Finance Management Application "Literate Consumer" (Front-end Part)

Author: Tikhon Zaikin
Study program: Open Informatics

Supervisor: Ing. Jiří Šebek

May 24, 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Zaikin Tikhon** Personal ID number: **491879**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Frontend for Personal finance management application

Bachelor’s thesis title in Czech:

Frontend pro Aplikaci pro správu osobních financí

Guidelines:

For the people wanting to optimize their money manipulation, create an application that facilitates the problem of finance management.
1) Understand the problem of finance management in the modern world and research available applications for it.
2) Research banking applications as they provide some finance management functionality as well.
3) Research user interface of the analyzed applications to provide a better user experience.
4) Based on the analysis, design and implement a web application.
5) It should include the essential functionality from the analyzed applications as well as a friendly and effective user interface.
6) Authorization should be implemented as well.
7) Choose appropriate technologies for developing and testing an application.
8) Take all the results into account and decide if the application can be enhanced.

Bibliography / sources:

1. Robert C. Martin: Clean Code. A Handbook of Agile Software Craftsmanship
2. Pressman R. S.: Software Engineering
3. Masse M.: REST API design rulebook: designing consistent RESTful web service interfaces
4. Wohlgethan E.: Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js: https://reposit.haw-hamburg.de/bitstream/20.500.12738/8417/1/BA_Wohlgethan_2176410.pdf
5. Schlatter T., Levinson D.: Visual usability: Principles and practices for designing digital applications

Name and workplace of bachelor’s thesis supervisor:

Ing. Jiří Šebek Center for Software Training FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **11.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Jiří Šebek
Supervisor’s signature

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student’s signature

Acknowledgements

Firstly, I would like to thank Ing. Jírí Šebek for becoming our supervisor, he practically saved our lives in the last semester.

Secondly, I would like to thank Ing. Božena Mannová, Ph.D. for supporting and giving useful pieces of advice for the basis of this project.

Thirdly, I would like to thank Ivan Sedakov for being a colleague on this project. It wouldn't be possible to create the application without him.

Lastly and most certainly not leastly, I would like to thank everyone, who surrounded me during my lifetime and helped me with any possible means.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

Abstract

The project aims at creating a web application that helps manage finances. Three applications for similar purposes were examined. Their interface and functional advantages and disadvantages were taken into account, so a better application was created. Different front-end frameworks, including user interface frameworks, were analyzed and chosen to be used in the development process. Then all the application requirements were created in accordance with the assignment. After that, a client architecture and user interface design were created. Then the client was implemented and tested synchronously. In the end, usability tests of the created application were conducted to see if user experience improvements are required.

Keywords: web application, software engineering, finances, vue.js, frontend, quasar, figma, nightwatch

Abstrakt

Cílem této práce je vytvořit webovou aplikaci, která pomáhá spravovat finance. Byly zkoumány tři aplikace pro podobné účely. Bylo analyzováno jejich rozhraní a funkční výhody a nevýhody, a jako důsledek byla vytvořena lepší aplikace. Byly analyzovány různé frontendové frameworky, včetně UI frameworků, a vybrány ty, které budou použity v procesu vývoje. Poté byly vytvořeny všechny požadavky na aplikaci v souladu se zadáním. Poté byla vytvořena architektura klienta a návrh uživatelského rozhraní. Poté byl implementován klient a v též době průběžně testován. Nakonec byly provedeny testy použitelnosti vytvořené aplikace, aby se zjistilo, zda je třeba vylepšit uživatelský komfort.

Klíčová slova: webová aplikace, softwarové inženýrství, finance, vue.js, frontend, quasar, figma, nightwatch

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter contains the motivation and the aim of the project. The subsequent chapters contain information about existing application research, application design, its implementation, and testing. The concluding chapters contain the analysis of the results and what could be done to enhance the application.

1.1 Motivation

These days finance management is one of the most important skills in our modern life, but it is not being taught as a compulsory subject. As a result, most people gain experience in financial literacy through learning from their mistakes. And some of those mistakes can be dreadful. For example, people have to deal with bankruptcy, defaults, and foreclosures and learn from it.[53]

Relying on one's head can be unwise when it comes to calculating finances. Using an application that computes and stores everything for a user is a better option. It can help reach ambitious personal goals that seemed to be unachievable. Overall, it can provide better control over one's finances, so one will not have to survive until getting paid.[11]

These problems are already solved by some applications, but they may have disadvantages, such as an unfriendly user interface or lack of some features. There are also several quality applications providing needful functionality. The LiterateConsumer application was developed with trying to absorb the best of the available finance management applications.

1.2 Aims and Objectives of the Project

The project aimed at creating a user-friendly useful effective application that optimizes economic aspects of life. The application was developed to be suitable for everyone eager to organize their data with a minimal number of clicks and minimum thinking. Thanks to a user-friendly interface, the application has low barriers to entry, so really small efforts should be made by a user to start using it to the fullest. [23]

The application was designed to provide clear information about users' incomes, outcomes, and current money balance. Users can categorize transactions, create "wallets" for different currencies (CZK, USD, EUR, RUB...), transfer and convert money between "wallets", and make debt records. In addition, users can organize a group to control the collective money flow. It can be useful for families or companies.

As for the objectives, detailed research on development technologies and similar applications was carried out. Advantages and disadvantages were analyzed to determine the business requirements that were divided into functional and non-functional ones. As a consequence, use cases were created from these requirements. The next step was to create a domain model. [26]

Then it came to designing an application architecture. A database structure was derived from the domain model. API methods were created according to use-cases. Afterward, an implementation process was started with selected technologies. During the implementation, the git version control system was used. Every public API method was documented and some difficult places in code as well.

Last but not least, after the implementation was finished, unit tests were written for all the public API methods as well as end-to-end front-end interface tests. Tests were written according to test scenarios derived from use cases.

Chapter 2

Research

Existing applications for finance management and front-end technologies are going to be analyzed in the next chapter. Based on research results, functional and non-functional requirements were determined. Consequently, possible use cases and an analytic domain model were derived.

2.1 Research of Existing Applications

Three applications with similar purposes were analyzed. The goal was to research the user interface and the functionality provided by the application.

2.1.1 George Go

The George Go[1] application is provided by the Česka Spořitelná bank. To be precise, this is not a finance management application, it is a bank application. Anyway, it includes some functionality related to finance management:

- Overview of every incoming and outgoing transaction.
- Account status (money balances).
- Transactions are automatically categorized, so a user can perform a search by them. A new category can be created.
- Statistics with graphs related to money flow.

Also, all the transactions are created automatically as it is a banking application. But, unfortunately, as a consequence, if a user does not have a bank card, he will not be able to use the application. Moreover, you cannot create custom accounts that could be used

[1] https://www.csas.cz/en/mobile-apps/george-go

for cash management, everything should be bound to the bank. As for the user interface, it is pretty simple and minimalistic and everyone can start using George Go easily.

2.1.2 CoinKeeper

The CoinKeeper[2] application was developed by a Russian IT company I-Free. Compared to George Go, it is a finance management application. It has such functions suitable for our application:

- Incoming and outgoing transactions.
- Accounts with different currencies, created by a user.
- Categories, assigning them to transactions.
- Statistics with graphs related to money flow.
- Importing data from real bank accounts as well as exporting data from accounts created in the application.

Also. the application allows a user to buy a premium subscription that unlocks such functions as:

- Automatic synchronization with banks.
- Collaborative finance management.
- Virtual card tracking all the incomes and outcomes.

In comparison to George Go, all the transactions should be created manually unless a user imports a list. Almost everything is done manually if you do not have a premium subscription. As for the user interface, it is minimalistic as well, but the account section was made quite unobvious as there were no hints. For example, when a user tries to create a new transaction, categories and accounts are mixed in the ”from” section, making the user confused.

2.1.3 Wallet

The Wallet[3] application was developed by a Czech startup company BudgetBakers. As CoinKeeper, it is a finance management application. It has such functions suitable for our application:

- Ingoing and outgoing transactions.
- Accounts with different currencies, created by a user.

[2] https://about.coinkeeper.me/
[3] https://web.budgetbakers.com/

- Categories, assigning them to transactions.
- Budgets (constant payments during periods).
- Goals.
- Debts.
- Shopping lists.
- Importing data from real bank accounts as well as exporting data from accounts created in the application.
- Collaborative (group) finance management.

Also. the application allows a user to buy a premium subscription that unlocks such functions as:

- Automatic synchronization with banks.
- Insightful reports, tips.
- Unlimited account number.
- Advanced budgets.

In comparison to CoinKeeper, Wallet has many more features in the free version. As for the user interface, it is minimalistic and friendly with using of material design style. However, the application itself has a huge amount of features with the same interface, so a user can be a little confused and it takes time to figure out and remember how to use everything.

2.1.4 Comparison Table

Feature	**Wallet**	**George Go**	**CoinKeeper**
Transactions	Yes	Yes	Yes
Categories	Yes	Yes	Yes
Accounts	Yes	Yes (only ČS)	Yes
Statistics, visualization	Yes	Yes	Yes
Bank data import	Yes	No	Yes
Bank synchronization	premium	Yes (only ČS)	premium
Goals	Yes	No	No
Debts	Yes	No	No
Shopping lists	Yes	No	No
Collaboration	Yes	No	No
Application rating	1	3	2

Table 2.1: Application comparison

2.1.5 Conclusion

Taking everything into account, the Wallet application was chosen as the best one due to having a big amount of features in the free version. George Go was considered to be the worst one because it is not even a finance management application, but it has some suitable features. CoinKeeper is the golden middle: it has basic functionality and a fine user-friendly interface. The Wallet application was taken as a basis, but its functionality is going to be reduced.

2.2 Research of Front-end Technologies

A large amount of JavaScript frameworks exist these days due to their short release cycle and other historical reasons. [38]

The main criteria are:

- Fast learning curve, so developers can learn technology in a minimum amount of time.
- Good community support and popularity, so developers can rely on the stability of the tool they are using.
- Outstanding performance, so users can have their best experience even if they have to deal with a big amount of data. Moreover, it increases development and testing speed.

I have already worked with some of the front-end frameworks, however, I have tried to keep the analysis unbiased by sticking to external sources and comparing each of the frameworks, despite having my own preferences.

The three most popular front-end frameworks [39] were analyzed that are: React, Angular and Vue.js.

2.2.1 Overview

In this subsection, we are going to describe some basic information about the frameworks.

2.2.1.1 Angular

Angular is an open-source web framework based on TypeScript created by Google employees. It allows developers to create component-based scalable applications that can be maintained by small and large teams. It includes various libraries for essential front-end features like routing (Angular Router), forms management (Angular Forms), or client-server communication (Angular HttpClient). It also provides different tools for testing. One of Angular's key features is straightforward maintaining and updating of the project that decreases the probability of having bugs. [6]

2.2.1.2 React

React is a component-based JavaScript library for building UIs. It uses JSX and styled components, based on usual HTML and CSS respectively. It was developed and is maintained by Facebook employees. It is an open-source project and it has its own repository on GitHub. One of React's key features is one-way data-binding. In contrary to Angular, it lacks the essential functionality like routing or its own HTTP client, so additional libraries are required to be installed (like React Router, Redux, or Axios). There also exists the React Native framework for mobile platforms that is almost identical to React and maintained by the same developers.

2.2.1.3 Vue.js

Vue.js is an open-source progressive component-based JavaScript library that can be integrated into the existing websites for custom components or be used as a basis of a full-fledged front-end application, thanks to Vue CLI. It was developed by Evan You and now is maintained by the community, led by him. They also have a repository on GitHub that everyone can contribute to. In comparison to React or Angular, it provides more flexibility and choice in coding style, however, it can lead to disorganization in the team if no standards of coding are designed. [48] As React, Vue.js is only a library, so additional tools like Vue Router, Vuex, or Axios should be installed.

2.2.2 Community Support

Community support of our chosen framework and libraries will be analyzed in this subsection. GitHub repository parameters were taken, such as open to closed issues ratio, stars, and other information found to consider the popularity and support among the community.

2.2.2.1 Angular

It is stated that Angular has a large community of developers and is backed by the Google developer team, so the project is always being contributed to. Also, you can always find some help from the Angular specialists on the web [4].

Let's take a look at Angular GitHub statistics for the 22nd of April, 2022.

– Opened to closed issues ratio: 1300 / 22500 = 0.0578. [21]

– Stars: 80,854. [21]

– Number of downloads: weekly 2,544,667 for April. [3]

The star and download ratio is normal (0.032), compared to other frameworks.

2.2.2.2 React

It is stated that React has a large community of developers and is backed by the Facebook developer team, so the project is always being contributed to. React documentation suggests visiting stackoverflow.com and other communities for any help [49]

Let's take a look at React GitHub statistics for the 22nd of April, 2022.

- Opened to closed issues ratio: 676 / 10,675 = 0.0633. [22]
- Stars: 186,792. [22]
- Number of downloads: weekly 13,424,756 for April.

The star to download ratio is low (0.014), compared to other frameworks.

2.2.2.3 Vue.js

It is stated that Vue.js has a developing community that is not backed by any corporate giant developer team, but the project is being maintained and developed efficiently and quickly. They have their own forum, where you can ask questions or report bugs. [43]

Let's take a look at Vue.js GitHub statistics for the 22nd of April, 2022. Currently, they have two repositories: one for Vue 2.x.x [45] and one for Vue 3.x.x [44]. Let's sum everything up and get statistics.

- Opened to closed issues ratio: (329 + 364) / (9,379 + 2,515) = 0.058. (Vue.js 2.x.x repo even has 329/9,379 = 0.0351) [22]
- Stars: 224,505. [22]
- Number of downloads: weekly 2,841,030 for April.

The star and download ratio is high (0.079), compared to other frameworks.

2.2.2.4 Conclusion

Measurement	**Angular**	**React**	**Vue.js**
Open-to-closed-issues ratio	0.0578	0.0633	0.058 (0.035 for Vue 2.x.x)
Number of downloads	2,544,667	13,424,756	2,841,030
Stars	80,854	186,792	224,505
Stars/Download ratio	0.032	0.014	0.079
Rating	2	3	1

Table 2.2: Framework community support comparison

As can be seen from the table, opened to closed issues ratio is almost the same for all the networks. However, the stars-to-download ratio is significantly higher than of other frameworks. To add, Vue.js has its own forum, but neither React nor Angular has. Vue.js seems to be the best option regarding this criterion.

2.2.3 Performance

In this section, we are going to compare framework performance with help of the results of the conducted experiment, where some operations were carried out on a table with a big amount of rows. In particular:

" Test applications have a data table and a menu, with five functions:

- Create a table with 1000 rows.
- Create a table with 10,000 rows.
- Add 1000 rows to the table.
- Add 10,000 rows to the table.
- Delete all rows.

Each test measured the time to load the home page, and the time it took to perform the functions called by buttons. Buttons "Create 1000" and "Create 10,000" were clicked twice, to see if there were any differences when rerunning the same function. Before clicking the "Create 10,,000", the page was cleared by clicking the clear button, so that the new 10,000 rows would be created on the empty page. Speeds were measured in milliseconds. "[1]

2.2.3.1 Angular

In the next table, we can see measurement results for Angular that were collected during conducting the experiment.

Action	Test 1.	Test 2.	Test 3.	Average
Load Page	389.6	393.9	425.7	403
Create 1000	404.4	439.5	306.6	384
Re-create 1000	420.2	287.3	284.4	331
Add 1000	288.5	237.9	234.8	254
Create 10,000	6508.6	6268	6306.6	6361
Re-create 10,000	7726.3	7735	7218.5	7427
Add 10,000	7823	7243.2	7804.2	7623
Remove all	2315.9	2205.9	2064.9	2196

Table 2.3: Angular performance measurements [1]

2.2.3.2 React

In the next table, we can see measurement results for React that were collected during conducting the experiment.

Action	Test 1.	Test 2.	Test 3.	Average
Load Page	259.4	268.8	279.4	269
Create 1000	400.9	400.6	459.1	420
Re-create 1000	174.2	206	191.2	190
Add 1000	364.2	364.5	380.5	370
Create 10,000	2787.1	2978.9	2831.7	2866
Re-create 10,000	1181.7	1201.3	1145.7	1176
Add 10,000	3888.7	3906.7	3766.1	3854
Remove all	589.4	619.9	611.3	607

Table 2.4: React performance measurements [1]

2.2.3.3 Vue.js

In the next table, we can see measurement results for Vue.js that were collected during conducting the experiment.

Action	Test 1.	Test 2.	Test 3.	Average
Load Page	233.3	264.3	240.44	246
Create 1000	248.1	213.5	234.2	232
Re-create 1000	125.6	111.8	112.6	117
Add 1000	193.7	180.7	180.2	185
Create 10,000	1446.8	1448.1	1446.7	1447
Re-create 10,000	553.2	533.9	553.9	547
Add 10,000	1423.2	1452.3	1433.1	1436
Remove all	567.6	334.8	613.9	505

Table 2.5: Vue.js performance measurements [1]

2.2.3.4 Conclusion

In the table below the worst performance is marked with the red-colored cell, the medium is marked with yellow and the best one is marked with green.

Two last rows are used for the average analysis of framework performances. The total row shows how much time was spent during all the actions and the multiplier row shows the total running speed compared to the fastest framework (Vue.js in this case).

Action	Angular	React	Vue.js
Load Page	403	269	246
Create 1000	384	420	232
Re-create 1000	331	190	117
Add 1000	254	370	185
Create 10,000	6361	2866	1447
Re-create 10,000	7427	1176	547
Add 10,000	7623	3854	1436
Remove all	2196	607	505
Total	24978	9752	4715
Multiplier (Total)	5.30	2.07	1.00

Table 2.6: Framework performance comparison [1]

From this table, it is seen that Vue.js is generally 2x faster than React and 5x faster than Angular.

2.2.4 Learning curve

In this section, we are going to look at each framework's documentation and syntax and try to conclude which one is the fastest to learn. Since all these JavaScript frameworks share almost identical structures and workflows, it was possible to compare them within 4 same topics: syntax, architecture, state management, and component lifecycle.

2.2.4.1 Angular

Angular has descriptive documentation explaining the framework. Different tutorials for the new developers can be found as well as the more in-depth documentation for more experienced Angular developers.

Angular template syntax is an extension of HTML, except for the ¡script¿ tag that is forbidden due to the prevention of XSS attacks[13]. JavaScript content can be inserted into the view with curly braces.

Angular applications consist of NgModules. Every application has a root module conventionally named AppModule that allows for launching the application. Organized code can be separated into individual modules to organize the development of large applications. Angular partly uses the MVC pattern, where the view layer is represented by templates and the controller layer is presented by components.

Angular supports two-way binding by listening to child component events and receiving data from them and passing props to them from parents.

Angular offers an advanced system of lifecycle hooks, so developers can control different stages of component lifecycle. [5]

2.2.4.2 React

React has documentation that is divided into two approaches. For the first one, it is proposed to write your own application and at the same time learn some theory by applying it to individual parts of the application. The second one allows developers to learn React step-by-step with theory from basic terms to advanced.

React uses JSX syntax which is a mixture of HTML and JavaScript. It may remind one of template language with having all the JavaScript. [20]

JSX is used to build components that are plain objects. React has a virtual DOM that is used to update the actual browser DOM. Virtual DOM is kept in the memory while the actual one is synced with help of libraries like React DOM[42].

React has a one-way data-binding that is represented by passing props to elements. They are immutable and cannot be passed upward to parents. Individual components can have a state by using the useState hook, but it is not necessary.

React does not have a full overview of lifecycle hooks and they are spread across the documentation.

