) are shown, accompanied by the asset's picture and its full name. Each asset has a dropdown button, which upon being pressed opens an additional section showing the distribution of this asset among different funds sources. Distribution percentages for the chart, converted total balance as well as converted balances of each asset separately are all calculated on the backend side. Figure 10 displays balances view and Figure 11 displays asset details dropdown.

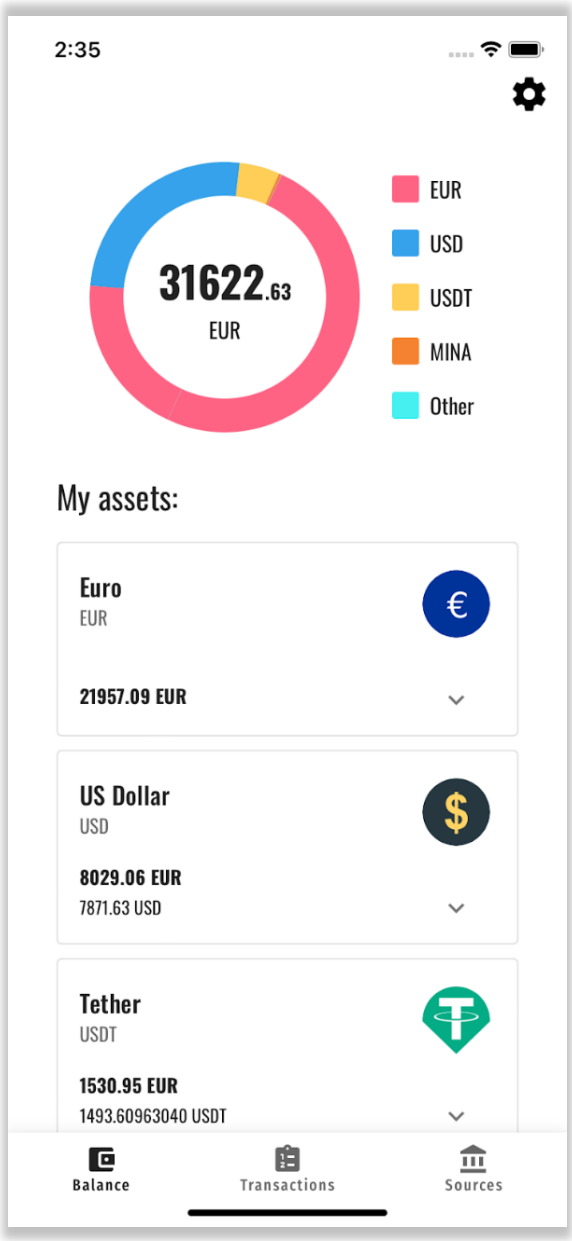

Figure 10. Balances view

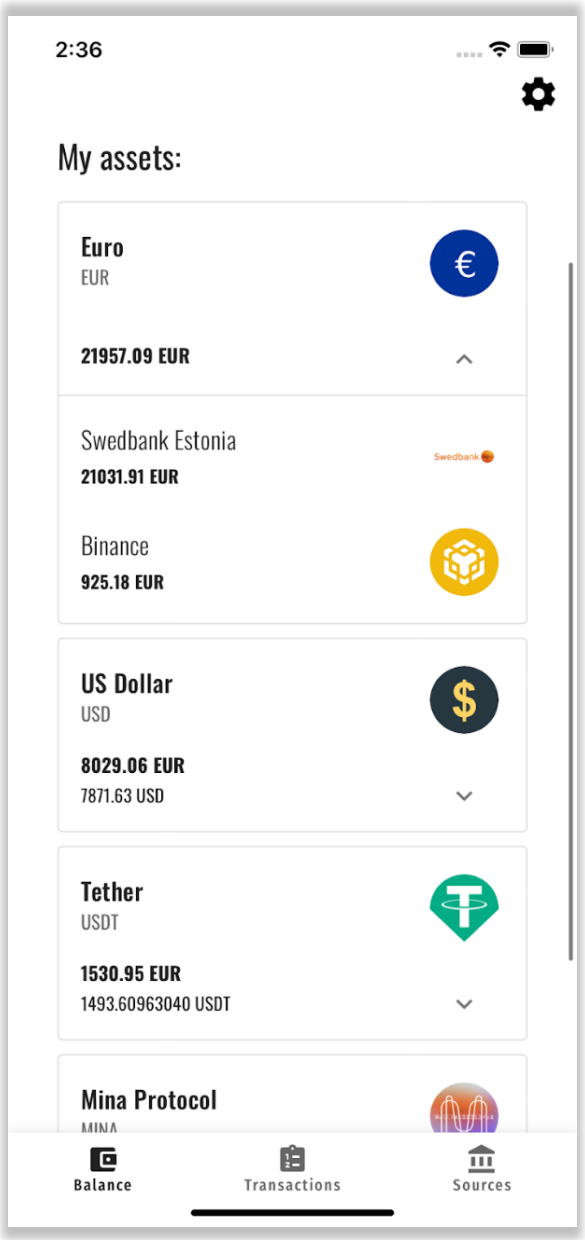

Figure 11. Asset details dropdown

To add new funds sources user must go to the sources view by clicking on the “Sources” in the navigation bar. Sources view has a selector, which allows to switch between banks and crypto exchanges. If the user has no funds sources connected, a default placeholder will be shown saying that a source must be added. By clicking on the button with the plus sign, the user gets redirected into the “choose your bank/exchange” view, where the needed bank/exchange can be selected. In the case of banks, a bank authentication URL is generated, and the user is redirected to this URL inside the in-app browser. Upon successful verification user gets sent back to the sources view using the “deep link” and the browser gets closed. Nordigen passes “ref” parameter which is used to verify that this

deep link belongs to the user. Upon successful verification of the link user can see the added bank in the list of banks. In the case of crypto exchanges, the flow is the same, but instead of generating an authentication URL, a new section with two text input fields is opened and users must enter their API key and API secret. Upon entering them and pressing the submit button the credentials get verified on the backend side and if the verification is successful user gets redirected to the exchange list page. Figure 12 displays sources view and Figure 13 displays bank choice view.