2.2.4.3 Vue.js

Vue.js has detailed documentation full of examples on its homepage. Essential concepts are explained and illustrated very thoroughly, even containing examples in JSFiddle.

Vue.js uses HTML-based template syntax. Each Vue.js template is valid HTML. Data can be inserted into templates with curly-brace syntax[37]. At the moment, Vue.js offers two types of templates: using Options API and Composition API. As for me, Options API is more novice-friendly as it offers an explicit division of individual component options like data, computed properties, methods, etc.

Vue.js uses component-based architecture with building a tree of components starting from the root one. All the components provide the same functionality for defining a template, a data section, computed properties, and methods.

Vue.js has a two-way data binding with listening to child component events from parents and passing props to them. It is also possible to access child attributes by using refs[36].

Vue.js has a descriptive page describing all the lifecycle hooks of a component [25].

2.2.4.4 Conclusion

Unfortunately, it is not possible to determine the learning curve quantitatively as every developer is an individual with their own preferences and background. Some sources mention that Vue.js is generally easier to learn [7]. If we compare Vue.js to Angular, we will see that a developer also has to learn TypeScript, Vue.js does not require this knowledge. If we compare Vue.js to React, Vue.js has a better documentation and more structurized boilerplates that allow user to start as fast as possible. Also React.js requires the user to learn JSX in addition to HTML, JavaScript and CSS.

In my opinion and from my experiences, Vue.js provides more friendly and descriptive documentation than React or Angular. Moreover, Vue.js has its own forum with developers sharing their experiences, but neither React nor Angular has.

2.2.5 Vue.js UI Frameworks

In this section, various UI frameworks are going to be analyzed. It was decided to use a framework because it can save us a lot of time for creating a prototype [40], because there are a lot of already implemented UI components with complex logic like calendars, modal windows, or configurable tables, so we do not have to waste our time writing them.

2.2.5.1 Vuetify

Vuetify is a UI framework that can be integrated into an application that is already under development. It has long-term and enterprise support. It provides different components with Material Design. Vuetify is also surrounded by a large community of developers and has detailed and meaningful documentation. However, it does not provide any CLI and has some problems with UI elements customization [16].

2.2.5.2 PrimeVue

PrimeVue is as well a UI framework, however, it provides not only Material Design themes. It is backed by an enterprise team. However, they have a poor set of UI elements that are not highly customizable or they have a lack of functionality, compared to other frameworks.

2.2.5.3 Quasar Framework

Quasar can't be integrated into a working app, and requires creating an application from scratch, however, it has some essential advantages:

1. It has its own CLI that helps initialize an application with a boilerplate structure [31].
2. A big number of useful UI elements with logic are already implemented, making the prototyping process really quick.
3. Very flexible framework in a sense of styling and using its features.
4. Can be easily used in Electron for desktop applications and Cordova for mobile applications, making the application prototype scalable.

Compared to other frameworks, it is backed by a single person and a community around. It also has descriptive and meaningful documentation and a small, but very active and solid community.

2.2.6 Conclusion

Quasar framework based on the Vue framework was chosen because it can help build the prototype front-end application as soon as possible. It already contains some UI elements with implemented logic, so there is no need in creating them. It also creates an application structure, represented by layer architecture, on project initialization. To add, it also has a small but very active community, so all the development problems can be solved quickly. Also, the Quasar application can be ported literally to any platform, so the scalability will be also good.

2.3 Requirements

After some applications were analyzed, requirements were derived from the analysis.

2.3.1 Functional Requirements

Here, the verb "manage" will be used to describe CRUD operations.

1. FR-1: Users can sign up and manage their profile.
2. FR-2: Users can manage their bank accounts.
3. FR-3: Users can manage their transactions.
4. FR-4: Users can manage their transaction categories.
5. FR-5: Users can manage their debts.
6. FR-6: Users can manage their goals.
7. FR-7: Users can manage their service subscriptions.
8. FR-8: Users can manage their collective bank accounts.
9. FR-9: Users can manage their groups with collective bank accounts.
10. FR-10: Users can view a graphical representation of their transaction statistics.
11. FR-11: System can send notifications about debts and subscriptions to a user.
12. FR-12: Users can paginate their transaction when viewing.

2.3.2 Use-Cases

Taking Functional and Non-functional Requirements into account, such use-case models were created. Most of them contain CRUD operations, essential operations for persistent storage.

The figure 2.1 demonstrates us derived use-cases for the account domain. Basic CRUD operations for an account are defined as well as CRUD operations for account transactions. We can see the implemented use cases on the page wireframes A.5 and A.6.

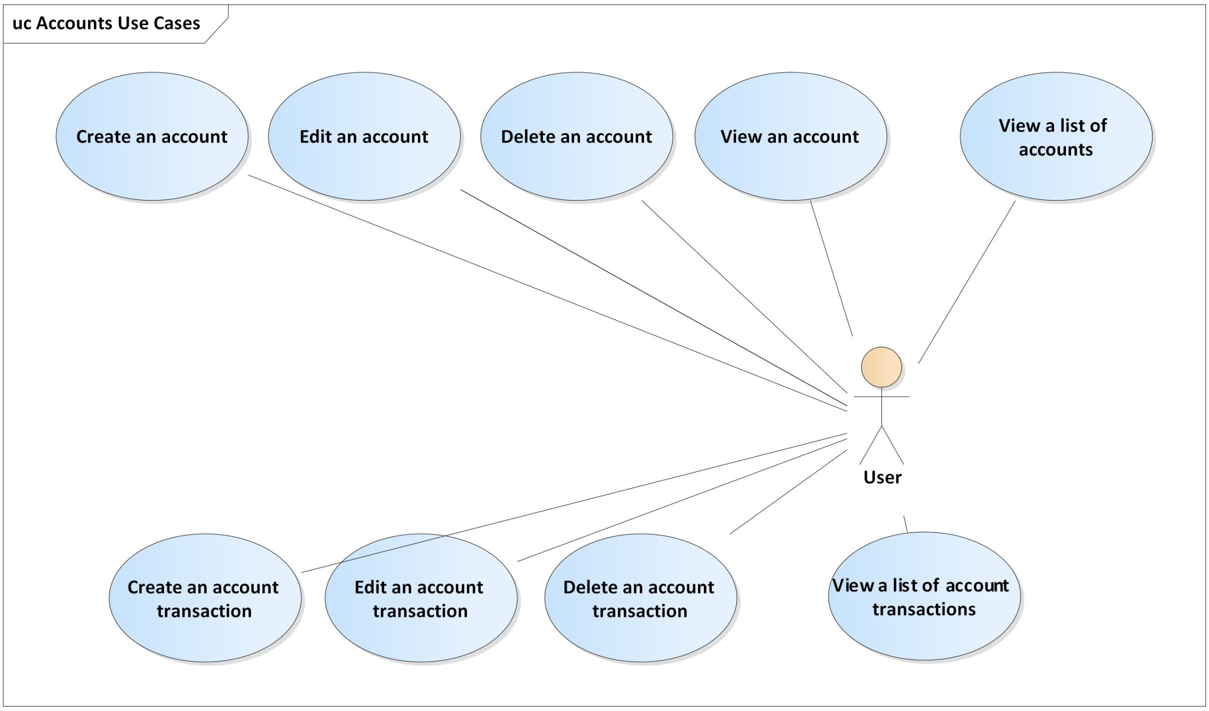

uc Accounts Use Cases
Create an account
Edit an account
Delete an account
View an account
View a list of accounts
User
Create an account transaction
Edit an account transaction
Delete an account transaction
View a list of account transactions

Figure 2.1: An account domain use-case model

Figure 2.2 demonstrates us derived use-cases for the authorization domain. As for now, users can create a profile to start using our application, restore password in case it is lost, or view their own profile. More use cases could be added after the MVP is created. We can see the implemented use cases on the page wireframes A.1 and A.2.

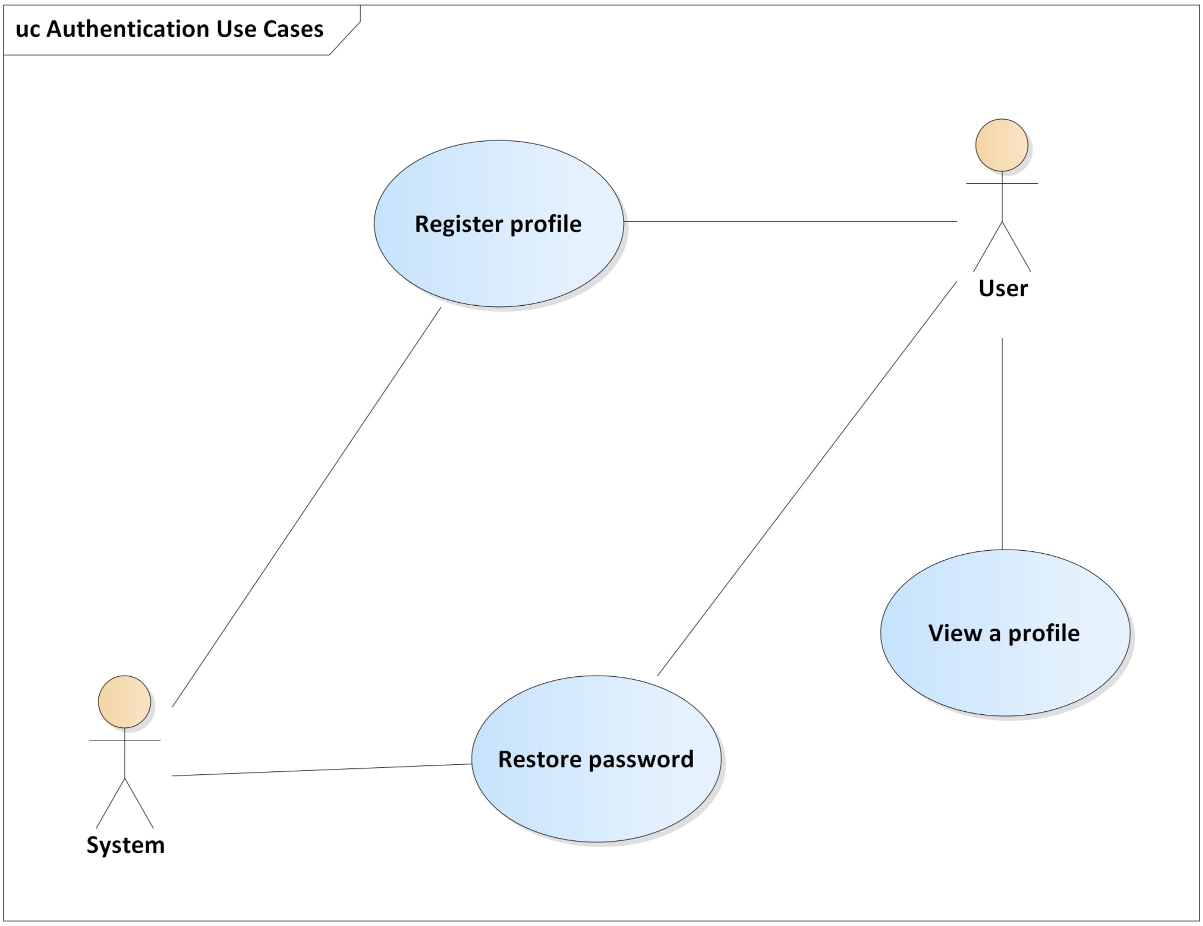


Figure 2.2: An authorization domain use-case model

Figure 2.3 demonstrates us derived use-cases for the category domain. As for now, CRUD operations for user categories are available as well as getting a list of the categories that are default for all the users. A table with implemented use cases can be seen on the page wireframe A.14.

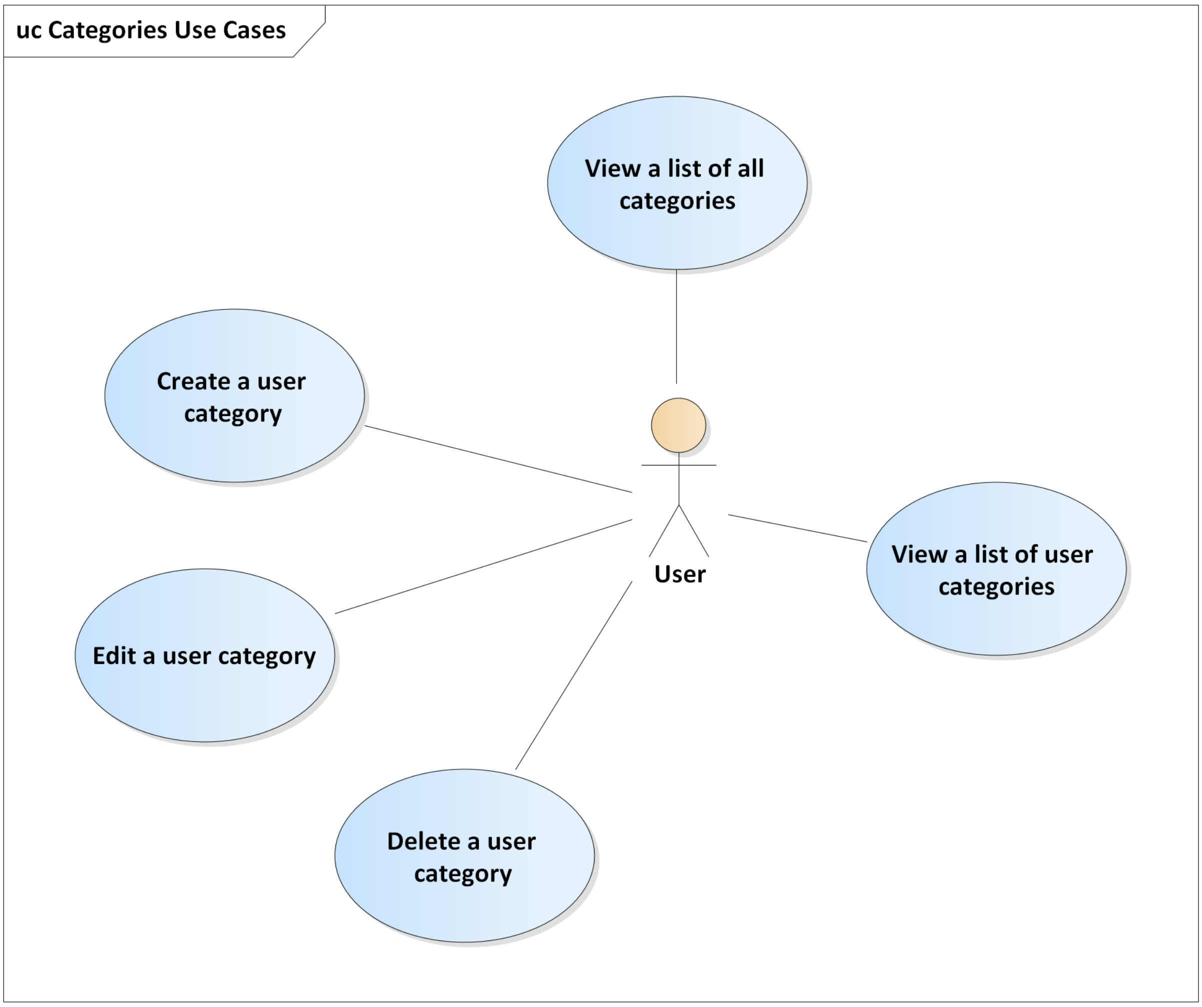


Figure 2.3: A category domain use-case model

Figure 2.4 demonstrates us derived use-cases for the dashboard domain. We can see the implemented use cases on the page wireframe A.3.

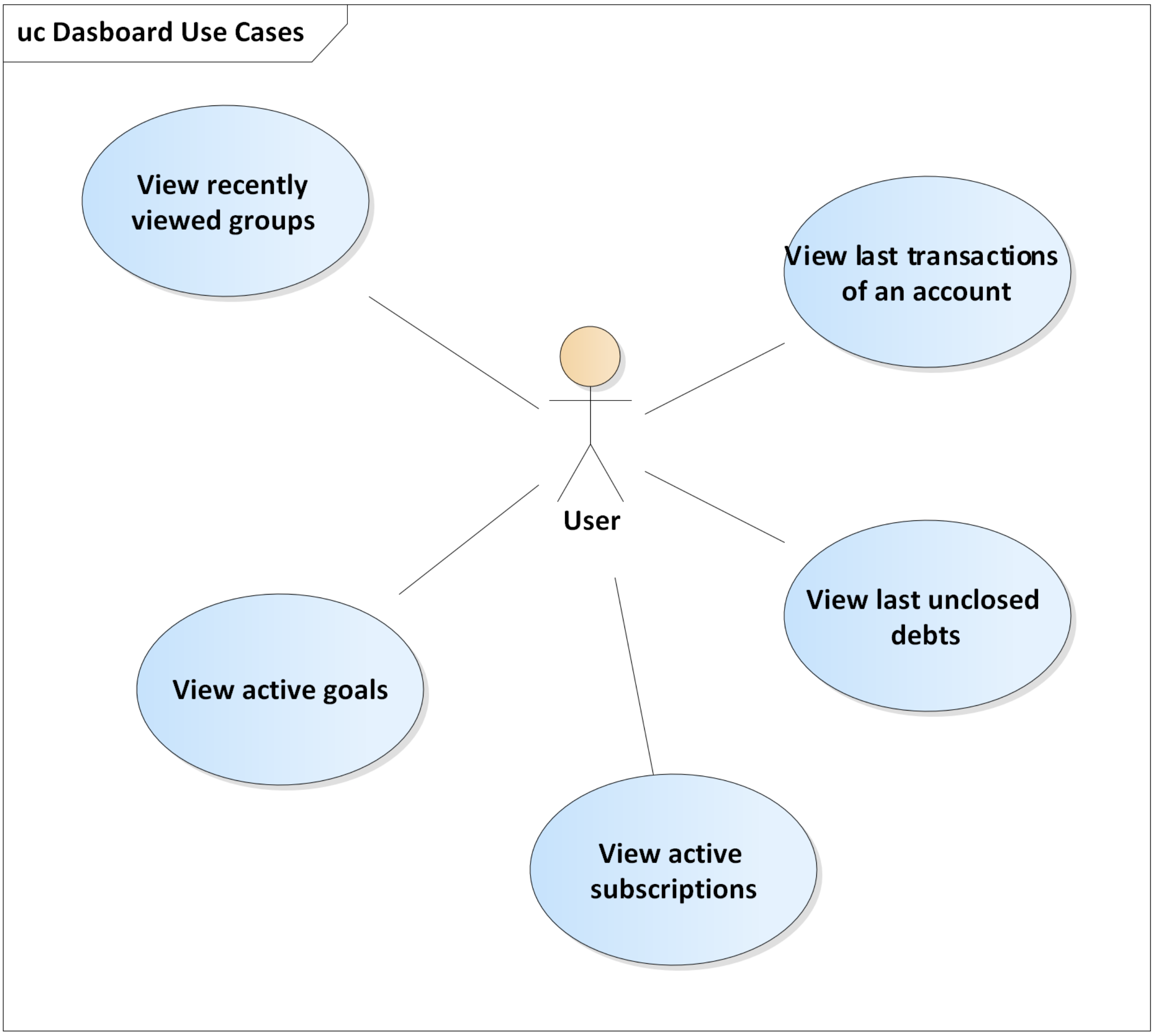

uc Dasboard Use Cases
View recently viewed groups
View last transactions of an account
User
View last unclosed debts
View active goals
View active subscriptions

Figure 2.4: A dashboard domain use-case model

Figure 2.5 demonstrates us derived use-cases for the debt domain. Only basic CRUD operations are available. We can see the implemented use cases on the page wireframe A.10.

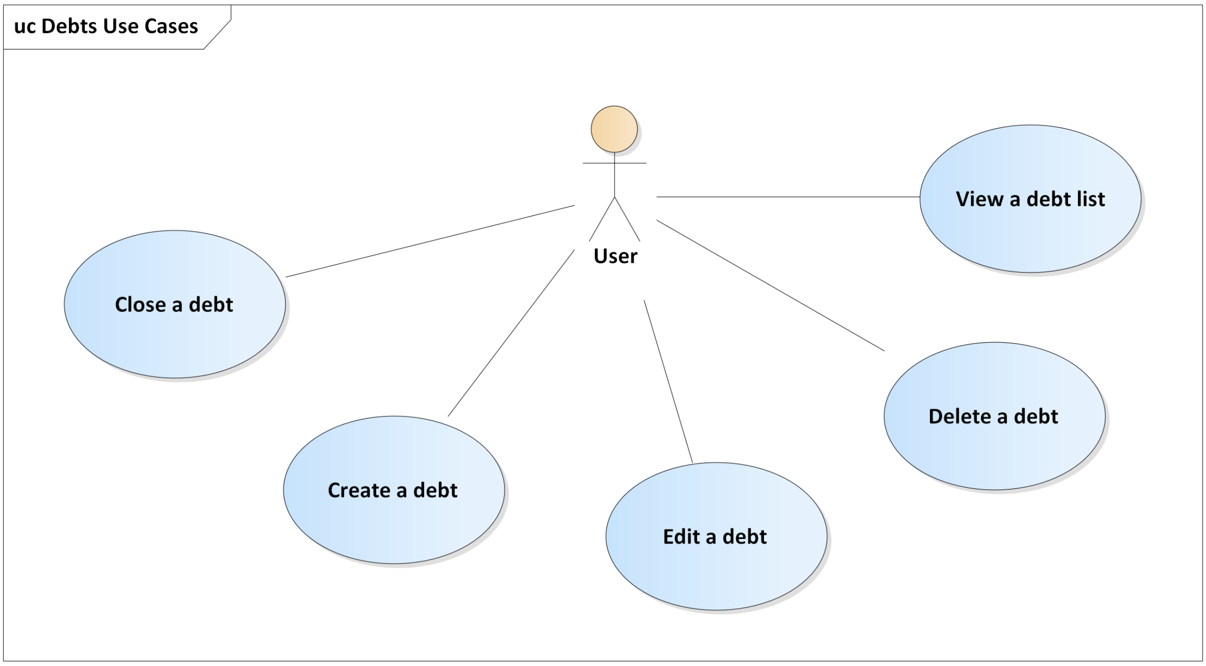


Figure 2.5: A debt domain use-case model

Figure 2.6 demonstrates us derived use-cases for the goal domain. Basic CRUD operations are available as well as a use case for adding or subtracting a particular amount of money invested in the goal. We can see the implemented use cases on the page wireframe A.11.

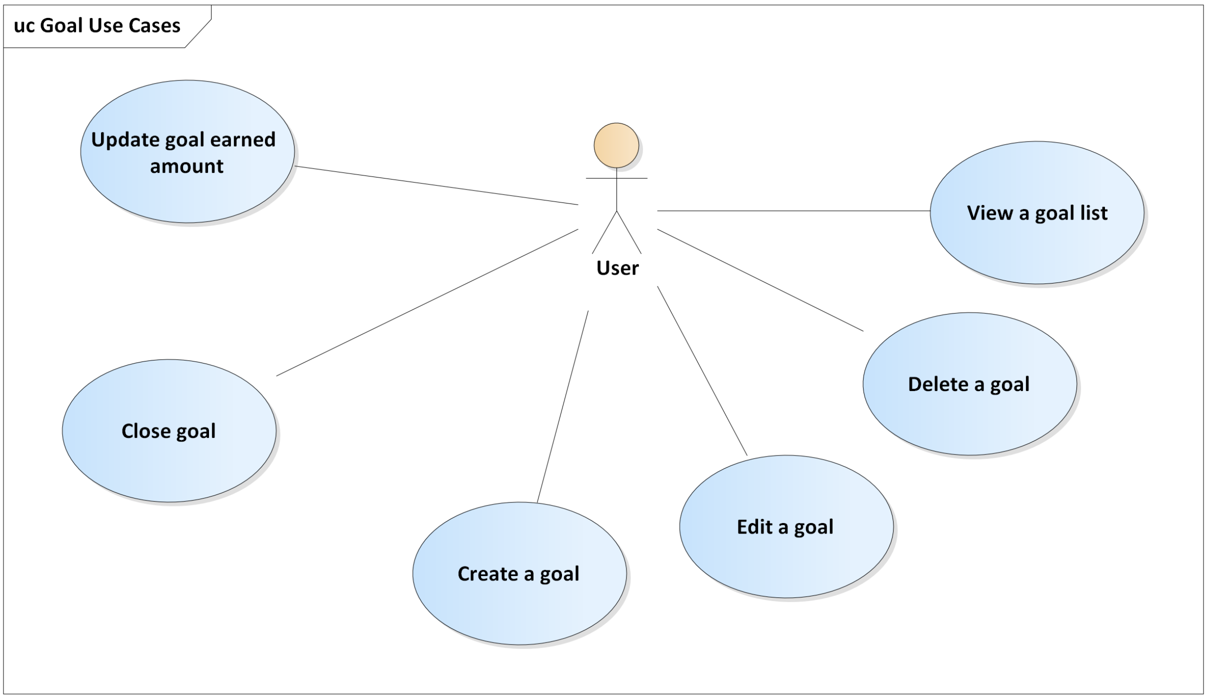


Figure 2.6: A goal domain use-case model

Figure 2.7 demonstrates us derived use-cases for the group domain. Basic CRUD operations are available. CRUD operations for group transactions are available. Also, a group can be managed with roles, where certain roles have different permissions. The use cases implemented in the UI are in the figures A.7, A.8 and A.9.

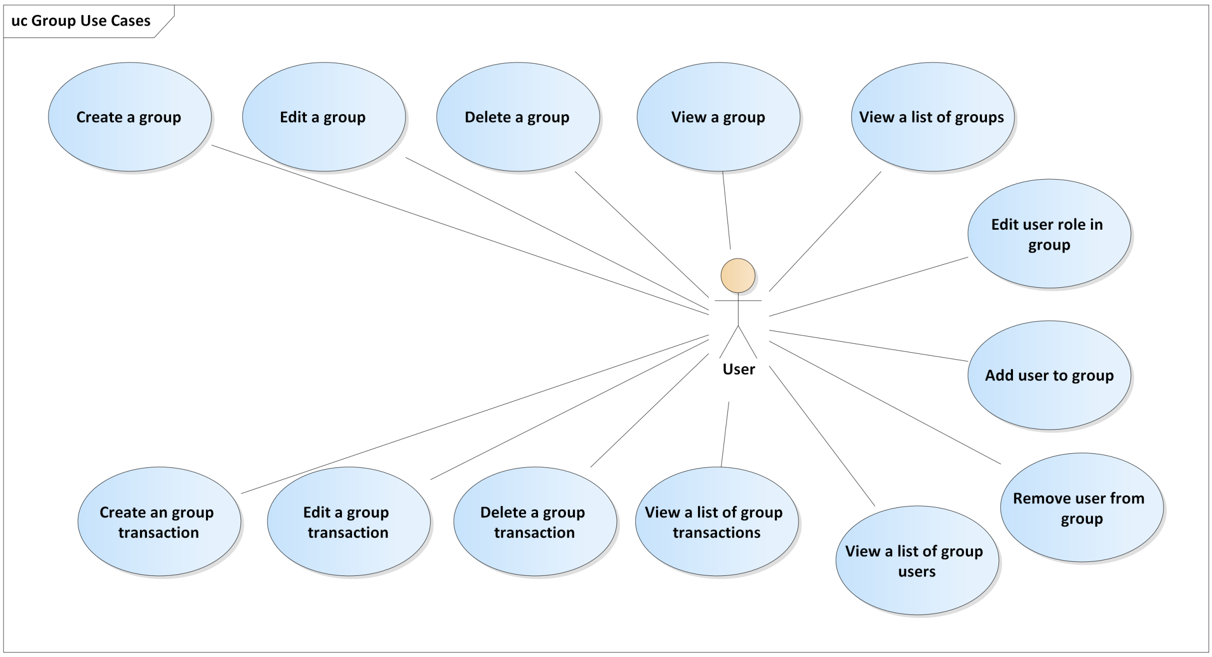


Figure 2.7: A group domain use-case model

Figure 2.8 demonstrates us derived use-cases for the group domain with respect to roles. The use cases implemented in the UI are in the figures A.7, A.8 and A.9.

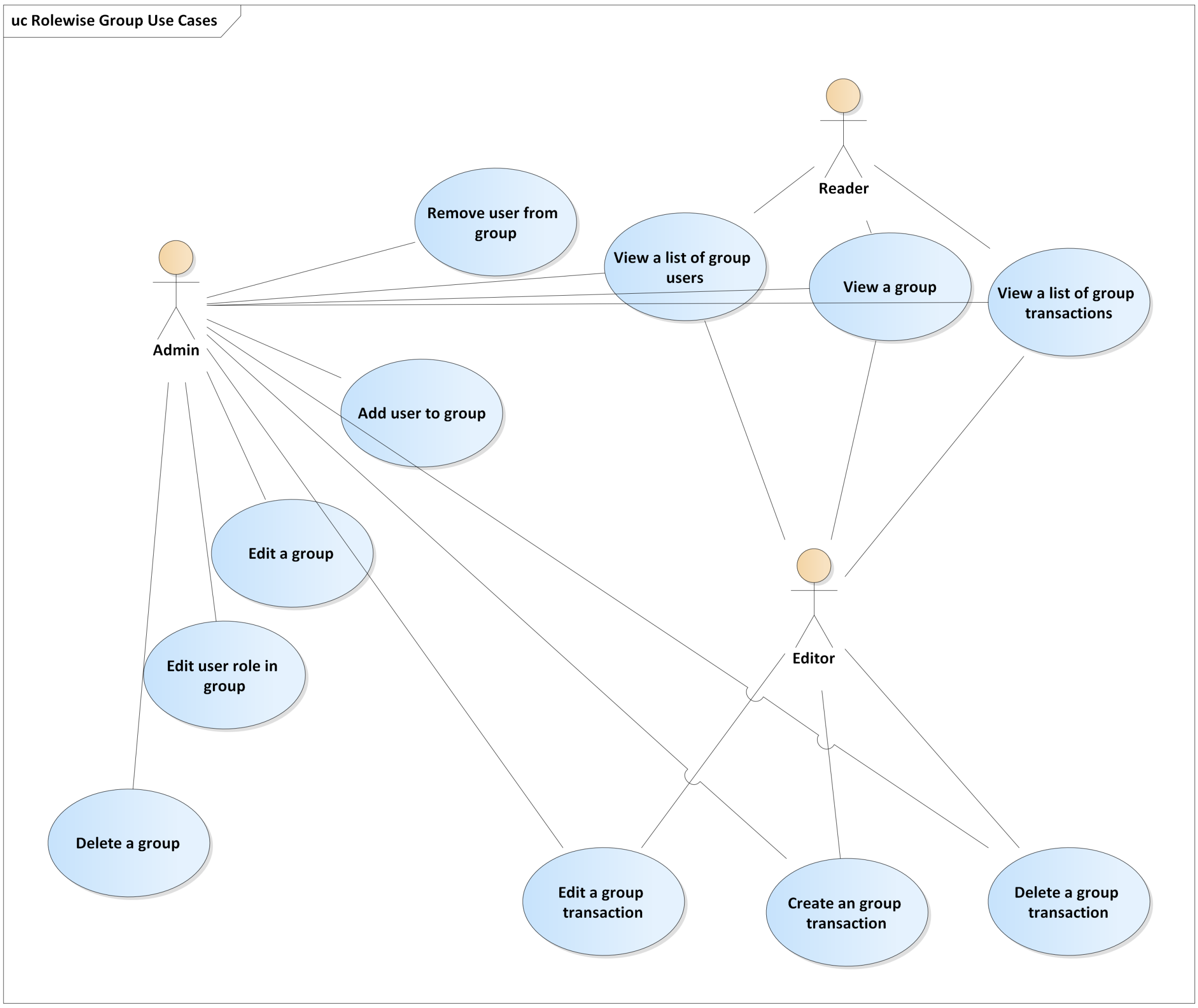


Figure 2.8: A rolewise group use-case model

Figure 2.9 demonstrates us derived use-cases for the subscription domain. Basic CRUD operations are available as well as a cron service can take subscription price money equivalent from an account that is associated with the subscription every certain period. The use cases implemented in the UI are in the figure A.12.

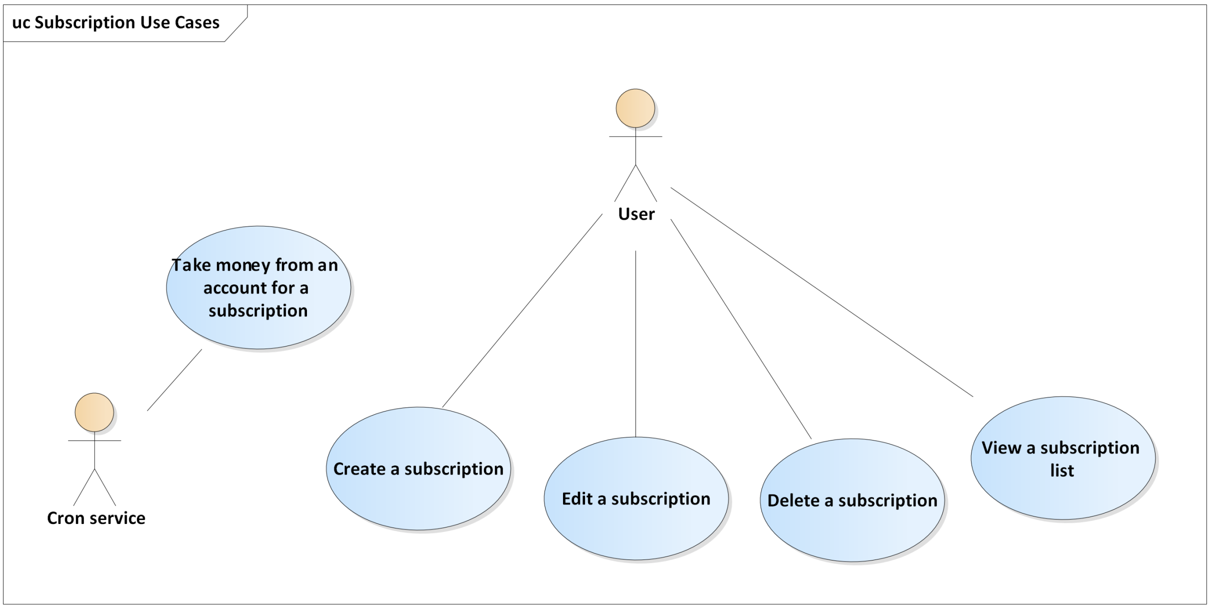


Figure 2.9: A subscription domain use-case model

Figure 2.10 demonstrates us derived use-cases for the global system action domain. Actually, it has only one case of sending a notification that a user should see if logged into a system. The notification window design can be seen in the page wireframe A.4.

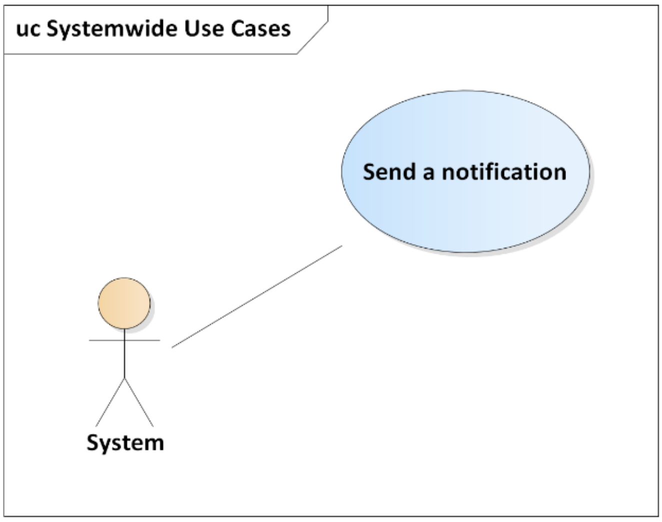


Figure 2.10: A global system action domain use-case model

2.3.3 Non-Functional Requirements

1. NR-1: Back-end server should have a monolithic architecture.
2. NR-2: Back-end server should be implemented in C# .NET framework[4] with PostgreSQL database[5].
3. NR-3: Front-end client should be implemented in Quasar framework[6] for Vue.js 3[7].
4. NR-4: Communication between back-end server and front-end client should be RESTful [8].
5. NR-5: JSON[9] should be used as data interchange format.
6. NR-6: Users can sign up either with email and password or with social networks.
7. NR-7: Only popular modern web browsers should be supported. Such as Google Chrome, Safari, Edge, Mozilla Firefox[10].
8. NR-8: Authentication should be implemented with JWT tokens.[11]
9. NR-9: Code should contain as few as possible duplications, so it's easier and cheaper to maintain. [27]

2.3.4 Analytic Domain Model

An analytic domain model is used to determine which business entities will be used in the application and how they relate to each other. The front-end client is using the same data schema as DTOs in the back-end API [33] to follow the principle of least astonishment. A currency enumeration type is created because we operate only with certain currencies that are supported by the back-end server. The same is true for the role and notification enumeration types. Users can have no entities or no relations at all, so all the entities related to users have 0..* multiplicity. GroupRole entity is used to represent a relation of a user to role in a certain group. Group to GroupRole association has 1..* association because group cannot exist without users. Notification parameters are coded into strings containing a JSON because different notifications can have different parameters.