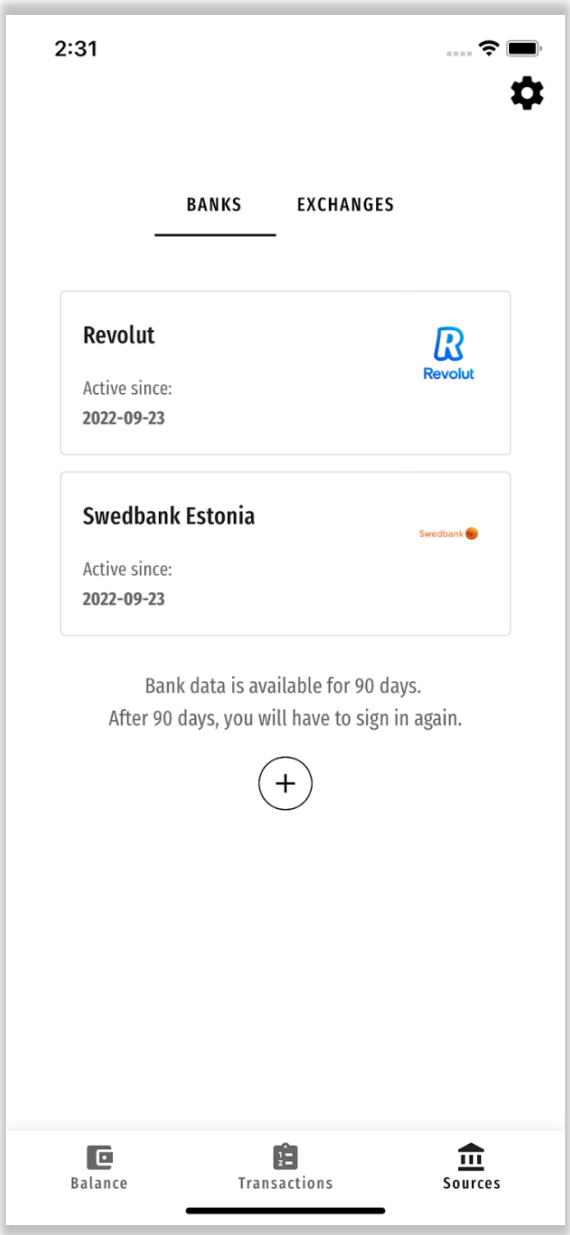

Figure 12. Sources list view

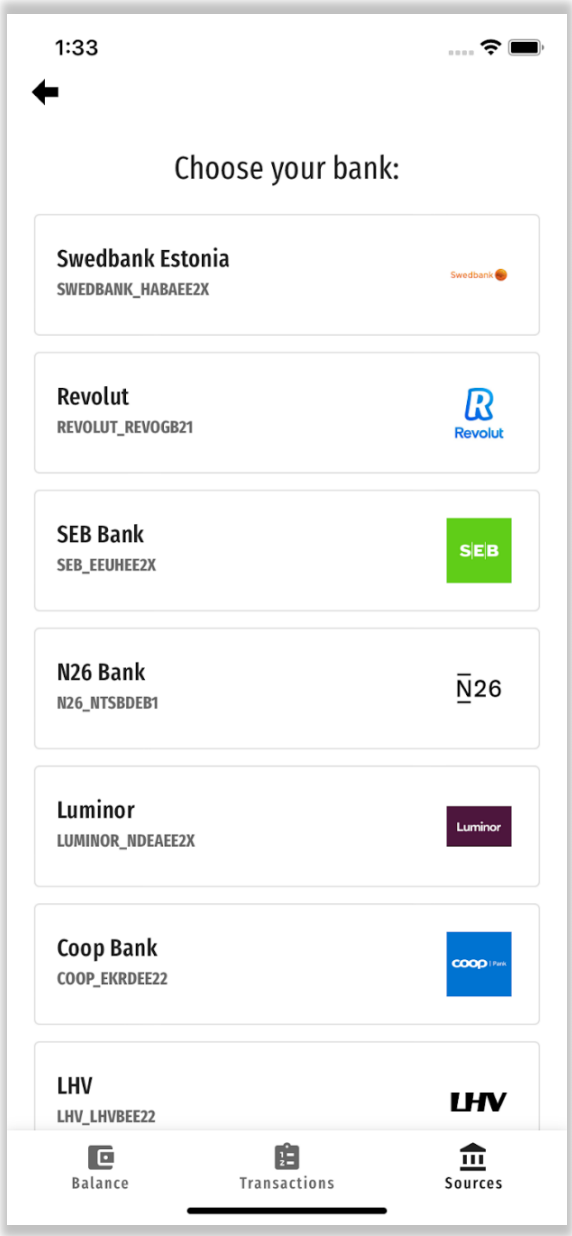

Figure 13. Bank choice view

When a user adds a new source of funds backend initiates a background task of loading the user's bank/exchange transactions. As soon as the transactions are loaded, the user can see them in the transactions view, which can be opened by clicking the corresponding button located on the bottom navigation bar. Transactions view has a selector separating bank and exchange transactions. Transactions are sorted in descending order, with the latest ones being on top. Transactions are divided into groups by the date they were made at. In case of bank transactions, the destination and the sum user has received or sent are shown. By clicking on the specific transaction, a details window appears on the screen, showing the description of the bank transaction as well as the bank picture. In case of

crypto transactions the short name of the currency is shown, followed by the action which was done with it (buy, sell, deposit, withdrawal, trade). By clicking on the specific crypto transaction, the details window is opened which contains in depth information about the transaction such as transaction fee, blockchain network, transaction identifier on the blockchain, current status of the transaction, and other things. Transactions and their statuses are updated daily through the cron job on the backend side. Figure 14 displays transactions view and Figure 15 displays transaction details popup.

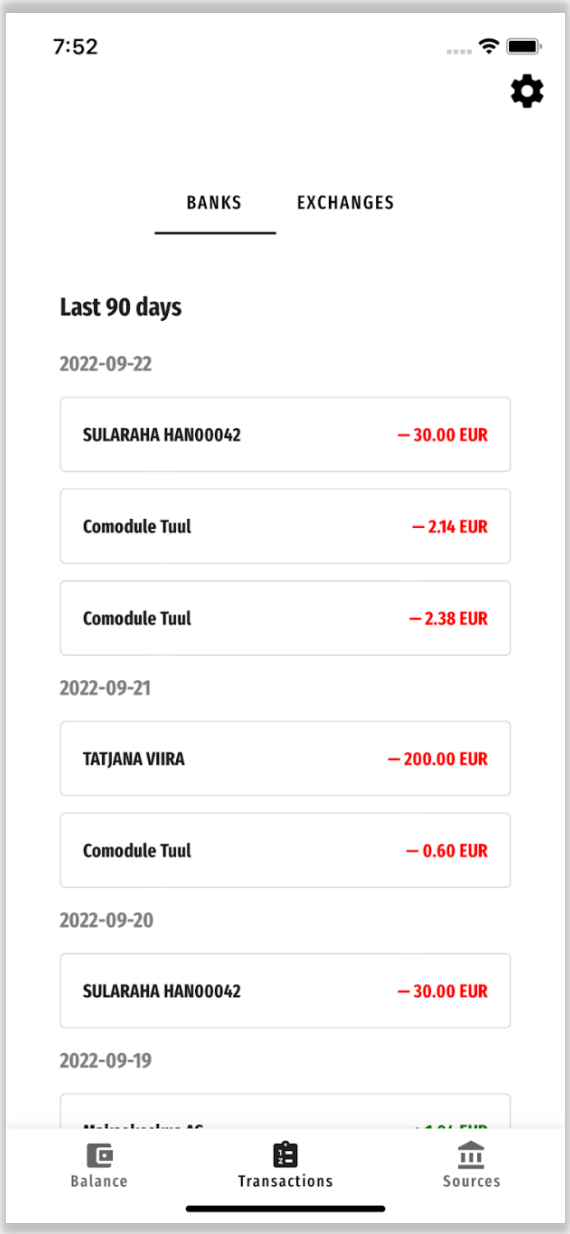

Figure 14. Transactions view

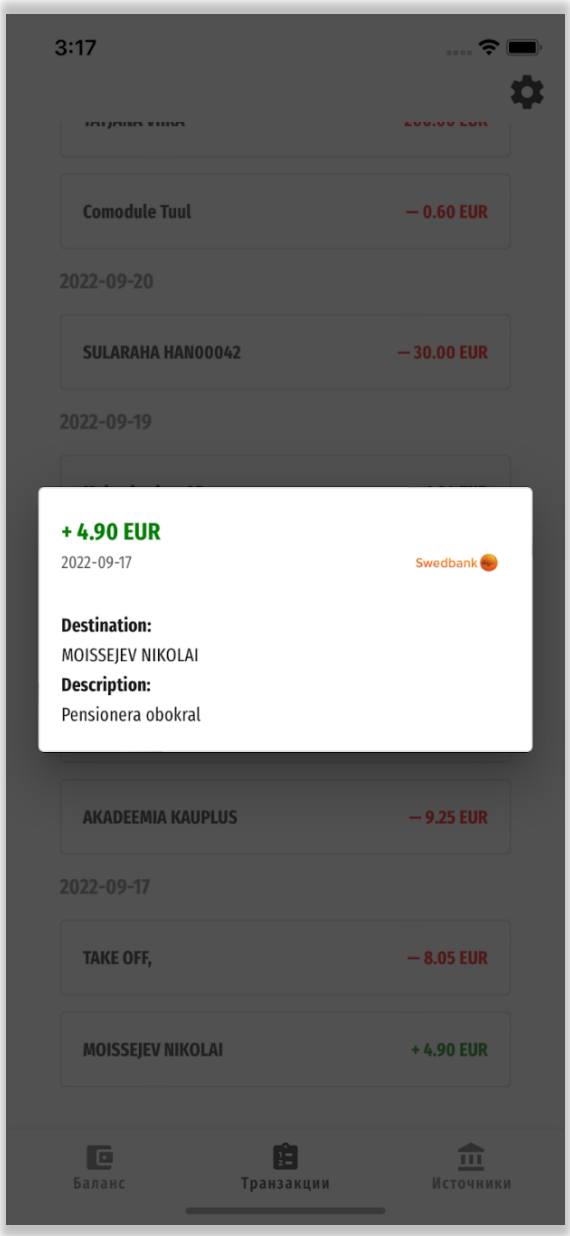

Figure 15. Transaction details popup

By clicking on the cogwheel icon located on top of the screen, the settings view can be opened. Settings view has 2 HTML selectors, where users can switch the language of the app, and the conversion currency as well as log out of the app. Figure 16 displays the settings view.

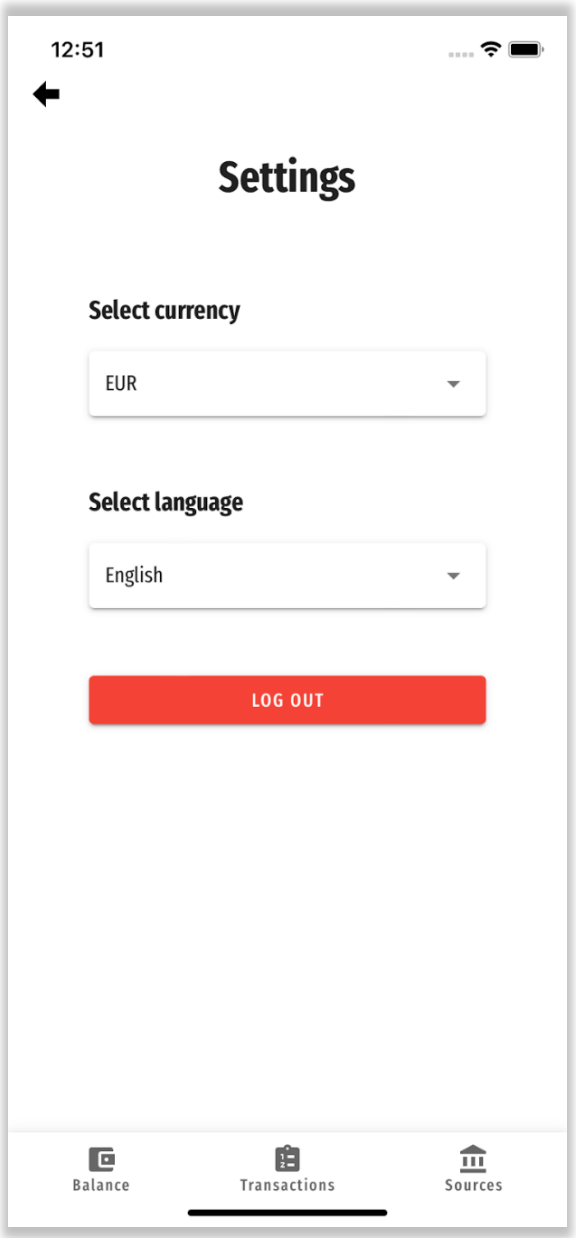

Figure 16. Settings view

6 Review of created application and possible improvements

To conduct a review, the author hosted the backend and frontend apps locally and launched the app on both iOS and Android devices. First, the author created an account and connected his bank and crypto exchange accounts to test whether the app meets all of the functional requirements. The author was able to successfully connect multiple funds sources; the balances were shown correctly, and the transactions were also loaded and displayed in the right way.