[4] https://dotnet.microsoft.com/en-us/

[5] https://www.postgresql.org/

[6] https://quasar.dev/

[7] https://v3.vuejs.org/

[8] https://restfulapi.net/

[9] https://restfulapi.net/introduction-to-json/

[10] https://gs.statcounter.com/browser-market-share

[11] https://auth0.com/docs/secure/tokens/json-web-tokens

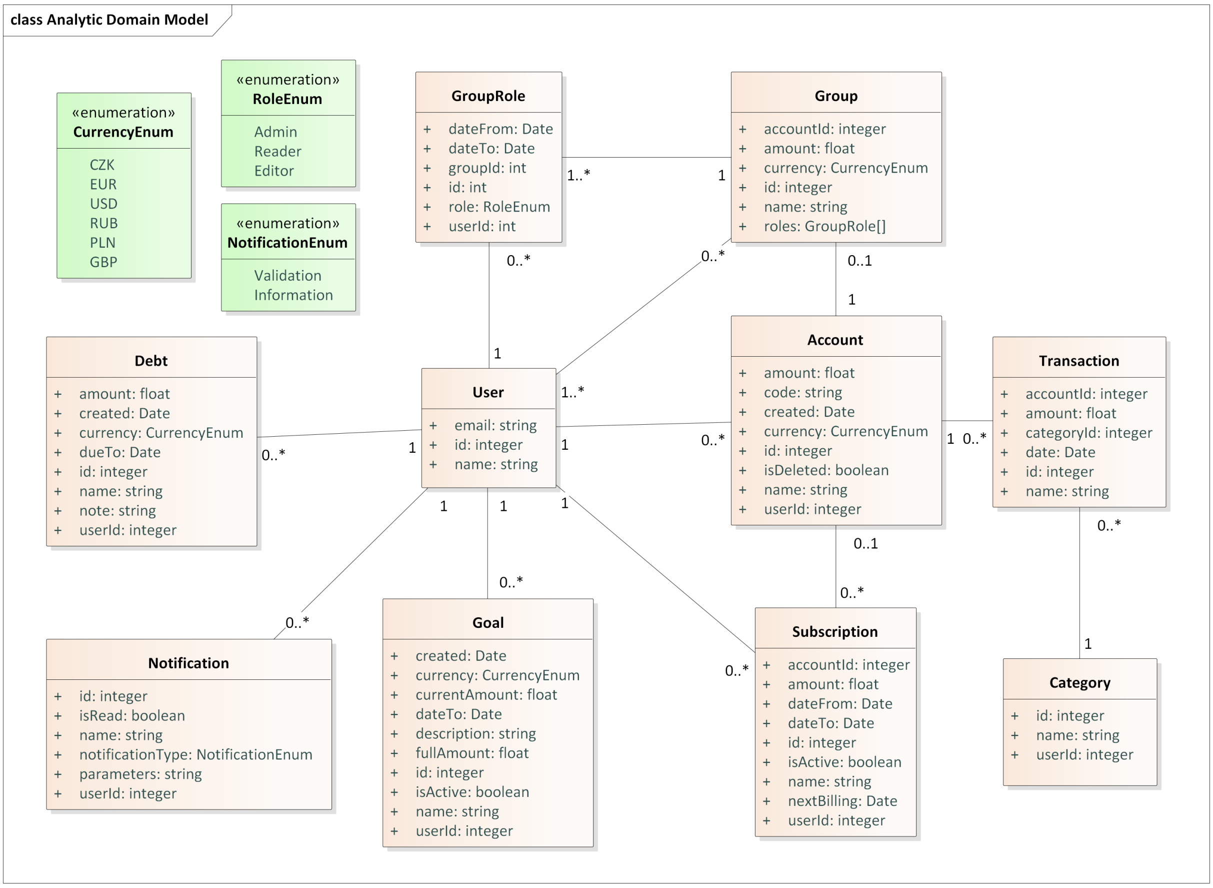


Figure 2.11: An analytic domain model

Chapter 3

Design

The application architecture is going to be described in this chapter. It was decided to use the Quasar framework based on the Vue.js library to build a scalable, maintainable, and robust web application. Firstly, the common front-end application architecture will be described to point out essential components. Secondly, UI/UX design will be presented that was created following the best practices.

3.1 Single-Page Application

Our front-end client was implemented as a SPA. It means that an initial HTML document will be generated with our Vue.js framework and will dynamically change its body while communicating with the back-end server and receiving data from it.

As a consequence, a user can use our application without using a lot of traffic for loading each of the pages. Instead, the front-end client sends small requests to the back-end server and receives small responses. All this makes the application more dynamic [35].

On the other hand, server-side rendering allows for better search engine optimization, because a page is rendered as a whole and not as a bundle, therefore search engine bots can analyze more accurately all the website [2].

As we do not care for SEO, but we do care for optimization and smooth user experience, the SPA approach was chosen.

3.2 Client

In this section, the common front-end client infrastructure design will be described.

3.2.1 View Layer

View Layer is responsible for interaction with a user and displaying an application state. There are two most commonly used types of interface: GUI (graphical user interface) and CLI (command-line interface).

LiterateConsumer application uses a graphical interface as it is more user-friendly and our application is designed for common users.

View Layer is used as a bridge between a user and a state of an application. The user does some inputs that can change the state. And the state provides the displayed output for a user.

3.2.2 State Management

State management is responsible for storing data required for our application. Our application view mostly depends on the current state. It is used for storing authorization information or the data that is required for user actions.

Store modules were created for each of the business domain entities shown in the figure 2.11.

A state contains all the data related to the application state and is used as a ”single source of truth” for all the application components. If the state changes, it

The state can also be used for communication between different components.

3.2.3 API Client

API Client is required to communicate with a back-end server. It can be used for plain half-duplex HTTP requests or for keeping a full-duplex WebSocket connection.

It is used for user authorization and retrieving data from the server. API Client should be used only to update the State layer, so it subsequently updates a graphical representation with the help of reactivity.

3.2.4 Component Diagram

Here we can see the front-end component diagram. It shows us the structural relations between components of the application.[41] A back-end server is represented here as a separate subsystem the client communicates with.

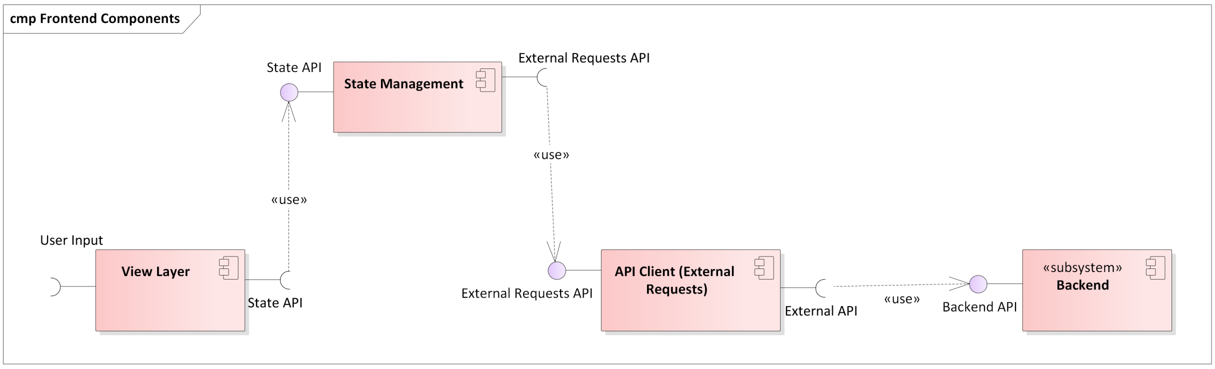


Figure 3.1: A client component diagram

3.3 UX Design

In this section, the creation of UI/UX design will be described. Figma tool was used to create all the layouts. The icon system comes from Material Design because we have chosen Quasar Framework which UI styling follows the Material Design principles. Also, all the use-cases were implemented in the graphical representation.

3.3.1 Low Fidelity Design

Two kinds of layouts were designed: for the login screen and the dashboard.

The first one is very simple: it is just a form with some background and title, asking for the relevant data. The user always knows on what page he is and what we want from him. We can see such a layout in Figure 3.2

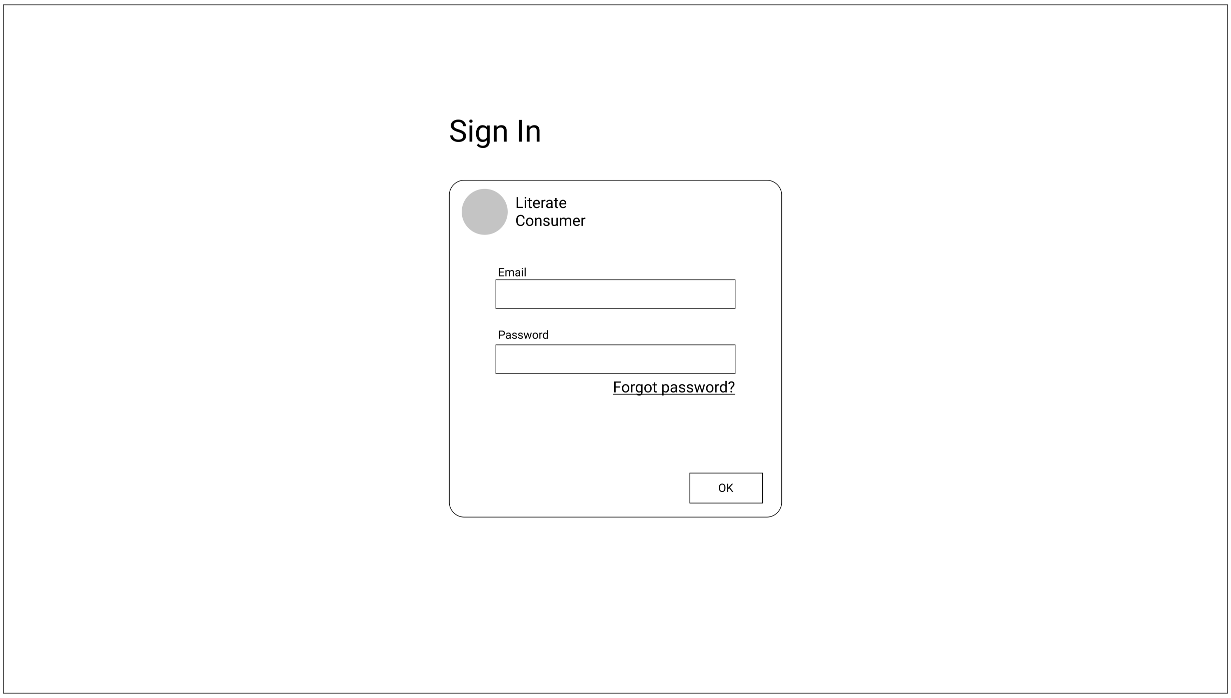
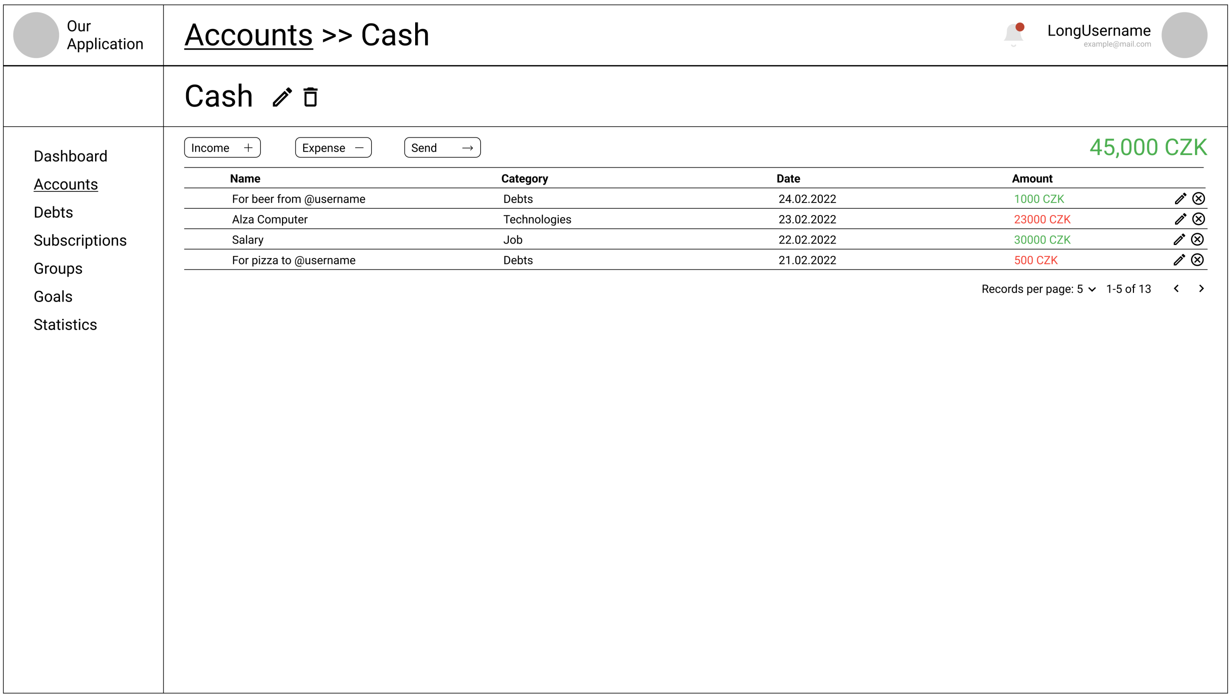


Figure 3.2: Sign in layout example

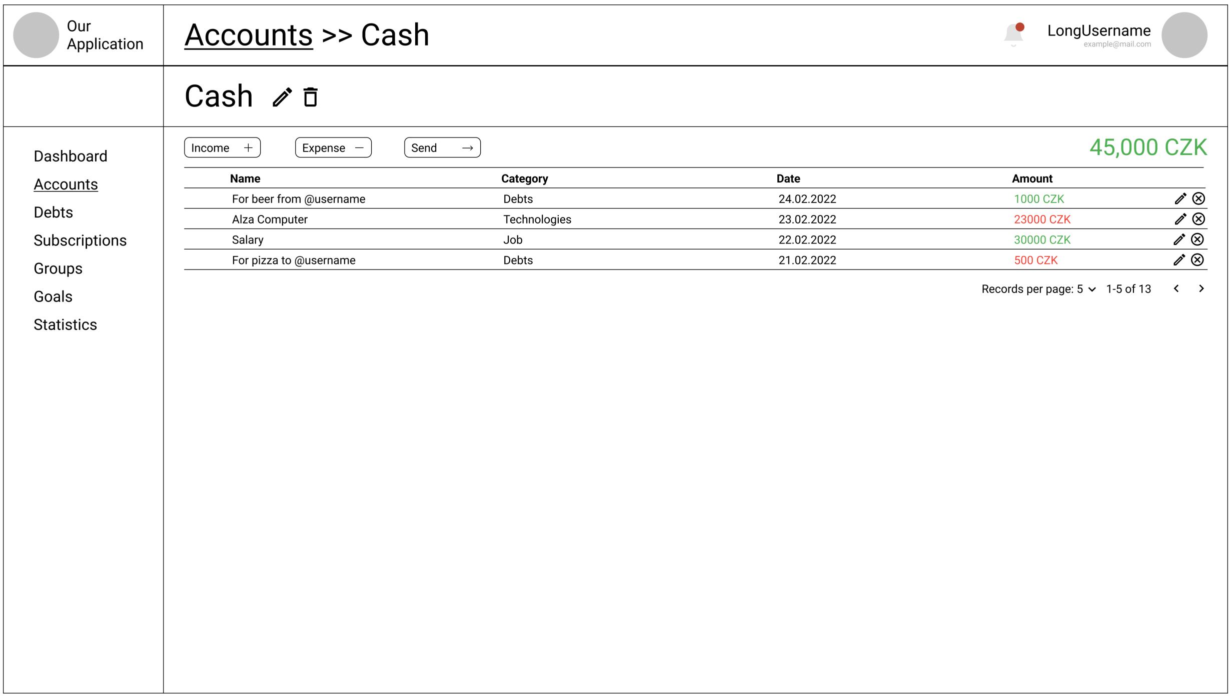
The second one has a more complex structure, so the left menu bar, as well as the breadcrumbs, are used as a navigation system. This allows users to quickly navigate across the application. That is essential because it is more likely for users not to use our application if they cannot do so. [23] An example of such a layout can be seen in Figure 3.3. Also, a search bar was designed to search across the application features or user entities. The user should always be able to know where he is in the program or should be able to navigate through it. [23]



Name	Category	Date	Amount
For beer from @username	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @username	Debts	21.02.2022	500 CZK

Figure 3.3: Dashboard layout page example

The dashboard home page was designed in such a way that users can quickly access the most frequently used functions such as creating transactions, closing debts, seeing their last transactions, and so on. We can see it in Figure 3.4. We have tried not to use many icons as their overflow is considered to be bad practice. One should prefer labeling over it. However, we have used some to make users look at the essential functions. [51]

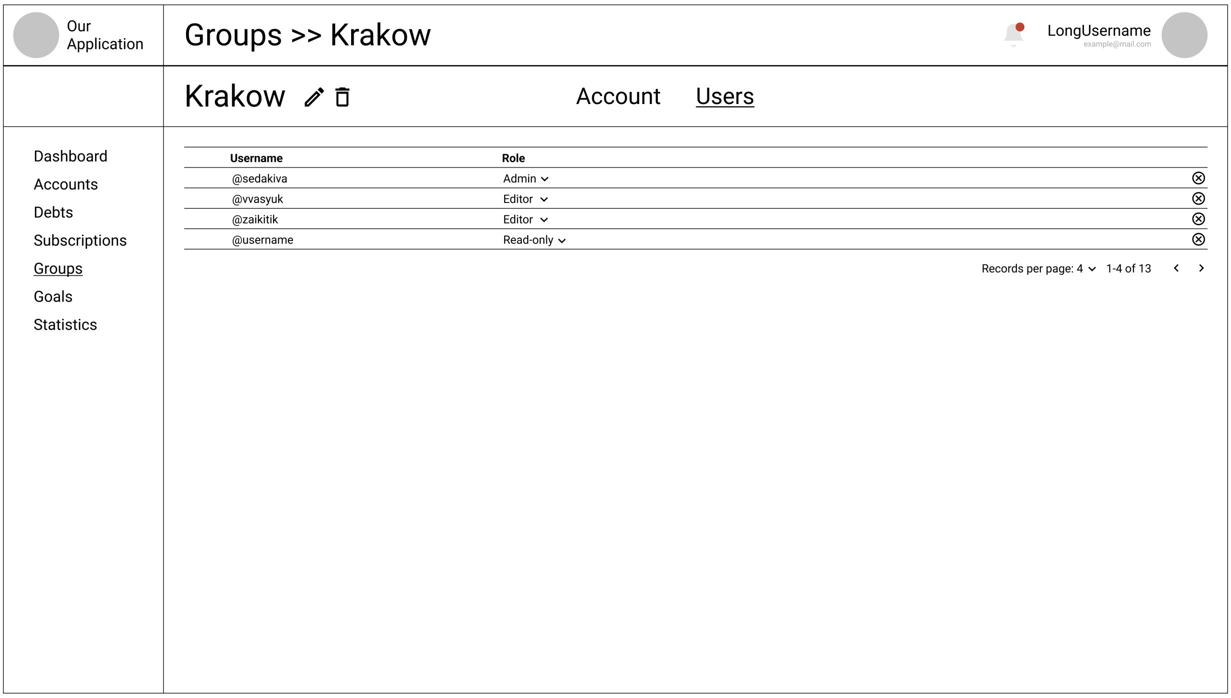


Name	Category	Date	Amount
For beer from @username	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @username	Debts	21.02.2022	500 CZK

Figure 3.4: Dashboard page example

Also, some modal windows and confirmation windows were added to the UI where appropriate. For example, for users' confirmation if they want to delete some entity. [10]

Most of the pages representing a list of entities without their own sub-entities are represented as items in the table (such as Categories, Debts, Transactions, and Goals). We can see them in the Figure 3.5.