After testing the app locally on one user, the author then asked 10 testers to create an account in the app on the author's device and connect all of the Estonian banks and crypto exchanges which they have. Based on user feedback, the author could verify that the user flow is understandable. Users were able to register, switch the language and the currency, connect their funds sources, see their aggregated transactions and balances, and log out of the app on both iOS and Android devices.

Since the iOS and Android apps are not hosted for a public beta test, the author could not perform an actual end-to-end performance test of the application the proper way. The author was able to find a workaround by using the Apache JMeter and the data collected from the testers to perform a load test. JMeter is a load-testing tool mostly used to test the performance of the APIs through concurrent HTTP calls. Running the performance tests on 10 concurrent users mostly resulted in fast responses. Thanks to the Redis cache, only the first request from a user took around 3 seconds, with all of the remaining ones resulting in response times of less than 100 milliseconds.

Originally, the author was considering other functionalities as well but had to leave them for later for the sake of time management. Potential future improvements include the following:

- Monthly account balance statistics such as profit and loss, portfolio performance and various charts
- Better security (current JWT based authentication is not secure enough)
- Ability to add crypto wallets and stock brokerage accounts
- Spending tracking with categories
- More conversion currencies (currently only EUR and USD)

7 Conclusions

Today, people maintain multiple financial accounts with various financial institutions and use different types of financial assets. It has become hard for users to maintain their multiple accounts in different financial institutions as well as keep track of all the different currencies they own. Throughout this research, a mobile app was developed to address the problem and provide users with a single app through which all their financial accounts can be viewed and managed in the currency of their choice.

The mobile app was tested on multiple users. Throughout testing, users were checking for the fulfilment of functional and non-functional requirements. Performance tests were also conducted to find out how well the app performs with multiple concurrent users. The author was left happy with the results of the user tests.

Thanks to this research, the author has learned how to conduct a proper early-stage analysis before implementing the project. Throughout the development process, the author has used some of the technologies such as Project Reactor or Capacitor for the first time and the knowledge acquired by using them will definitely come in handy in the future.

7.1 Answers to research questions

In the beginning of the research two research questions were formulated. In this section, the author will attempt to answer them based on the results of the analysis and the experiment.

7.1.1 Question 1: What are the potential benefits of combining all financial applications into one?

After finishing the experiment and collecting the user feedback, the author was able to formulate the benefits of combining all financial applications into one. Firstly, the app allows users to focus on the data they actually need, leaving all of the unnecessary data behind. Secondly, the users get access to the aggregated data in a matter of seconds instead of having to login into the individual apps and calculate everything themselves.

7.1.2 Question 2: How can the cryptocurrencies and traditional fiat currencies be combined inside one application?

The author was able to find answer to this question throughout the analysis. The value of cryptocurrencies and fiat currencies is not based on any specific product or commodity and depends only on their exchange market price. On the exchanges, currencies are traded using the currency pairs. Based on that information, the author combined the cryptocurrencies and fiat currencies by converting them to a specific currency pair. This allowed to erase the difference between the different currency types for the regular users.

References

[1] Investopedia, “Accounting History and Terminology” 28 April 2022 [Online]. Available: https://www.investopedia.com/articles/08/accounting-history.asp [Accessed 18 August 2022].

[2] The World Bank, “The Global Findex Database 2021: Financial Inclusion, Digital Payments, and Resilience in the Age of COVID-19”. June 2022 [Online].
Available: https://www.worldbank.org/en/publication/globalfindex [Accessed 20 August 2022].

[3] The New York Times, “Will Cash Disappear?”. 14 November 2017 [Online].
Available:
https://www.nytimes.com/interactive/2017/11/14/business/dealbook/cashless-economy.html [Accessed 20 August 2022].

[4] Binance Research, “2021 - Global Crypto User Index”. 28 January 2021 [Online].
Available: https://research.binance.com/en/analysis/global-crypto-user-index-2021 [Accessed 20 August 2022].

[5] Toshl Finance, “Personal finance app” [Online].
Available: https://toshl.com/personal-finance/ [Accessed 23 August 2022].

[6] Kubera, “Stock & Crypto Portfolio Tracker” [Online].
Available: https://www.kubera.com/ [Accessed 23 August 2022].

[7] Cleveroad, “How to Choose Technology Stack for Web Application Development: Tips To Follow”. 23 May 2019 [Online].
Available: https://www.cleveroad.com/blog/web-development-stacks/ [Accessed 05 September 2022].

[8] Stack Overflow, “Stack Overflow Developer Survey 2022”. May 2022 [Online]
Available: https://survey.stackoverflow.co/2022 [Accessed 05 September 2022].

[9] Altexsoft, “The Good and the Bad of Java Programming”. 09 August 2018 [Online].
Available https://www.altexsoft.com/blog/engineering/pros-and-cons-of-java-programming/ [Accessed 05 September 2022].

[10] Altexsoft, “The Good and the Bad of C# Programming”. 29 October 2021 [Online].
Available: https://www.altexsoft.com/blog/c-sharp-pros-and-cons/ [Accessed 05 September 2022].

[11] AplusTopper, “Advantages and Disadvantages of Python”. 07 January 2022 [Online].

Available: https://www.aplustopper.com/advantages-and-disadvantages-of-python// [Accessed 05 September 2022].

[12] Medium, “The Pros and Cons of Node.js Web App Development: A Detailed Look”. 28 February 2022 [Online].
Available: https://javascript.plainenglish.io/the-pros-and-cons-of-node-js-web-app-development-a-detailed-look-c91a22f013c [Accessed 05 September 2022].

[13] Infiraise, “Why Choose Java For Web Application Software Development?”. 04 August 2022 [Online].
Available: https://www.infiraise.com/why-choose-java-for-web-application-software-development [Accessed 05 September 2022].

[14] Codegym, “Is Java Still Relevant? What Big Companies Use It?”. 22 June 2022 [Online].
Available: https://codegym.cc/groups/posts/771-is-java-still-relevant-what-big-companies-use-it [Accessed 05 September 2022].

[15] Rollbar, “Most Popular Java Web Frameworks in 2022”. 23 July 2022 [Online].
Available: https://rollbar.com/blog/most-popular-java-web-frameworks/ [Accessed 08 September 2022].

[16] Spring, “Spring Framework” [Online].
Available: https://spring.io/projects/spring-framework [Accessed 08 September 2022].

[17] Spring, “Spring Boot” [Online].
Available: https://spring.io/projects/spring-boot [Accessed 08 September 2022].

[18] DigitalOcean Community, “SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems?”. 21 February 2014 [Online].
Available: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems [Accessed 10 September 2022].

[19] MongoDB, “What Is MongoDB?” [Online].
Available: https://www.mongodb.com/what-is-mongodb [Accessed 10 September 2022].

[20] Redis, “Introduction to Redis [Online].
Available: https://redis.io/docs/about/ [Accessed 10 September 2022].

[21] Crio.do, “What is Redis and Why is it used by leading industries?”. 02 June 2021 [Online].
Available: https://www.crio.do/blog/what-is-redis/ [Accessed 10 September 2022].

[22] Enappd, “React Native vs Ionic vs Flutter”. 2019 [Online].
Available: https://enappd.com/blog/react-native-vs-ionic-vs-flutter/91/ [Accessed 12 September 2022].

[23] Ionic, “Ionic vs. React Native: Performance Comparison”. 23 March 2022 [Online].
Available: https://ionicframework.com/blog/ionic-vs-react-native-performance-comparison/ [Accessed 12 September 2022].

[24] Codeinwp, “Angular vs React vs Vue: Which Framework to Choose in 2022”. 17 August 2022 [Online].
Available: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/ [Accessed 14 September 2022].

[25] Invozone, “10 Reasons To Choose Vue.js For Web UI Development”. 23 September 2021 [Online].
Available: https://invozone.com/blog/10-reasons-to-choose-vue-js-for-web-ui-development/ [Accessed 14 September 2022].

[26] Vue Community, “UI Libraries [Online].
Available: https://vue-community.org/guide/ecosystem/ui-libraries.html [Accessed 12 September 2022].

[27] JetBrains, “Java Programming - The State of Developer Ecosystem 2021”. 16 July 2021 [Online].
Available: https://www.jetbrains.com/lp/devecosystem-2021/java/ [Accessed 16 September 2022].

[28] Amazon Web Services, “What is an API?” [Online].
Available: https://aws.amazon.com/what-is/api/ [Accessed 18 September 2022].

[29] Postman, “2022 State of the API Report. API Technologies.” August 2022 [Online].
Available: https://www.postman.com/state-of-api/api-technologies/#api-technologies [Accessed 18 September 2022].

[30] OpenAPIHub API Blog, “3 Major Types of API – Public API, Private API & Partner API”. 10 January 2022 [Online].
Available: https://blog.openapihub.com/en-us/3-major-types-of-api-public-api-private-api-partner-api/ [Accessed 18 September 2022].

[31] LHV, “Open banking” [Online].
Available: https://www.lhv.ee/en/open-banking [Accessed 18 September 2022].

[32] The Berlin Group, “PSD2 Access to Bank Accounts” [Online].
Available: https://www.berlin-group.org/psd2-access-to-bank-accounts [Accessed 18 September 2022].

[33] LHV, “Introduction to LHV Open Banking” [Online].
Available: https://partners.lhv.ee/en/open-banking/#access-and-usage [Accessed 18 September 2022].

[34] Open Banking Tracker, “Open banking in Estonia” [Online]. Available: https://www.openbankingtracker.com/country/estonia [Accessed 18 September 2022].

[35] Salt Edge, “Open banking for every business” [Online]. Available: https://www.saltedge.com/ [Accessed 18 September 2022].

[36] Salt Edge, “Countries and connected banks. Estonia.” [Online]. Available: https://www.saltedge.com/products/account_information/coverage/ee [Accessed 18 September 2022].

[37] Salt Edge Docs, “Your Account” [Online]. Available: https://docs.saltedge.com/general/#your_account [Accessed 18 September 2022].

[38] Enable Banking, “The Connectivity Engine that Puts You in Charge” [Online]. Available: https://enablebanking.com/ [Accessed 18 September 2022].

[39] Enable Banking, “Open Banking APIs” [Online]. Available: https://enablebanking.com/open-banking-apis [Accessed 18 September 2022].

[40] Enable Banking Docs, “Certificate upload and application registration” [Online]. Available: https://enablebanking.com/docs/tilisy/latest/#certificate-upload-and-application-registration [Accessed 18 September 2022].

[41] Nordigen, “About us” [Online]. Available: https://nordigen.com/en/company/about-us/ [Accessed 18 September 2022].

[42] Nordigen, “Coverage” [Online]. Available: https://nordigen.com/en/coverage/ [Accessed 18 September 2022].

[43] Nordigen, “Pricing” [Online]. Available: https://nordigen.com/en/pricing/ [Accessed 18 September 2022].

[44] Bitpanda, “What's the difference between a cryptocurrency like Bitcoin and fiat money?” [Online]. Available: https://www.bitpanda.com/academy/en/lessons/whats-the-difference-between-a-cryptocurrency-like-bitcoin-and-fiat-money/ [Accessed 20 September 2022].

[45] Investopedia, “Cryptocurrency Exchanges: What They Are and How to Choose” [Online]. Available: https://www.investopedia.com/tech/190-cryptocurrency-exchanges-so-how-choose/ [Accessed 20 September 2022].

[46] CoinMarketCap, “Top Cryptocurrency Spot Exchanges” [Online]. Available: https://coinmarketcap.com/rankings/exchanges/ [Accessed 20 September 2022].

[47] Binance, “Binance API Docs” [Online].
Available: https://binance-docs.github.io/apidocs/spot/en [Accessed 20 September 2022].