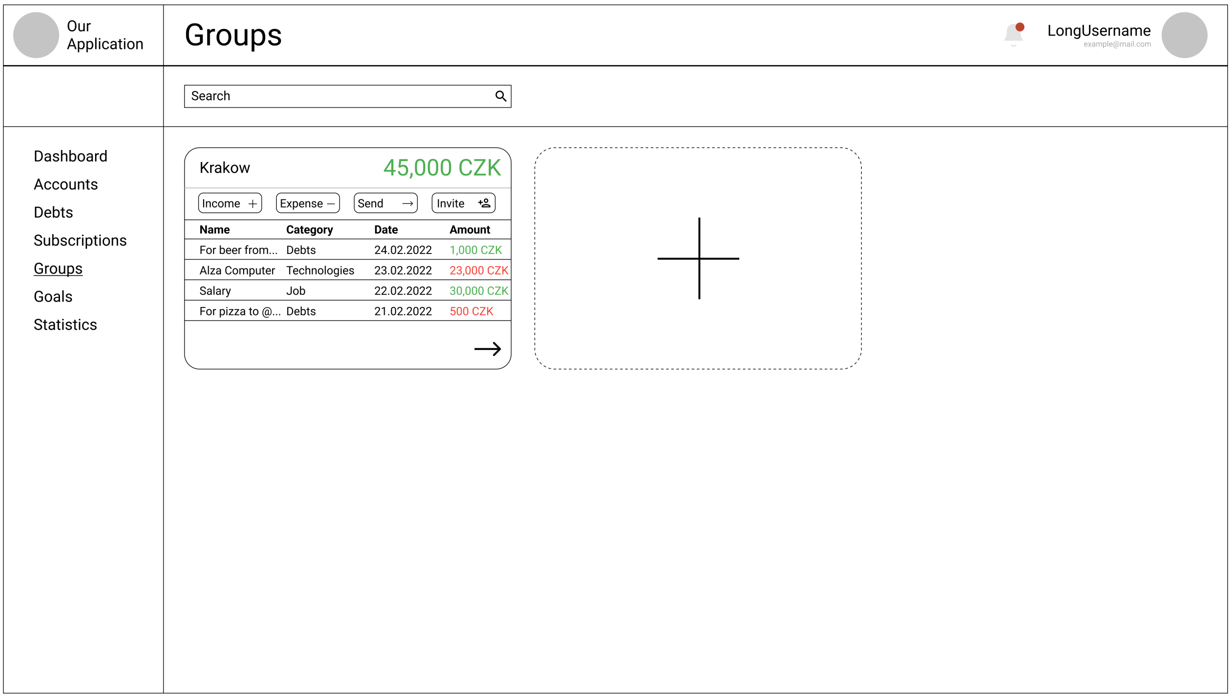


Username	Role
@sedakiva	Admin ⌄
@vvasyuk	Editor ⌄
@zaikitik	Editor ⌄
@username	Read-only ⌄

Records per page: 4 ⌄ 1-4 of 13 ‹ ›

Figure 3.5: Group user list UI example

More complex entities like groups or Accounts are represented as cards. They can be seen in Figure 3.6.



Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1,000 CZK
Alza Computer	Technologies	23.02.2022	23,000 CZK
Salary	Job	22.02.2022	30,000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

Figure 3.6: Group list UI example

3.4 Deployment

The deployment diagram shows the relations between hardware and software components in our system.[17] Basically, our server can be accessed from any PC or any mobile phone having an appropriate browser. Moreover, thanks to Quasar, the application can be ported to Cordova with minimal effort to create a mobile application. [46]

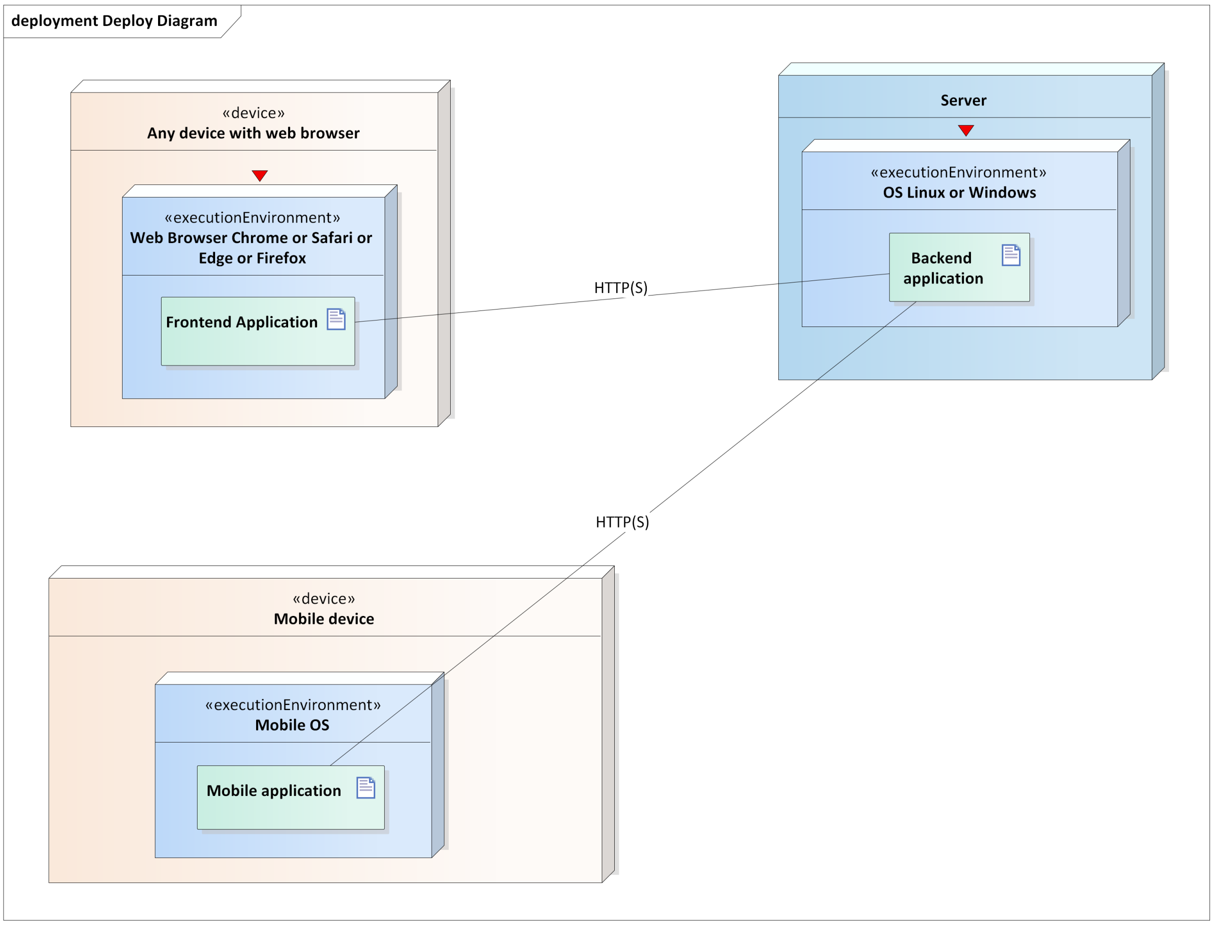


Figure 3.7: A deploy model

Chapter 4

Implementation

Quasar framework based on Vue.js was chosen to implement the front-end application. It allowed us to create a prototype application in a small amount of time. In this chapter, the implementation process of the application will be described. We will see how the individual layers were created and into what components applications were separated. It will be also shown how some useful and essential features for a better user experience were created.

This chapter is going to describe how the front-end client was fully developed. We will begin by listing the prerequisites. Then we will see how the application is being initialized with help of the tools from prerequisites. As our project was initialized, we constructed the project structure and set the configuration. Then we will take a closer look at the individual components and modules of the client and the means they were built with.

4.1 Prerequisites

We have chosen the Quasar framework, so we have to be in accordance with its prerequisites. Firstly, Node 12+ was required, so the Yarn package manager could be used which is recommended by Quasar developers. Then @quasar/cli package was installed with Yarn, so the project could be initialized with it [31].

The Yarn was also chosen because it is generally better than npm in a sense of performance and dependency locking [52].

4.2 Quasar Project Initialization

Once we have the Yarn and the Quasar CLI installed, we can move forward to project initialization.

```
yarn create quasar
```

Listing 4.1: Command to start project initialization

After that, the Quasar CLI option should be chosen and the project directory name should be entered. Then a Quasar version should be chosen. The Quasar 2 is going to be used as it supports the most modern version of Vue.js.

Then the particular language (JavaScript or TypeScript) should be chosen. We are going to choose plain JavaScript as TypeScript is suitable for bigger teams and can be an overkill for a small one.

Then we should choose between Webpack and Vite. Our choice is going to be Webpack as I had more experience with it.

Then we should choose the styling language. An SCSS was chosen as it is more readable and therefore more maintainable.

During the next step, other libraries should be chosen. Libraries are available for such purposes: HTTP client, internationalization, state management, and code linting. Axios, I18n, Vuex, and ESLint were chosen.

Also, we are being asked for an ESLint configuration. I personally prefer the Airbnb configuration as it is the strictest and thanks to it we always get organized and homogeneous code[29].

Then we are getting our application finally initialized.

4.3 Project Structure

In this section, the project file structure is going to be described by the directory tree with notes in Figure 4.1.

```
src
├── api
├── assets ........ Different graphic resources, e.g. images
├── axios ........ Axios configuration files
│   └── index.js .. File containing initialization configuration of Axios HTTP
│       Client
├── boot. . . Directory containing boot configuration files, i.e. what to preload
│   before application start
├── common ........ Globally used resources and constant values
│   ├── enums
│   └── constants.js
├── components
│   ├── global ........ Globally used components like modal windows
│   ├── layouts ........ Components used for building layouts
│   └── pages ........ Components used for building certain pages
├── css ........ Directory for global styling configuration files
├── i18n ........ Directory for language configuration files
├── layouts ........ Directory for layout components
├── mixins ........ Directory for mixins
│   ├── components .... Directory for mixins related to individual components
│   ├── decorators ...... Directory for mixins created with decorator pattern
│   ├── resources . . . Directory for mixins containing some read-only resources
│   └── store ........ Directory for mixins containing logic for each store
├── pages ...... Directory for components that represent individual pages (or
│   ”views”)
├── router ........ Directory for application router configuration files
├── store ........ Directory for store modules
│   └── addNew.js ........ Script for generating new store module attributes
├── utils ........ Directory for utilities used across all the application
└── App.vue ........ Vue.js root component
.env ........ Vue.js root component
generateStore.js ........ Script for generating new store modules
quasar.conf.js ........ Quasar core config file
```

Figure 4.1: Project file structure

4.4 Project Configuration

Our project is going to be configured with .env. This is a configuration file for defining environment variables. On the front-end it can be used to store private API keys or application back-end URLs, so every developer can set their own on their environment

[18].

Our config contains only one line:

```
VUE_APP_API_BASE_URL=http://localhost:44359/api
```

Listing 4.2: Project .env configuration

This config could be extended to store some private authentication keys, e.g. for reCaptcha.

4.5 View Components

Vue.js view layer is represented by Vue SFC files, where components are stored, which is a natural extension of HTML, CSS, and JavaScript. Their main advantages are:

- Components are modularized and separated.
- CSS styling is component scoped, so it gives more code separation.
- Optimization during compile-time.
- IDE support for autocompletion, so it decreases development time and the probability of writing typos.

The essential functions of SFC Options API that are used in a project are: data, props, computed, methods, watch, mixins and different component life cycle stage handlers [8].

Data option is used to store the component state. If something is changed in the state, it will reactively reflect in the view. Listing 4.3 demonstrates us how the option is used here. We are storing the values of input fields in an AddAccountDialog component.

```
export default {
  name: 'AddAccountDialog',
  data() {
    return {
      account: {
        currency: 0,
        name: '',
        amount: 0.0,
      },
      selectedCurrency: this.$t('account.selectCurrency'),
    };
  },
  ...
}
```

Listing 4.3: Data option example

Props option (properties) is used to pass read-only arguments to child components. For example, they can be used if we want to pass some function from parent component to be executed after a button click in the child component, so we can have customly defined callbacks that can lead to reusage of components. Or if we want to pass a data to be displayed in a table component. Listing 4.4 demonstrates us how the option is used here. We are passing some properties like transactions to display them in the table and we pass functions like onEdit to execute them during after some action, for example after editing a record in the table. IsLoading property is used to set table's loading from a parent component.

```
 props: {
   transactions: {
     type: Array,
     default: () => [],
   },
   pagination: {
     type: Object,
     default: () => {},
   },
   onDelete: {
     type: Function,
     default: () => null,
   },
   onEdit: {
     type: Function,
     default: () => null,
   },
   onRequest: {
     type: Function,
     default: () => null,
   },
   isLoading: {
     type: Boolean,
     default: false,
   },
   transactionType: String,
   parentId: Number,
 },
```

Listing 4.4: Props option example

Computed option (computed properties) is used to reactively compute component properties if some of the dependencies were changed. They are not set explicitly, the computed property dependencies are those variables that affect the result of the handler that is assigned to the computed property. They can be used as mappers or aggregators for more complex objects that are made from smaller dependencies. Listing 4.5 demonstrates to us how the option is used in CloseDebtDialog.vue component. Here the computed option is used as a mapper that converts account entities from the store to options in the selection dropdown element.

```
computed: {
    accounts() {
      return this.accountsGetter.map((account) => ({
        label: account.name,
        value: account.id,
      }));
    },
},
```

Listing 4.5: Computed option example

Watch option stands for a set of watchers – handlers – that are being executed on watched dependency change. Those are used for tracking some changes and applying changes to the component state. Changing the component state in the computed handlers is considered to be a bad practice and is generally forbidden. Listing 4.6 demonstrates us how the option is used in a GoalTable.vue component. In this case the option is used to update the variable in the data option. It is used to make a pagination object, passed from the parent component as an immutable value, mutable, because we pass it then as a v-model to the q-table component and v-model is required to be mutable.

```
  props: {
    ...
    pagination: {
      type: Object,
      default: () => {},
    },
    ...
  },
  data() {
    return {
      initialPagination: {
        ...this.pagination,
      },
      selectedRow: {},
    };
  },
  watch: {
    pagination: {
      handler(newValue) {
        this.setPagination(newValue);
      },
      deep: true,
    },
  },
```

Listing 4.6: Watch option example

Methods option is used to separate SFC logic into several functions for structurization and readability. Listing 4.7 demonstrates to us how the option is used in the Account.vue component that is responsible for showing a list of accounts to users. Here we can see an example of how methods prevent code duplication and make a code more readable as functions have meaningful and short names [27].

```
<template>
  <q-page class="q-px-md">
    <q-btn @click="handleAdd" class="q-ml-md">{{ $t("account.add") }}</q-btn>
    <div class="row">
      <AccountCard
        v-for="account in accountsGetter"
        :key="account.id"
        :account="account"
        @add:transaction="refreshPage"
        @edit="refreshPage"
        @delete="refreshPage"
      />
    </div>
  </q-page>
</template>

<script>
...
export default {
  ...
  methods: {
    async handleAdd() {
      this.$q.dialog({
        component: AddAccountDialog,
      }).onOk(this.refreshPage);
    },
    async refreshPage() {
      await this.withLocalLoadingAndErrorDialog(this.fetchAccountsAction);
    },
  },
  async mounted() {
    await this.refreshPage();
  },
```

Listing 4.7: Methods option example

Mixins option is used to inject a set of options (methods, data, watchers, etc.) into other components to reduce duplication and increase structurization. More information and examples can be found in the section 4.11.

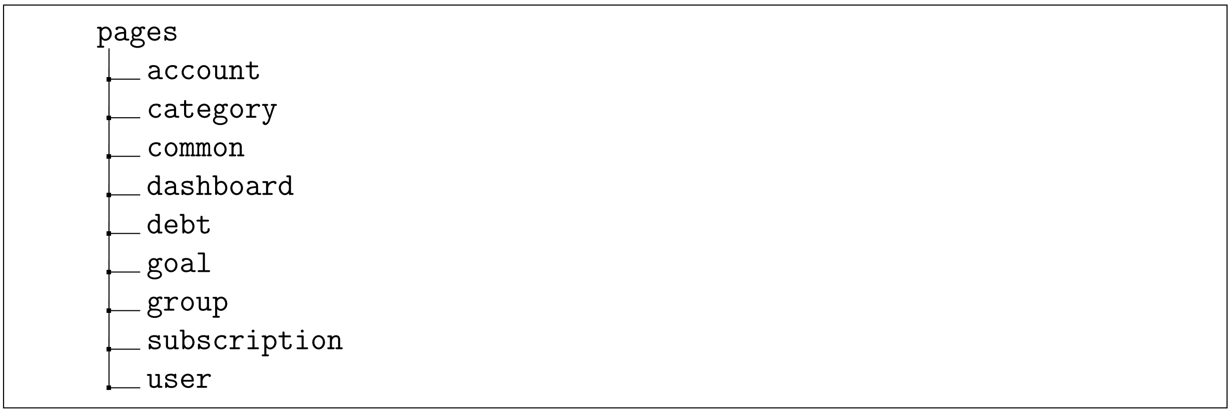
Life cycle hooks are used to correctly render a component and to remove possible component runtime dependencies. For example, beforeMount hook is used to prefetch data before the component is loaded, so it is displayed correctly. Or beforeDestroy hook can be used to close WebSocket connections to prevent memory leaks. The previous listing 4.7 demonstrates us how the mounted handler is used to fetch data after mounting.

All the components are stored in three directories: components, layouts, and pages.

The first directory, components, contains subcomponents for the page or layout components (layouts and page subdirectories respectively) or some global components like modal windows or component loading stubs.

The second directory, layouts, contains SFC files for individual layouts. Layouts are used to manage the application windows and to wrap page components with navigation elements. They are not mandatory, however, they help to structurize the application[24].

The third directory, pages, contains SFC files for individual pages. Pages are used to represent big sections of UI that allows working with particular application modules. For example, we have two pages, related to the Account entity: Account.vue and Accounts.vue. The first one is used to perform use cases, related mainly to a single account instance. The second one is used to perform use cases, related to a list of user accounts. This directory is also separated into several directories that are responsible for the particular business domain of the front-end client. Figure 4.2 demonstrates the structure.



```
pages
 ├── account
 ├── category
 ├── common
 ├── dashboard
 ├── debt
 ├── goal
 ├── group
 ├── subscription
 └── user
```

Figure 4.2: Page subcomponent directory structure

4.6 Styling

SCSS is the kind of syntax of SASS language. SASS is a preprocessor of the CSS language. Its main aim is to reduce duplicates in code, so developers can spend less time and make fewer mistakes by using nesting or creating variables. It is used for UI styling in our project[32]. Listings 4.8 and demonstrate two essential features of SCSS that are creating variables and nesting elements to reduce possible duplicates and make source code more readable.

```scss
// source .scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;
$secondary-color: #666;

body {
  font: 100% $font-stack;
  color: $primary-color;
}

nav {
  li { display: inline-block; color: $secondary-color; }
  p {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
    color: $secondary-color;
  }
}

// compiled .css
body {
   font: 100% Helvetica, sans-serif;
   color: #333;
}
 nav li {
   display: inline-block;
   color: #666;
}
 nav p {
   display: block;
   padding: 6px 12px;
   text-decoration: none;
   color: #666;
}
```

Listing 4.8: SCSS to CSS comparison

All the styles are stored mainly in two places: in the ”style” tag in SFC files and in two files in the CSS directory: app.scss and quasar.variables.css. In app.scss, we can add global styles like ”.text-underline”, which allows adding an underline to a text, or other globally usable styles. Quasar.variables.css is used for configuring color settings. Quasar has a set of already predefined colors, though you can change them for your own color palette. [12]

As for the styles, stored in SFCs, they are related only to the component they are written in if the scoped flag is set for ”style” tag, otherwise, styling will affect child components as well. [34]

4.7 Internationalization

Vue I18n plugin was used to create an application internationalization. Its configuration files are actually objects mapping a phrase identifier and a concrete phrase. Listing 4.9 demonstrates to us such an object. That way different configurations can be created for a front-end application with the ability to switch them. In the future, it can help increase our target market audience [50].

```
// sample translation configuration
export default {
  failed: ’Action failed’,
  success: ’Action was successful’,
  confirmRegistration: ’Registration was confirmed, you can use your credentials to sign in’,
  account: {
    add: ’Add’,
    selectAccount: ’Select an account’,
    selectCurrency: ’Select a currency’,
    income: ’Income’,
    name: ’Account name’,
    amount: ’Account amount’,
    goToAccount: ’Go to account’,
  },
};

// usage in SFC
data() {
    return {
      selectedCurrency: this.$t(’account.selectCurrency’), // will result in selectedCurrency === ’Select a currency’ for the english language
    };
},
```

Listing 4.9: I18n configuration and usage example

4.8 Vue Router

Vue.js Router is used to manage views. DOM is being dynamically rerendered depending on the path of the page user is on. It is used for navigation in an application. All the router actions should be performed on a view layer, because it does not manage the state of the application, but manages only what the user sees.

Router is being configured with a list of paths as can be seen in listing 4.10.

```
export const ROUTES = {
  root: ’/’,
  signIn: ’/sign-in’,
  accounts: ’/accounts’,
  account: ’/accounts/:accountId’,
};

export const routes = [
  {
    path: ROUTES.root,
    component: () => import(’layouts/MainLayout.vue’),
    children: [
      { path: ’’, component: () => import(’pages/Index.vue’) },
    ],
  },
  {
    path: ROUTES.signIn,
    component: () => import('layouts/MainLayout'),
    children: [{ path: '', name: ROUTES.signIn, component: () => import('pages/Auth.vue') }],
  },
  {
    path: ROUTES.accounts,
    component: () => import('layouts/MainLayout'),
    children: [{ path: '', name: ROUTES.accounts, component: () => import('pages/Accounts.vue') }],
  },
  {
    path: ROUTES.account,
    component: () => import('layouts/MainLayout'),
    children: [{ path: '', name: ROUTES.account, component: () => import('pages/Account.vue') }],
  },

  // Always leave this as the last one,
  // but you can also remove it
  {
    path: '/:catchAll(.*)*',
    component: () => import('pages/Error404.vue'),
  },
];
```

Listing 4.10: VueRouter configuration example

Here we can see such a pattern demonstrated in listing 4.11:

```
component: () => import('layouts/MainLayout.vue'),
    children: [
      { path: '', component: () => import('pages/Index.vue') },
    ],
```

Listing 4.11: VueRouter layout pattern

It is used to wrap the inside component (Index.vue) into the layout (MainLayout.vue) that contains a header, menu, footer, and other common elements of an application.

The router can also be configured to fetch dynamic arguments from a path (/accounts/:accountId), so it can be used, for example, to retrieve an entity from the server with a corresponding identificator.

4.9 State Management

Vue.js state management layer is frequently represented by Vuex library providing a reactive store as it historically was the recommended library. Its core concepts are modules, state, getters, mutations, and actions. Each module can contain the next nested modules, state, getters, mutations, and actions.

A state is an object that contains necessary data and is related to some module. Listing 4.12 demonstrates us an example of such a state. Function getInitialState is used for setting a state of a store without giving pointer access to an initial state object or its children. The initial state can be sometimes used to adjust content rendering in component (when there is a getter in the component that points to an undefined value and it is used in a template, it can sometimes result in an error).

```
export const AccountStoreMeta = {
  getInitialState: () => ({
    account: {
      id: 0,
      code: ’string’,
      currency: 0,
      name: ’string’,
      userId: 0,
      amount: 0,
    },
    accounts: [
      {
        id: 1,
        code: ’CZK-1-TH’,
        currency: 0,
        name: ’Test CZK account’,
        userId: -1,
        amount: 1000,
      },
      {
        id: 2,
        code: ’USD-2-TH’,
        currency: 2,
        name: ’Test USD account’,
        userId: -1,
        amount: 100,
      },
      {
        id: 3,
        code: ’EUR-3-TH’,
        currency: 1,
        name: ’Test EUR account’,
        userId: -1,
        amount: 100,
      },
      {
        id: 4,
        code: ’EUR-4-TH’,
        currency: 1,
        name: ’Test EUR account’,
        userId: -1,
        amount: 100,
      },
      {
        id: 5,
        code: 'EUR-5-TH',
        currency: 1,
        name: 'Test EUR account',
        userId: -1,
        amount: 100,
      },
    ],
  }),
};


const state = {
  ...AccountStoreMeta.getInitialState(),
};
```

Listing 4.12: Store module state example

Getters are reactive computed properties of the store that are used to acquire data from the store. Thanks to their reactivity, they help to update the application view. Listing 4.13 demonstrates us getters for account list and an account that are stored and displayed in the account list and account pages respectively.

```
import { AccountStoreGetters } from 'src/store/account/getters.meta';

export const getters = {
  [AccountStoreGetters.getAccount]: (state) => state.account,
  [AccountStoreGetters.getAccounts]: (state) => state.accounts,
};
```

Listing 4.13: Store module state example

Mutations are used to register changes in the store state and to apply the changes. They should be used only inside actions as they are required to be synchronous and some heavy synchronous mutation can lead to application freezing. To add, it's anyway better to stick to consistency and use only actions in the view layer. Listing 4.14 shows us the mutations for account entities that are basically setters for a list of accounts and for a single account.

```
import { AccountStoreMutations as AccountMutations } from 'src/store/account/mutations.meta';

export const mutations = {
  [AccountMutations.setAccount](state, account) {
    state.account = account;
  },
  [AccountMutations.setAccounts](state, accounts) {
    state.accounts = accounts;
  },
};
```

Listing 4.14: Store module state example

Actions are used to perform asynchronous actions, mostly requests to the back-end server, by applying changes to the application states, using mutations. Listing 4.15 shows us account actions. Firstly, some request is being sent to the server and then if the response has some meaningful data, we save it to the store with help of our mutations.

```
import { AccountStoreActions } from ’src/store/account/actions.meta’;
import { AccountStoreMutations } from ’src/store/account/mutations.meta’;
import { AccountService } from ’src/api/AccountService’;

export const actions = {
  async [AccountStoreActions.fetchAccounts](context) {
    const { data: accounts } = await AccountService.fetchMany();
    context.commit(AccountStoreMutations.setAccounts, accounts);
  },
  async [AccountStoreActions.fetchAccount](context, accountId) {
    const { data: account } = await AccountService.fetchOne(accountId);
    context.commit(AccountStoreMutations.setAccount, account);
  },
  async [AccountStoreActions.createAccount](context, { name, currency,
   amount }) {
    const { data: account } = await AccountService.createOne({ name,
   currency, amount });
    return account;
  },
  async [AccountStoreActions.deleteAccount](context, accountId) {
    await AccountService.deleteOne(accountId);
  },
  async [AccountStoreActions.updateAccount](context, account) {
    await AccountService.updateOne(account.id, account);
  },
};
```

Listing 4.15: Store module state example

Our module has its own boilerplate structure. Figure 4.3 directory tree with explanation notes can be seen below.

```
module
├── actions.js .... Actions implementation with primarily using API Service
│                   layer calls
├── actions.meta.js ............ Action names saved as object of constants
├── getters.js ................................... Getters implementation
├── getters.meta.js ............ Getter names saved as object of constants
├── index.js .... Main file that collects and initializes all module components
├── mutations.js ............................. Mutations implementation
├── mutations.meta.js ........ Mutation names saved as object of constants
├── store.js ....... File containing function that generates store initial state
└── utils.js ...................... Commonly used functions across module
```

Figure 4.3: Store module boilerplate directory structure

It is seen that .meta.js files are used to store action, getter, and mutation names. It is considered to be a pattern that helps use the benefits of Linter and IDE autocomplete as well as reduce the chance for a developer mistake when writing a wrong name of any of those actions.

4.10 API Client

Axios library was used as the API Client. In addition, @microsoft/signalr library was used to keep a WebSocket connection with the server to receive notifications.

4.10.1 Axios

The State Management layer in Vue.js is represented by the Axios. The Axios is used to send HTTP requests to the server. Also, it provides additional functionality for handling requests like an interception. API Service is required only for making and processing requests.

4.10.1.1 Boot

The Axios is required to be initialized before being used in an application. It is due to initializing headers if an authorization session was saved locally or for initializing API base URL. Listing 4.16 demonstrates an example boot file. The API base URL is loaded from the .env file.

```
import axios from 'axios';
import store from 'src/store';
import { AuthStoreActions } from 'src/store/auth/actions.meta';
import { AuthServiceMeta } from 'src/api/AuthService/service.meta';

```

```
export const baseURL = process.env.VUE_APP_API_BASE_URL;

const api = axios.create({
  baseURL,
});

api.interceptors.response.use(
    ...
);

export default api;
```

Listing 4.16: Axios boot file example

4.10.1.2 Interceptor

An interceptor can be used to additionally handle incoming requests. For example, it can be used to check if a user is correctly authorized:

```
api.interceptors.response.use(
  (response) => response,
  async (error) => {
    const e = error;
    const originalRequest = e.config;
    if (
      e.response && e.response.status === 401
      && originalRequest.url !== AuthServiceMeta.ROUTES.SIGN_OUT
    ) {
      await store.dispatch(AuthStoreActions.signOut);
    }
    return Promise.reject(error);
  },
);
```

Listing 4.17: Interceptor example

4.10.1.3 Service Object Boilerplate

Our typical API Service consists of these files:

1. **service.meta.js** – file containing an object with route names and other meta information required for requests. It is used to contain route values to prevent duplication.
2. **index.js** – file containing an implementation of service.

Here is an example of User API Service in the listing 4.18:

```
//service.meta.js
export const UsersServiceMeta = {
  Routes: {
    getCurrentUser: '/users/me',
  },
};

//index.js
import api from ’src/axios’;
import { UsersServiceMeta } from ’src/api/UsersService/service.meta’;

export const UsersService = {
  fetchCurrentUser() {
    return api.get(UsersServiceMeta.Routes.getCurrentUsers);
  },
};
```

Listing 4.18: User service boilerplate example

As most of the use case domains contained CRUD operations, it was decided to create a CRUD generic class, which methods could be overwritten in certain cases. Listing 4.19 demonstrates the code of the CRUD service. Functions are based on REST API concepts [19].

```
import api from ’src/axios’;

export const CrudService = {
  generateCrudRouteFunction: (baseUrl) => (id, patchAttribute) => ‘${
    baseUrl}${id || ’’}${(patchAttribute && ‘/${patchAttribute}‘) || ’’
    }‘,
  get Routes() {
    return { Crud: this.generateCrudRouteFunction(’’) };
  },
  fetchOne(id, params = {}) {
    return api.get(this.Routes.Crud(id), { params });
  },
  fetchMany(params = {}) {
    return api.get(this.Routes.Crud(), { params });
  },
  deleteOne(id, body, params = {}) {
    return api.delete(this.Routes.Crud(id), { params });
  },
  createOne(body, params = {}) {
    return api.post(this.Routes.Crud(), body, { params });
  },
  updateOne(id, body, params = {}) {
    return api.put(this.Routes.Crud(id), body, { params });
  },
  patchOne(id, attribute, body = null, params = {}) {
    return api.patch(this.Routes.Crud(id, attribute), body, { params });
  },
};
```

Listing 4.19: CRUD generic API service

Listing 4.20 demonstrates the code of the account service, inherited by CRUD with the means of JavaScript. AccountServiceMeta is used to set the base route of the API resource.

```javascript
//service.meta.js
export const AccountServiceMeta = {
  Routes: {
    Crud: CrudService.generateCrudRouteFunction('/accounts/'),
  },
};
//index.js
export const AccountService = {
  ...CrudService,
  ...AccountServiceMeta,
};
```

Listing 4.20: API service inhertied from CRUD example

4.10.2 SignalR Hub

SignalR Hub is used to asynchronously send messages to the server or to get them from it. It uses WebSocket by default, however, if it is not available, it chooses another most appropriate way to maintain communication. Listing 4.21 demonstrates us the boot file for the SignalR Hub.

```javascript
import { HubConnectionBuilder, LogLevel } from '@microsoft/signalr';
import LocalStorageApi, { localStorageFields } from 'src/utils/localStorage';

export default ({ app }) => {
  const buildHubConnection = (endpoint) => new HubConnectionBuilder()
    .withUrl(endpoint, {
      accessTokenFactory: () => `${LocalStorageApi.getRaw(localStorageFields.accessToken)}`,
    })
    .withAutomaticReconnect()
    .configureLogging(LogLevel.Information)
    .build();

  app.config.globalProperties.$notificationHub = buildHubConnection('hub/notification');
};
```

Listing 4.21: SignalR boot file example

In our application, it is used to receive notifications asynchronously if some transaction was added to a group where the user belongs as can be seen from listing 4.22.

```javascript
async initializeHub() {
  await this.$notificationHub.start();
  this.$notificationHub.on('SendNotification', async () => {
    await this.fetchNotificationsAction(this.notificationPagination);
    this.$q.notify({
      message: 'notifications.newNotification',
      caption: 'notifications.rightNow',
      color: 'positive',
    });
  });
},
```

Listing 4.22: Function used to initialize SignalR Hub in the root component App.vue

4.11 Mixins

Mixins are a powerful Vue.js Options API feature that allows for reducing code duplication a lot. All the options and properties, defined at a mixin, will be inherited by a component, where the mixin is used. If the component has options or properties sharing the same names, they will override all the mixins.

For example, our application uses mixins for placing all the store module-related actions, getters, and mutations into a single file that could be imported using the mixins option to use all the functionalities, so the developer won't have to write the same code again. An example of such a store mixin, implementing a category store, can be seen in the listing 4.23

```
import { mapActions, mapGetters } from 'vuex';
import { CategoryStoreActions } from 'src/store/category/actions.meta';
import { CategoryStoreGetters } from 'src/store/category/getters.meta';

export default {
  computed: {
    ...mapGetters({
      categoriesGetter: CategoryStoreGetters.getCategories,
      userCategoriesGetter: CategoryStoreGetters.getUserCategories,
      defaultCategoriesGetter: CategoryStoreGetters.getDefaultCategories,
    }),
  },
  methods: {
    ...mapActions({
      fetchCategoriesAction: CategoryStoreActions.fetchCategories,
      fetchUserCategoriesAction: CategoryStoreActions.fetchUserCategories
   ,
      fetchDefaultCategoriesAction: CategoryStoreActions.
   fetchDefaultCategories,
      deleteCategoryAction: CategoryStoreActions.deleteCategory,
      updateCategoryAction: CategoryStoreActions.updateCategory,
      createCategoryAction: CategoryStoreActions.createCategory,
    }),
  },
};
```

Listing 4.23: Category store mixin example

Also one of the notable use is using mixins for distributing loading decorators. For example, the mixin from the listing 4.24 is used to store functions that are responsible for setting loading states and showing error windows during the executing of other functions.

```
import errorDialogMixin from 'src/mixins/decorators/withErrorDialogMixin'
    ;
import loadingMixin from 'src/mixins/decorators/withLoadingMixin';

export default {
  mixins: [errorDialogMixin, loadingMixin],
  methods: {
    async withGlobalLoadingAndErrorDialog(tryProcedure, catchProcedure) {
      await this.withGlobalLoading(() => this.withErrorDialog(
     tryProcedure, catchProcedure));
    },
    async withLocalLoadingAndErrorDialog(tryProcedure, catchProcedure) {
      await this.withLocalLoading(() => this.withErrorDialog(tryProcedure
    , catchProcedure));
    },
    async withComponentLoadingAndErrorDialog(tryProcedure, component,
     catchProcedure) {
      await this.withComponentLoading(
        () => this.withErrorDialog(tryProcedure, catchProcedure),
        component,
      );
    },
  },
};
```

Listing 4.24: Decorator mixin example

4.11.1 Decorator Mixins

A concept of the decorator pattern is used and implemented with help of JavaScript means. Actually, just decorator functions are used in the application, so you can pass some functions and additional parameters and it is going to be executed, but with some modifications [14].

In our project, it is mainly used for calling functions while a loading needs to be set or while an error should be captured and shown during the function execution. This way the code duplication can be removed and all the other code can become much more readable.

Our decorator mixins can be considered as decorators for functions with no return value and accepting no arguments, related to function. Thanks to JavaScript means, we can always create such an arrow function [9] or create a separate SFC method and pass it as an argument.

4.12 Modal Windows

Modal windows are implemented, using the Quasar Dialog plugin, a separate wrapper component, and a mixin as a workaround for providing the required modal window API, so the component could be recognized as a Quasar Dialog [15].

Modal windows can be separated into three types: showing information, asking for confirmation, and having their own component inside. For the first two types, neither mixin nor component is required and its calling is as easy as it is shown in the listing 4.25.

```
this.$q.dialog({
    title: this.$t('global.confirm'),
    message: this.$t('transaction.areYouSureToDelete'),
    cancel: true,
    persistent: true,
}).onOk(this.deleteTransaction);
```

Listing 4.25: Creating a simple dialog example

Calling a modal window and passing props also will not take much time to write code and is seen in the listing 4.26.

```
this.$q.dialog({
    component: EditDebtDialog,
    componentProps: {
      debt,
    },
}).onOk(this.onEdit);
```

Listing 4.26: Creating a component dialog example

The decorator function withErrorDialog is used to open a modal window with an error that occurred during the execution of a function that was passed to the decorator.

4.13 Component Loading

Component loading is an essential feature as one of the UX best practices is that users should always know about the state of the application, so the loading state indicates that the user is doing everything fine, but they should wait for something [23].

In our application, it is implemented with help of a loading state mixin that contains a data section for loading indication with decorator methods for loading, and a Quasar UI loading component that is used to show loading plug or animation to a user.

Showing loading in the view could be triggered with functions withGlobalLoading, withLocalLoading, and withComponentLoading.

Function withGlobalLoading is used to set the global loading of the application in the store during action execution, so the spinner is shown on all the application pages. Listing 4.27

demonstrates us the usage of this method to set all the application to loading state until it loads the most essential data and make necessary connections.

```
methods: {
  async initializeApplication() {
   await this.loadSession();
   await this.initializeHub();
   if (this.user?.id > 0) {
    await this.fetchNotificationsAction(this.notificationPagination);
   }
  },
},
async mounted() {
  await this.withGlobalLoadingAndErrorDialog(this.initializeApplication
 );
},
```

Listing 4.27: Making a request with page component example with store and loading mixins in pages/account/Accounts.vue

Function withLocalLoading is used to set the loading state of a component where the function is called during action execution, so the spinner is shown only on a component's view. Listing 4.28 demonstrates the usage of the loading mixin for fetching the account list and showing the loading state of the component performing the fetch action to the user. It is also a great example of using mixins (described in section 4.11) to reduce the total amount of code.

```
mixins: [accountStoreMixin, withLoadingAndErrorDialogMixin],
methods: {
  async refreshPage() {
   await this.withLocalLoadingAndErrorDialog(
    this.fetchAccountsAction
   );
  },
},
```

Listing 4.28: Making a request with page component example with store and loading mixins in pages/account/Accounts.vue

Function withComponentLoading is used to set the loading state of a component that was referenced by the refs feature, for example, to set the loading state of the child component. Listing 4.29 demonstrates to us how we are setting a child component to loading while fetching data to pass it then to this exact component as a props option.

```
async refreshAccount() {
 await this.withComponentLoading(
  this.fetchAccount,
  this.$refs.accountCard,
 );
},
```

Listing 4.29: Example of setting a child component to loading state

4.14 Boot

Boot files in Quasar Framework are used to inject dependencies into the root Vue app object before the object was instantiated. It can be also used to structurize dependencies. Boot files are being enabled and disabled in quasar.conf.js. Listing 4.30 demonstrates the i18n boot file. It is being injected into the Vue app object as a plugin.

```
import { boot } from 'quasar/wrappers';
import { createI18n } from 'vue-i18n';
import messages from 'src/i18n';

const i18n = createI18n({
  locale: 'en-US',
  messages,
  fallbackWarn: false,
  missingWarn: false,
});

export default boot(({ app }) => {
  // Set i18n instance on app
  app.use(i18n);
});

export { i18n };
```

Listing 4.30: I18n boot file

4.15 Logger

A logger with different log levels is used to debug the application and provide meaningful information. The minimum log level can be set with LogLevelEnum. It is implemented with a boot dependency that injects into the root Vue app object. Default console logging is wrapped and used under the hood.