[48] Coinbase Cloud, “Documentation” [Online].
Available: https://docs.cloud.coinbase.com/sign-in-with-coinbase/docs [Accessed 20 September 2022].

[49] Kraken, “REST API Documentation” [Online].
Available: https://docs.kraken.com/rest/ [Accessed 20 September 2022].

[50] TolaData, “Qualitative and Quantitative data collection methods in M&E”. 17 June 2021 [Online].
Available: https://www.toladata.com/blog/qualitative-and-quantitative-data-collection-methods-in-monitoring-and-evaluation/ [Accessed 22 August 2022].

[51] Atlassian, “User stories with examples and a template”. August 2018 [Online]
Available: https://www.atlassian.com/agile/project-management/user-stories [Accessed 25 August 2022].

[52] Altexsoft, “Functional and Nonfunctional Requirements: Specification and Types”. 23 July 2021 [Online].
Available: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/ [Accessed 25 August 2022].

[53] Indeed, “9 Nonfunctional Requirements Examples”. 4 May 2021 [Online].
Available: https://www.indeed.com/career-advice/career-development/non-functional-requirements-examples [Accessed 25 August 2022].

[54] HowToDoInJava, “Java – Create a Secure Password Hash” [Online].
Available: https://howtodoinjava.com/java/java-security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/ [Accessed 06 October 2022].

[55] JWT, “Introduction to JSON Web Tokens” [Online].
Available: https://jwt.io/introduction [Accessed 06 October 2022].

[56] HowToDoInJava, “Java AES-256 Encryption and Decryption” [Online].
Available: https://howtodoinjava.com/java/java-security/aes-256-encryption-decryption/ [Accessed 06 October 2022].

[57] Medium, “Single-page application vs. multiple-page application”. 02 December 2016 [Online].
Available: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58 [Accessed 15 October 2022].

[58] Capacitor Docs, “Installing Capacitor” [Online].
Available: https://capacitorjs.com/docs/getting-started [Accessed 18 October 2022].

[59] Capacitor Docs, “Live Reload” [Online].

Available: https://capacitorjs.com/docs/guides/live-reload [Accessed 18 October 2022].

[60] Capacitor Docs, “Environment Setup” [Online]. Available: https://capacitorjs.com/docs/getting-started/environment-setup [Accessed 18 October 2022].

[61] Capacitor Docs, “Deep Linking with Universal and App Links” [Online]. Available: https://capacitorjs.com/docs/guides/deep-links [Accessed 18 October 2022].

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

I, Nikita Viira

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Financial Data Aggregation from Multiple Sources”, supervised by Mohammad Tariq Meeran
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

04.01.2023

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 – Implementation of R2DBC

```
@Configuration
@EnableR2dbcRepositories
@RequiredArgsConstructor
@Slf4j
public class DatabaseConfiguration extends AbstractR2dbcConfiguration
{
  private final ObjectMapper objectMapper;

  @Value("${database.host}")
  private String host;
  @Value("${database.port}")
  private Integer port;
  @Value("${database.password}")
  private String password;
  @Value("${database.username}")
  private String username;
  @Value("${database.name}")
  private String databaseName;

  @Bean
  @Override
  public ConnectionFactory connectionFactory() {
    var connectionFactory =
ConnectionFactories.get(ConnectionFactoryOptions.builder()
      .option(DRIVER, "pool")
      .option(PROTOCOL, "postgresql")
      .option(HOST, host)
      .option(PORT, port)
      .option(USER, username)
      .option(PASSWORD, password)
      .option(DATABASE, databaseName)
      .option(MAX_SIZE, 30)
      .build());
    var configuration =
ConnectionPoolConfiguration.builder(connectionFactory)
      .maxIdleTime(Duration.ofMinutes(30))
      .initialSize(10)
      .maxSize(30)
      .maxCreateConnectionTime(Duration.ofSeconds(1))
      .build();
    return new ConnectionPool(configuration);
  }
}
```

Figure 17. R2DBC configuration

```
@Repository
@RequiredArgsConstructor
@Slf4j
public class UserRepository {
  private final R2dbcEntityTemplate template;

  public Mono<Integer> updateLastTransactionSync(List<Long> userIds) {
    return this.template.update(query(where("id").in(userIds)),
      update("last_transaction_sync", Instant.now()), User.class);
  }

  public Mono<User> save(User user) {
    return this.template.insert(User.class).using(user);
  }

  public Mono<User> find(String email) {
    return this.template.selectOne(query(where("email").is(email)),
User.class);
  }
}
```

Figure 18. Spring repository

Appendix 3 – Implementation of reactive Redis cache

```
@Configuration
@RequiredArgsConstructor
public class RedisConfiguration {
  private final ObjectMapper objectMapper;
  @Value("${spring.redis.host}")
  private String host;
  @Value("${spring.redis.port}")
  private int port;

  @Bean
  public ReactiveRedisConnectionFactory connectionFactory() {
    return configureLettuceConnectionFactory();
  }

  private LettuceConnectionFactory configureConnectionFactory(){
    ClientOptions clientOptions = ClientOptions.builder()
      .autoReconnect(true)
      .pingBeforeActivateConnection(true)
      .build();

    LettucePoolingClientConfiguration clientConfig =
LettucePoolingClientConfiguration.builder()
      .clientOptions(clientOptions)
      .commandTimeout(Duration.ofSeconds(2))
      .shutdownTimeout(Duration.ZERO)
      .build();

    return new LettuceConnectionFactory(new
RedisStandaloneConfiguration(host, port), clientConfig);
  }

  @Bean
  @DependsOn("objectMapper")
  public ReactiveRedisTemplate<String, CachedAssetMetadata>
reactiveTemplateCachedCurrencyRate() {
    var serializer = new
Jackson2JsonRedisSerializer<>(CachedAssetMetadata.class);
    serializer.setObjectMapper(objectMapper);

    RedisSerializationContext.RedisSerializationContextBuilder<String,
CachedAssetMetadata> builder =
      RedisSerializationContext.newSerializationContext(new
StringRedisSerializer());
    RedisSerializationContext<String, CachedAssetMetadata> context =
builder.value(serializer).build();
    return new
ReactiveRedisTemplate<>(reactiveRedisConnectionFactory(), context);
  }
}
```