4.16 Enums

Enums are used to store possible constant values or options of some types. They are implemented with help of Object.freeze which creates an immutable structure [28]. They are basically used for switching between transaction types when dealing with transaction components or for distinguishing log levels. Listing 4.31 demonstrates such a pattern.

```
export const LogLevelEnum = Object.freeze({
  Debug: 1,
  Trace: 2,
  Info: 3,
  Warning: 4,
  Error: 5,
});
```

Listing 4.31: Log level enumeration

4.17 Security

Application authentication is implemented with JWT tokens. The front-end client stores the JWT in the local storage to save sessions if a user closes a page with an application. All the main top-level logic can be seen in the listing 4.32. The token is being set on a successful signing in and being unset on a logout. Then it is being loaded from the local storage and set in the authorization header of the Axios instance and current user data is being fetched. If there is no token set, the session is destroyed.

```
export const actions = {
  [AuthStoreActions.loadSessionFromLocalStorage]: async (context) => {
    const accessToken = LocalStorageApi.getRaw(localStorageFields.accessToken);
    if (accessToken) {
      context.commit(AuthStoreMutations.setAccessToken, accessToken);
      api.defaults.headers.common.Authorization = ‘Bearer ${accessToken}‘;
      await context.dispatch(UserStoreActions.fetchCurrentUser);
      return;
    }

    context.commit(AuthStoreMutations.setAccessToken);
    await context.dispatch(
      UserStoreActions.setCurrentUser,
      { ...UserStoreMeta.getInitialState().user },
    );
  },

  [AuthStoreActions.signIn]: async (context, { email, password, rememberMe }) => {
    const { data } = await AuthService.signIn({ email, password, rememberMe });
    setAccessToken(data.accessToken);
    await context.dispatch(AuthStoreActions.loadSessionFromLocalStorage);
  },

  [AuthStoreActions.signOut]: async (context) => {
    unsetAccessToken();
    await context.dispatch(AuthStoreActions.loadSessionFromLocalStorage);
    await AuthService.signOut();
  },

  [AuthStoreActions.signUp]: async (context, { email, password, name }) => {
    await AuthService.register({ email, password, name });
  },
};
```

Listing 4.32: Auth store

Chapter 5

Testing

In this chapter, we are going to cover the testing of our front-end application. It will include describing front-end functional and end-to-end tests and usability tests for UX. Testing UI is necessary to speed up the development pace in the future, maintain the client, and find currently existing bugs [47].

5.1 Functionality Tests

Nightwatch was used for testing our front-end application using the end-to-end methodology. A headless web driver was used to simulate user's web browser and his interaction with the application. All the communication between the front-end client, back-end server, and database are going to be tested this way. Tests were created for each of the use-cases, including tests with wrong data to test front-end validation.

One of the reasons for such a choice was the fact that Nightwatch tests are also written in JavaScript, so it is easier to maintain a project written in one language and it is easier to find a person who will maintain it.

A set of tests was created to cover all the use-cases. Thanks to it, several bugs were detected on the front-end client and the back-end server. An example of such a test is shown in the listing 5.1. Identificators for various elements that do not have duplicates were created to make a test writing process easier, so we don't have to pick the selectors manually if UI changes. A random name is used to check if a category was successfully created and its name appeared in the table.

```
describe('Sign in and create a category successfully', () => {
  const randomName = (Math.random() + 1).toString(36).substring(2);

  it('test', (browser) => {
    browser
      .url('http://localhost:8080/sign-in')
      .setValue('input[name=email]', 'snusmumrmail@gmail.com')
      .setValue('input[name=password]', '123')
      .click('button[type=submit]')
      .url('http://localhost:8080/categories')
      .click('#add-category-button)
      .setValue(
        'input[name=category-name]',
        randomName,
      )
      .click('#dialog-base-modal-ok')
      .assert.containsText('#category-table', randomName)
      .end();
  });
});
```

Listing 5.1: Nightwatch test example

5.2 Usability Testing

Usability testing is made to understand if your application is user-friendly. Three usability tests were conducted with the different scenarios for the different application domains, so users won't get tired of testing. The testers were primarily young (20-27 years) and with some advanced technical knowledge. Such testers were chosen because we target the people with age 18-40 because the older ones tend not to use online applications because of a potential risk of a data breach [30].

The first test conducted had the following scenario:

1. Register a user profile.
2. Using the registered profile, log into the application.
3. Create an account that already has 100 USD with any name you prefer.
4. Increase an account's money amount by creating a transaction up to 1000 USD.
5. Then create a transaction that lowers the account's money amount down to 500 USD.
6. Create a group with the name Berlin and the EUR currency.
7. Create a transaction that increases the group money amount by 100 EUR.
8. Create a transaction that decreases the group by 25 EUR.
9. Log out of the system.

This test was conducted without any serious usability problems. One problem was that the logout button was put in an unfamiliar place. Also, it was notable that steps 7-9 were performed much faster than 3-5. Looks like our design may allow users to learn using the application at a higher pace, which is a good result [23].

The second test conducted had the following scenario:

1. Using the registered profile, log into the application.
2. Create a debt record for a beer that you have borrowed from your friend. Keep in mind, that you have to return the debt by tomorrow.
3. Create a debt record for a shot of vodka that you have bought for your friend. Keep in mind, that he has to return the debt by tomorrow as well.
4. Now make all the debts closed with the only existing account and check the money amount for correctness.
5. Create a goal with any name and the total sum of 100 USD.
6. Add 25 USD to the goal.
7. Add 74 USD to the goal and try to close the goal.
8. Add 1 USD to the goal and try to close the goal.
9. Log out of the system.

The second testing has revealed a bigger number of problems. For example, it was not intuitive for a person, how to create such a debt that you owe to someone or someone owes to you. It was not clear, how to close the goal. Then it was revealed that we do not have a proper error window if the "close goal" action was dispatched. Clicking the button with no explicit error shown made the tester confused.

The third test conducted had the following scenario:

1. Using the registered profile, log into the application.
2. Create a category record named "Cookies".
3. Create a transaction record where you have bought some cookies for 100 USD.
4. Create a record for your most important subscription if you have one.
5. Log out of the system.

During the third testing, the tester encountered the same logout button problem as in the first test. The tester also was confused that he was unable to choose the period of the subscription as he had a yearly subscription for the Ultimate Guitar application.

To conclude, our user interface has to be improved and provide more information to users, because some problems were still revealed, despite sticking to some methodology.

5.3 Conclusion

Functionality testing helped us find several bugs on both ends. In general, it also contributes to speeding up the development.

Also, usability testing was conducted that helped us find problems that could be potentially encountered by some users.

Chapter 6

Conclusion

The main aim was to create a user-friendly finance management application that allows people to control their money flow in a sufficient amount of time.

Firstly, we have conducted research on existing applications to see what features are the most essential and most demanded by users. During the research, some ideas for our UI were taken such as a button for quick transactions from the GeorgeGo application or separating business domains in the application into several pages, so the user won't get confused as it was with the CoinKeeper application.

Then the existing popular front-end technologies were researched, so we can rely on chosen technologies. Vue.js was considered to be the most suitable framework for our project. To narrow down, the Quasar UI framework was chosen for the much faster prototyping of the application. To add, we can test our application on the different target audiences, thanks to the easy Cordova and Electron integrations.

Taking everything into account, functional requirements were derived, then the use cases were derived, then it was possible to create an analytic domain model and non-functional requirements, so we can structurize our project and always have an overview of what is going on in the project.

Then the common architecture of the front-end client was designed, so the client is easily scalable and maintainable and all the essential client responsibilities are separated into individual layers. Then, based on the use cases, the UX low fidelity design was created to implement all the cases and to make the front-end development easier.

After the design was created, the front-end client development was started. During the development, various manuals for Vue.js, Quasar, Vuex store, i18n were used to adopt the best practices and learn about technology API. Different specific development methods were used like using decorator functions to reduce the amount of code and to improve readability or creating objects for store methods name to reduce the probability of developer's mistakes. The main focus was on making the code readable, maintainable and scalable with a minimal amount of duplicates.

During the implementation, some tests were created to check the system functionality, so developers can make more confident changes and consequently code at a higher pace. After the implementation, usability tests were conducted to check for user-friendliness. Results collected from the testing will be used in future improvements.

To conclude, all the aims were fulfilled: a minimal value product meeting functional requirements with a user-friendly interface and was developed in a sufficient amount of time.

6.1 Future Improvements

A front-end client prototype was developed with Quasar Framework, but it can be rewritten in other technologies on demand. Quasar is a good choice for the start, but it may lack some functions or vice versa it can have too many out-of-the-box features that can slow down our application without proper optimization.

For historical reasons, as the development started before Dec 2021, the Vuex store was used, but it is going to be refactored for Pinia. It will reduce the big amount of code to maintain and will speed up development and can even fix some bugs if such exist.

If we want to scale up in a sense of targeted devices, the mobile applications could be written firstly in Cordova as MVP and then into React native to use more of the native OS functions.

UI has to be improved, so the application is user-friendly to a bigger amount of users.

Unfortunately, due to management problems it was not possible to implement a statistics page in time, so it was left as the future improvement.

A basis for internationalization feature was created, but adding other language packs is still required.

Appendix A

UI Design Wireframes

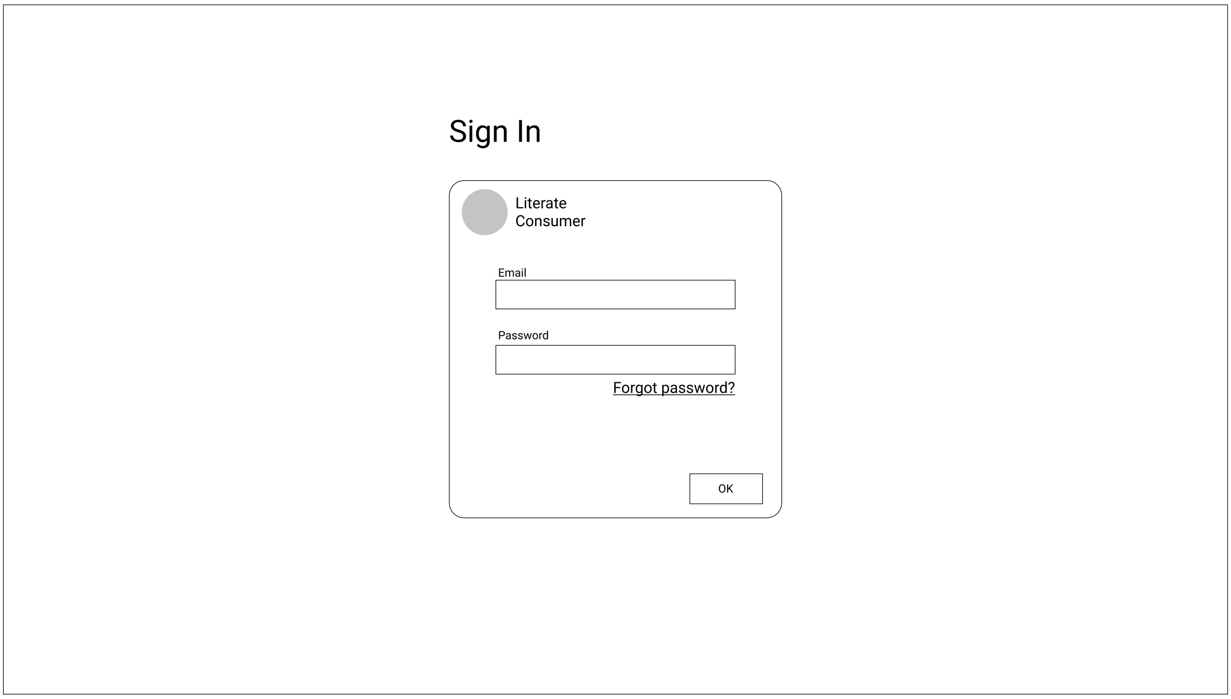


Figure A.1: Sign in page

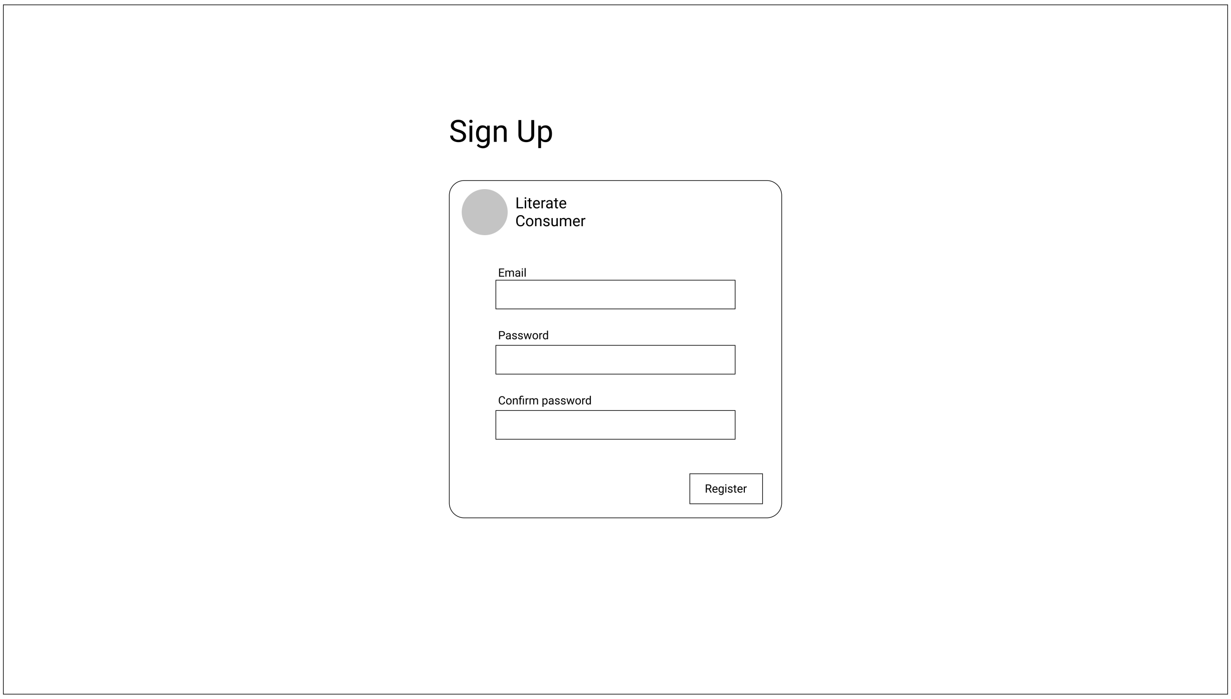
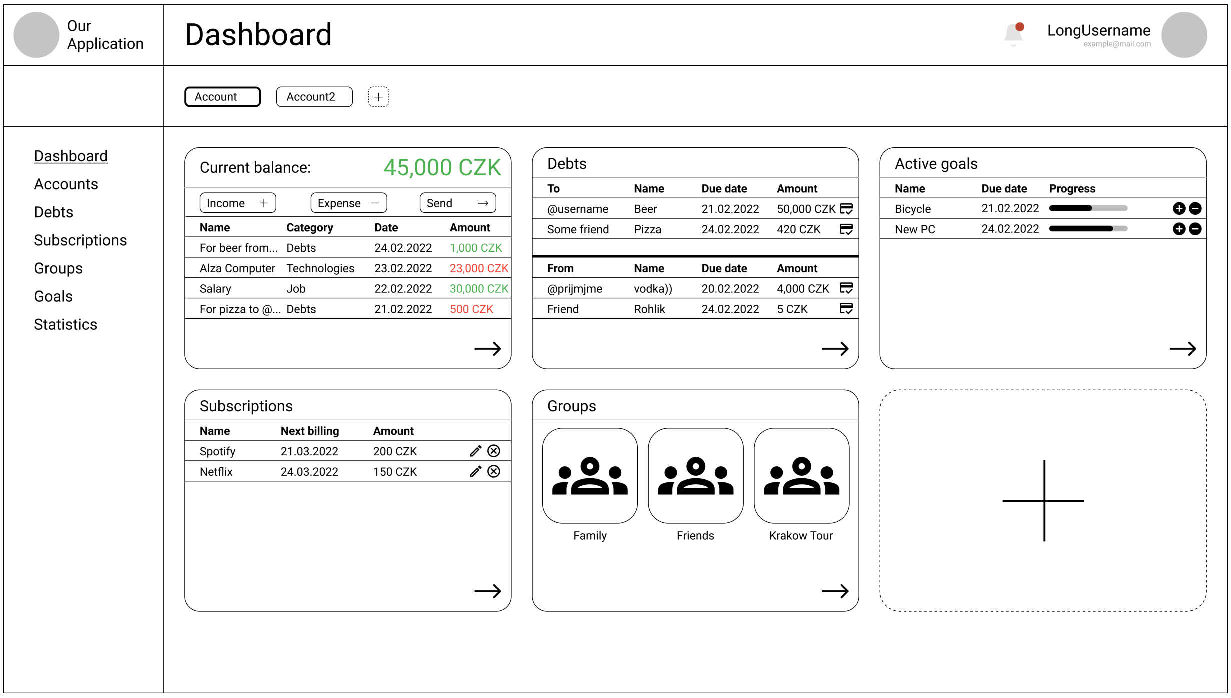


Figure A.2: Sign up page



Current balance: 45,000 CZK

Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1,000 CZK
Alza Computer	Technologies	23.02.2022	23,000 CZK
Salary	Job	22.02.2022	30,000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

Debts

To	Name	Due date	Amount
@username	Beer	21.02.2022	50,000 CZK
Some friend	Pizza	24.02.2022	420 CZK

From	Name	Due date	Amount
@prijmjme	vodka))	20.02.2022	4,000 CZK
Friend	Rohlik	24.02.2022	5 CZK