Figure 19. Reactive Redis configuration

```
@Component
public class AssetMetadataCache {
  private final ReactiveValueOperations<String, CachedAssetMetadata>
valueOps;
  private static final Duration ttl = ofHours(1);

  public AssetMetadataCache(ReactiveRedisTemplate<String,
CachedAssetMetadata> reactiveCachedAssetMetadataTemplate) {
    this.valueOps = reactiveCachedAssetMetadataTemplate.opsForValue();
  }

  public Mono<CachedAssetMetadata> save(CachedAssetMetadata
cachedAssetMetadata) {
    return valueOps.set(id(cachedAssetMetadata.symbol),
cachedAssetMetadata, ttl).thenReturn(cachedAssetMetadata);
  }

  public Mono<CachedAssetMetadata> get(String symbol) {
    return valueOps.get(id(symbol.toUpperCase()));
  }

  private String id(String symbol) {
    return format("asset-metadata:%s", symbol.toUpperCase());
  }
}
```

Figure 20. Usage of reactive Redis

Appendix 4 – Authorization implementation

```
@Component
@Slf4j
public class HeaderCallerContextArgumentResolver implements
HandlerMethodArgumentResolver {
  @Value("${token.secret}")
  private String secret;
  @Value("${token.issuer}")
  private String issuer;

  @Override
  public boolean supportsParameter(@NotNull MethodParameter parameter)
{
    return
AuthorizationContext.class.isAssignableFrom(parameter.getParameterType
());
  }

  @Override
  public Mono<Object> resolveArgument(MethodParameter parameter,
                                      BindingContext bindingContext,
                                      ServerWebExchange exchange) {
    return Mono.just(extractToken(exchange))
      .map(token -> {
        verifyToken(token);
        var decodedJwt = JWT.decode(token);
        return new AuthorizationContext(
          decodedJwt.getClaim("userId").asLong(),
          decodedJwt.getClaim("email").asString()
        );
      })
      .cast(Object.class)
      .onErrorResume(e -> {
        log.error(e.getMessage());
        return Mono.error(new UnauthorizedException("unauthorized-
error"));
      });
  }

  private void verifyToken(String token) {
    try {
      JWT.require(HMAC512(secret))
        .withIssuer(issuer)
        .build()
        .verify(token);
    } catch (Exception e) {
      throw new TechnicalException("Failed to verify JWT token", e);
    }
  }

  private String extractToken(ServerWebExchange serverWebExchange) {
```

```
    return serverWebExchange.getRequest()
      .getHeaders()
      .getOrDefault(AUTHORIZATION, List.of())
      .stream()
      .findFirst()
      .filter(h -> h.startsWith("Bearer "))
      .map(h -> h.substring("Bearer ".length()))
      .orElseThrow(() -> new UnauthorizedException("unauthorized-
error"));
  }

  @AllArgsConstructor
  @NoArgsConstructor
  public static class AuthorizationContext {
    public Long userId;
    public String email;
  }
}
```

Figure 21. Implementation of the HeaderCallerContextArgumentResolver interface

Appendix 5 – WebClient implementation

```
@Configuration
public class WebClientFactory {
  public static final int TIMEOUT_IN_SECONDS = 5;

  public static WebClient createWebClient(String baseUrl) {
    var connector = new ReactorClientHttpConnector(HttpClient.create()
        .option(CONNECT_TIMEOUT_MILLIS, TIMEOUT_IN_SECONDS * 1000)
       .doOnConnected(c -> c.addHandlerLast(new
ReadTimeoutHandler(TIMEOUT_IN_SECONDS))
       .addHandlerLast(new WriteTimeoutHandler(TIMEOUT_IN_SECONDS)))
       .runOn(LoopResources.create("reactor-webclient")));

    var build = ExchangeStrategies.builder()
        .codecs(configurer -> configurer.defaultCodecs()
        .maxInMemorySize(-1))
        .build();

    return WebClient.builder()
            .exchangeStrategies(build)
            .clientConnector(connector)
            .baseUrl(baseUrl)
            .build();
  }
}
```

Figure 22. WebClient configuration

```
public Mono<String> createAuthLink(String accessToken, String
bankUniqueCode) {
  var requestBody = new AuthLinkRequest(bankUniqueCode, redirectUrl);
  return nordigenRequest(POST, "/requisitions/",
AuthLinkResponse.class, requestBody, Optional.of(accessToken))
    .map(resp -> resp.ref);
}

private <T> Mono<T> nordigenRequest(HttpMethod method, String url,
Class<T> responseType, Object requestBody, Optional<String>
accessToken) {
  var request = nordigenApiClient.method(method).uri(url);
  if (requestBody != null) {
    request = (WebClient.RequestBodySpec)
request.body(fromValue(requestBody));
  }

  return request
    .headers(headers -> {
      headers.add(CONTENT_TYPE, APPLICATION_JSON_VALUE);
      headers.add(ACCEPT, APPLICATION_JSON_VALUE);
      accessToken.ifPresent(token -> headers.add(AUTHORIZATION,
"Bearer " + token));
    })
    .retrieve()
    .onStatus(HttpStatus::isError, response -> onErrorHandler(url,
response))
    .bodyToMono(responseType)
    .retryWhen(backoff(RETRY_COUNT, ofMillis(500L)));
}

private Mono<Throwable> onErrorHandler(String url, ClientResponse
response) {
  return response.createException().flatMap(resp -> {
    log.error("Requesting Nordigen API failed with status " +
resp.getRawStatusCode()
      + "; URL=" + url + "; Body: " + resp.getResponseBodyAsString());
    return Mono.error(new TechnicalException("Nordigen API failure",
resp));
  });
}
```

Figure 23. WebClient usage

Appendix 6 – Spring exception handler

```
@RestControllerAdvice
@Slf4j
public class ExceptionHandler extends AbstractErrorWebExceptionHandler
{
  private final TranslationUtil translationUtil;
  private final Map<Class, BiFunction<Throwable, Locale,
Mono<ServerResponse>>> errorMapping =
    Map.ofEntries(
      entry(ServerException.class, this::badRequest),
      entry(UnauthorizedException.class, this::unauthorizedRequest)
    );

  public ExceptionHandler(ErrorAttributes errorAttributes,
ApplicationContext applicationContext,
                          ServerCodecConfigurer serverCodecConfigurer,
TranslationUtil translationUtil) {
    super(errorAttributes, new WebProperties.Resources(),
applicationContext);
    super.setMessageReaders(serverCodecConfigurer.getReaders());
    super.setMessageWriters(serverCodecConfigurer.getWriters());
    this.translationUtil = translationUtil;
  }

  @Override
  protected RouterFunction<ServerResponse>
getRoutingFunction(ErrorAttributes errorAttributes) {
    return RouterFunctions.route(RequestPredicates.all(),
this::renderErrorResponse);
  }

  private Mono<ServerResponse> renderErrorResponse(ServerRequest
request) {
    Throwable error = getError(request);
    Locale locale =
translationUtil.getLocale(request.headers().firstHeader("Accept-
Language"));

    for (var entry : errorMapping.entrySet()) {
      if (entry.getKey().isInstance(error)) {
        return errorMapping.get(entry.getKey()).apply(error, locale);
      }
    }

    return internalServerError(error, locale);
  }

  public final Mono<ServerResponse> badRequest(Throwable error, Locale
locale) {
    return ServerResponse
      .status(BAD_REQUEST)
```

```
      .body(Mono.just(new
ErrorDto(translationUtil.getMessage(error.getMessage(), locale))),
ErrorDto.class);
  }

  public final Mono<ServerResponse> unauthorizedRequest(Throwable
error, Locale locale) {
    return ServerResponse
      .status(UNAUTHORIZED)
      .body(Mono.just(new
ErrorDto(translationUtil.getMessage(error.getMessage(), locale))),
ErrorDto.class);
  }

  public final Mono<ServerResponse> internalServerError(Throwable
error, Locale locale) {
    log.error("Unexpected error occurred while executing request: " +
error.getMessage(), error);
    return ServerResponse
      .status(INTERNAL_SERVER_ERROR)
      .body(Mono.just(new
ErrorDto(translationUtil.getMessage("unexpected-error", locale))),
ErrorDto.class);
  }
}
```

Figure 24. Spring exception handler

Appendix 7 – docker-compose.yml file

```
version: '3.5'

services:
  test-db:
    container_name: aggregator-test-db
    image: postgres:13.2
    restart: always
    ports:
      - "25432:5432"
    tmpfs:
      - /var/lib/postgresql/data:rw
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: SuperSecret
      POSTGRES_DB: aggregator
    volumes:
      - postgres_data:/data/db

  test-redis:
    image: redis:6-alpine
    restart: always
    ports:
      - "16379:6379"
    command: [ sh, -c, "rm -f /data/dump.rdb && redis-server" ]
    hostname: redis
    volumes:
      - redis-data:/data

  integration-test-mock-server:
    image: jamesdbloom/mockserver:mockserver-5.11.2
    restart: always
    ports:
      - "1080:1080"
    environment:
      - LOG_LEVEL=WARN

volumes:
  redis-data:
  postgres_data:
```

Figure 25. docker-compose.yml file

Appendix 8 – Frontend implementation

```
router.beforeEach(async (to, from, next) => {
  let token = store.getters['user/getToken'] as string | undefined;
  let userData = store.getters['user/getUser'] as UserData |
undefined;

  if (userData === undefined) {
    await store.dispatch('user/loadStorage')
      .then(() => {
        token = store.getters['user/getToken'] as string;
        userData = store.getters['user/getUser'] as UserData;
      });
  }

  if (userData === undefined && to.meta.authRequired) {
    return router.push({ name: 'signup', params: { locale:
translation.currentLocale } });
  }

  if (userData !== undefined) {
    if (to.meta.authRequired) {
      Axios.defaults.headers.common.Authorization = 'Bearer ' + token;
    }

    if (to.meta.disallowAuthed) {
      return router.push({ name: 'balances', params: { locale:
translation.currentLocale } });
    }
  }

  return next();
});
```

Figure 26. Vue router's beforeEach hook configuration

```
@Module({ namespaced: true })
class User extends VuexModule {
  private token: string | undefined = undefined;
  private userCached: UserData | undefined = undefined;

  get isLoggedIn(): boolean {
    return !!this.token;
  }

  get getToken(): string | undefined {
    return this.token;
  }

  get getUser(): UserData | undefined {
    return this.userCached;
  }

  @Mutation
  setUserInfo(payload: LoginResponse): void {
    this.token = payload.jwtToken;
    this.userCached = payload.user;
  }

  @Mutation
  removeUserInfo(): void {
    this.token = undefined;
    this.userCached = undefined;
  }

  @Action({ rawError: true })
  async loadStorage(): Promise<void> {
    const userInfo = await Storage.get({ key: 'userInfo' });
    const jwt = await Storage.get({ key: 'jwt' });

    if (userInfo.value && jwt.value) {
      this.context.commit('setUserInfo', { jwtToken: jwt.value, user:
JSON.parse(userInfo.value) });
    }
  }

  @Action({ rawError: true })
  register(credentials: RegisterRequest): Promise<void> {
    return new Promise((resolve, reject) => {
      UserController.register(credentials)
        .then(async resp => {
          const { jwtToken, user } = resp.data;

          await Storage.set({ key: 'userInfo', value:
JSON.stringify(user) });
          await Storage.set({ key: 'jwt', value: jwtToken });

          this.context.commit('setUserInfo', { jwtToken, user });

          resolve();
```

```
        })
        .catch(async err => {
          await Storage.remove({ key: 'userInfo' });
          await Storage.remove({ key: 'jwt' });

          reject(err);
        });
    });
  }

  @Action({ rawError: true })
  login(credentials: LoginRequest): Promise<void> {
    return new Promise((resolve, reject) => {
      UserController.login(credentials)
        .then(async resp => {
          const { jwtToken, user } = resp.data;

          await Storage.set({ key: 'userInfo', value:
JSON.stringify(user) });
          await Storage.set({ key: 'jwt', value: jwtToken });

          this.context.commit('setUserInfo', { jwtToken, user });

          resolve();
        })
        .catch(async err => {
          await Storage.remove({ key: 'userInfo' });
          await Storage.remove({ key: 'jwt' });

          reject(err);
        });
    });
  }

  @Action({ rawError: true })
  async logout(): Promise<void> {
    await Storage.remove({ key: 'userInfo' });
    await Storage.remove({ key: 'jwt' });

    this.context.commit('removeUserInfo');
    delete Axios.defaults.headers.common.Authorization;
  }
}

export default User;
```

Figure 27. Vuex store authorization module