Active goals

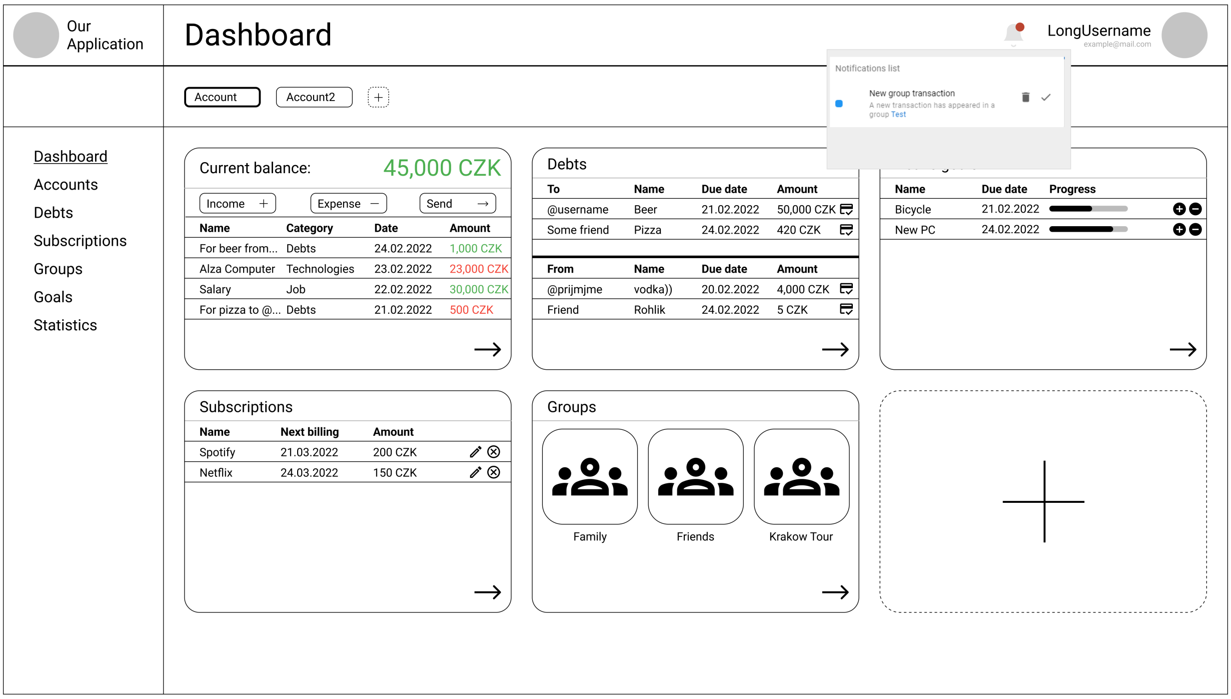
Name	Due date	Progress
Bicycle	21.02.2022	
New PC	24.02.2022	

Subscriptions

Name	Next billing	Amount
Spotify	21.03.2022	200 CZK
Netflix	24.03.2022	150 CZK

Groups

Figure A.3: Dashboard page



Notifications list

New group transaction
A new transaction has appeared in a group Test

Current balance: 45,000 CZK

Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1,000 CZK
Alza Computer	Technologies	23.02.2022	23,000 CZK
Salary	Job	22.02.2022	30,000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

Debts

To	Name	Due date	Amount
@username	Beer	21.02.2022	50,000 CZK
Some friend	Pizza	24.02.2022	420 CZK

From	Name	Due date	Amount
@prijmjme	vodka))	20.02.2022	4,000 CZK
Friend	Rohlik	24.02.2022	5 CZK

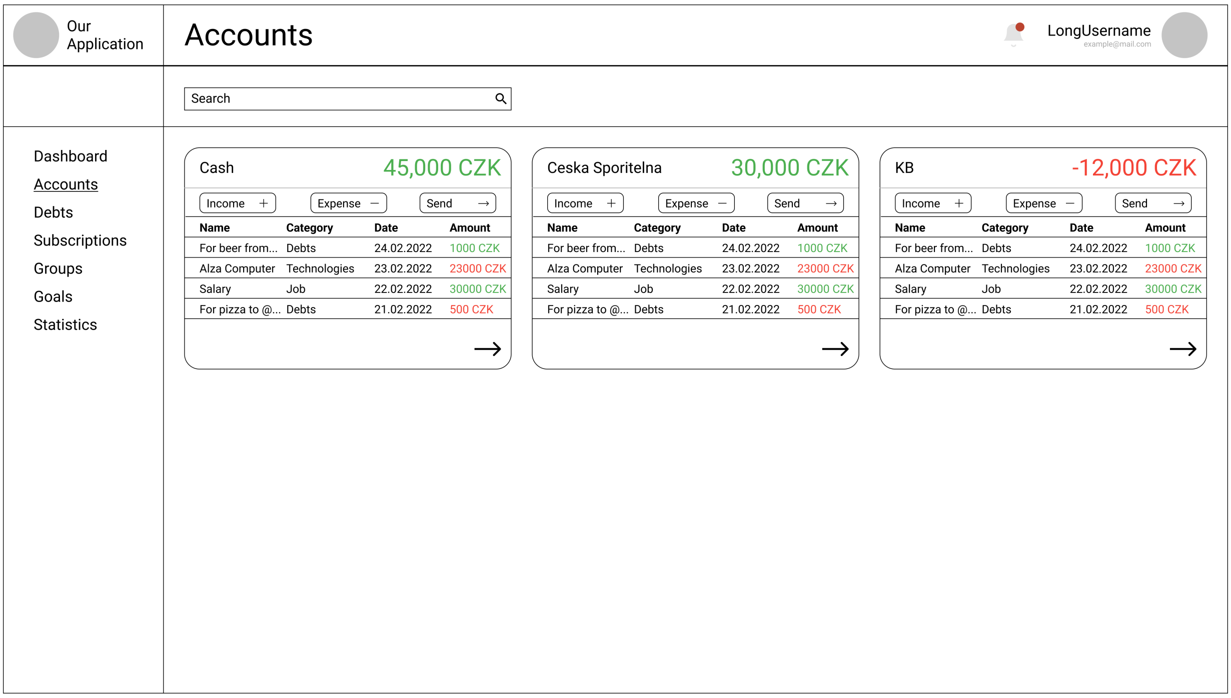
Name	Due date	Progress
Bicycle	21.02.2022	
New PC	24.02.2022	

Subscriptions

Name	Next billing	Amount
Spotify	21.03.2022	200 CZK
Netflix	24.03.2022	150 CZK

Groups

Figure A.4: Notifications page



Cash 45,000 CZK

Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

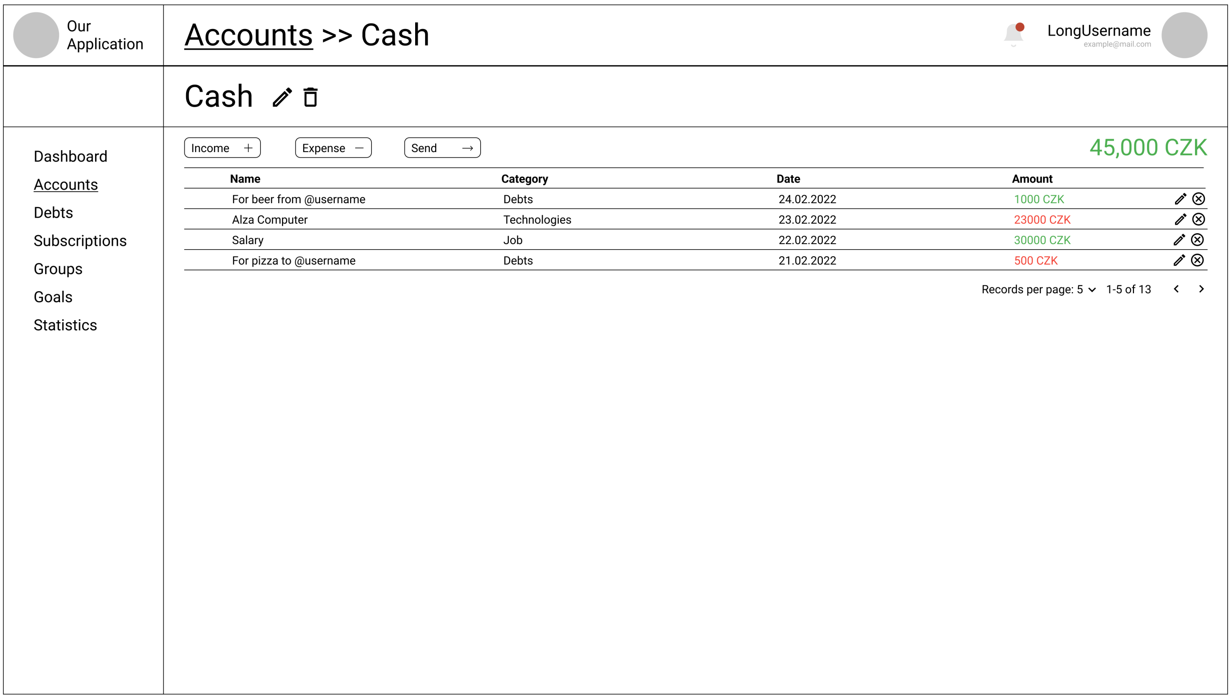
Ceska Sporitelna 30,000 CZK

Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

KB -12,000 CZK

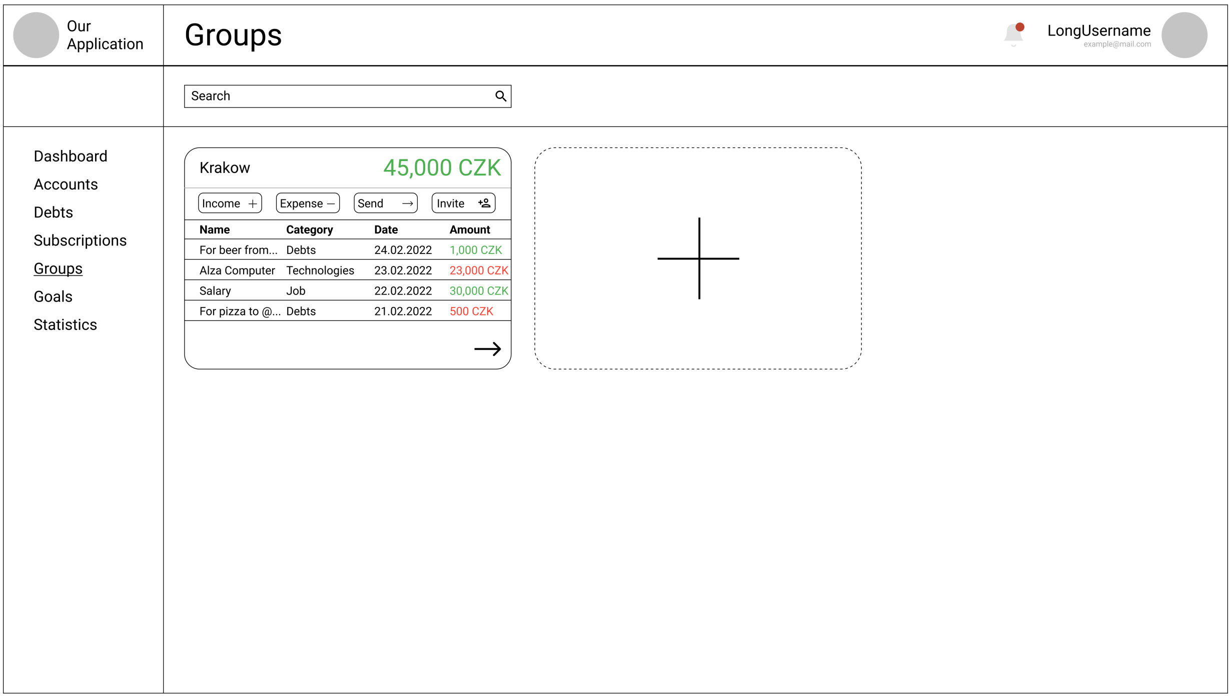
Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

Figure A.5: Accounts page



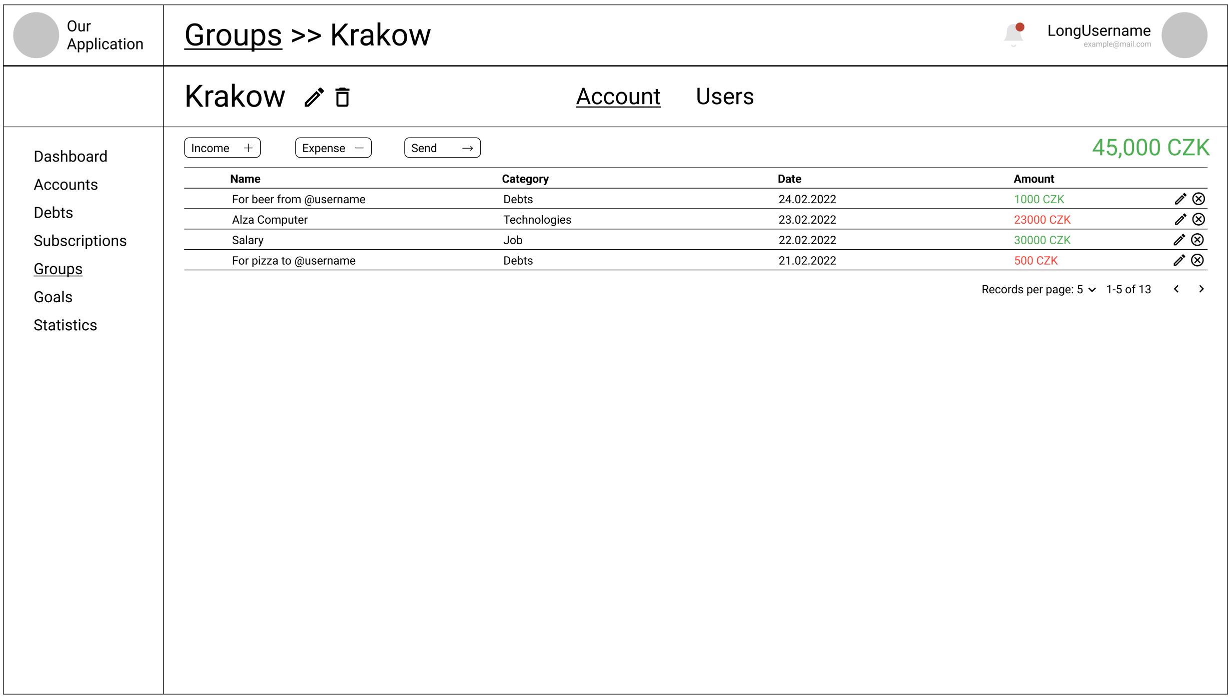
Name	Category	Date	Amount
For beer from @username	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @username	Debts	21.02.2022	500 CZK

Figure A.6: Account page



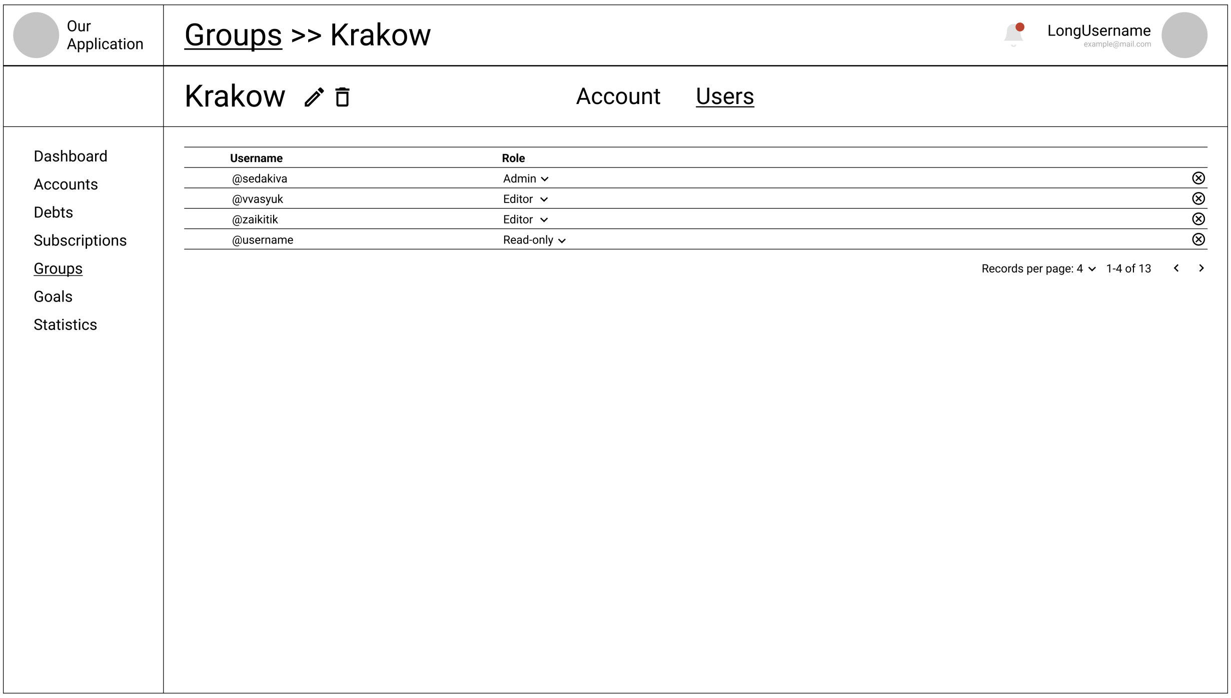
Name	Category	Date	Amount
For beer from...	Debts	24.02.2022	1,000 CZK
Alza Computer	Technologies	23.02.2022	23,000 CZK
Salary	Job	22.02.2022	30,000 CZK
For pizza to @...	Debts	21.02.2022	500 CZK

Figure A.7: Group list page



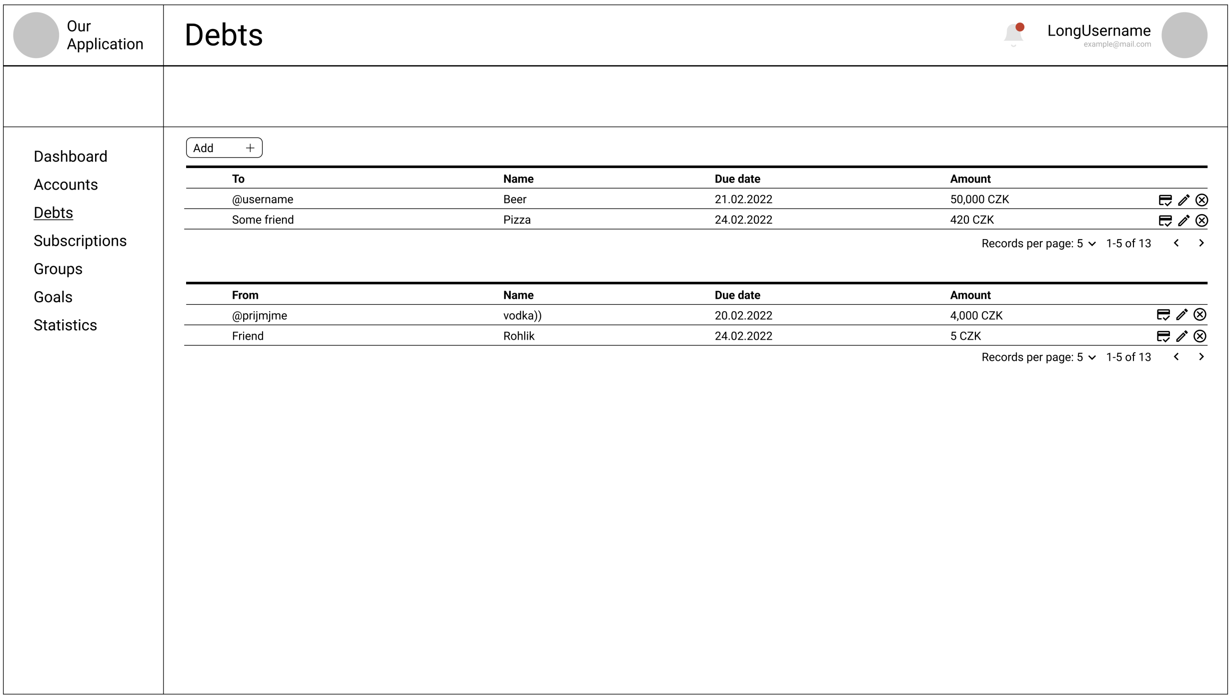
Name	Category	Date	Amount
For beer from @username	Debts	24.02.2022	1000 CZK
Alza Computer	Technologies	23.02.2022	23000 CZK
Salary	Job	22.02.2022	30000 CZK
For pizza to @username	Debts	21.02.2022	500 CZK

Figure A.8: Group account page



Username	Role
@sedakiva	Admin
@vvasyuk	Editor
@zaikitik	Editor
@username	Read-only

Figure A.9: Group user list page



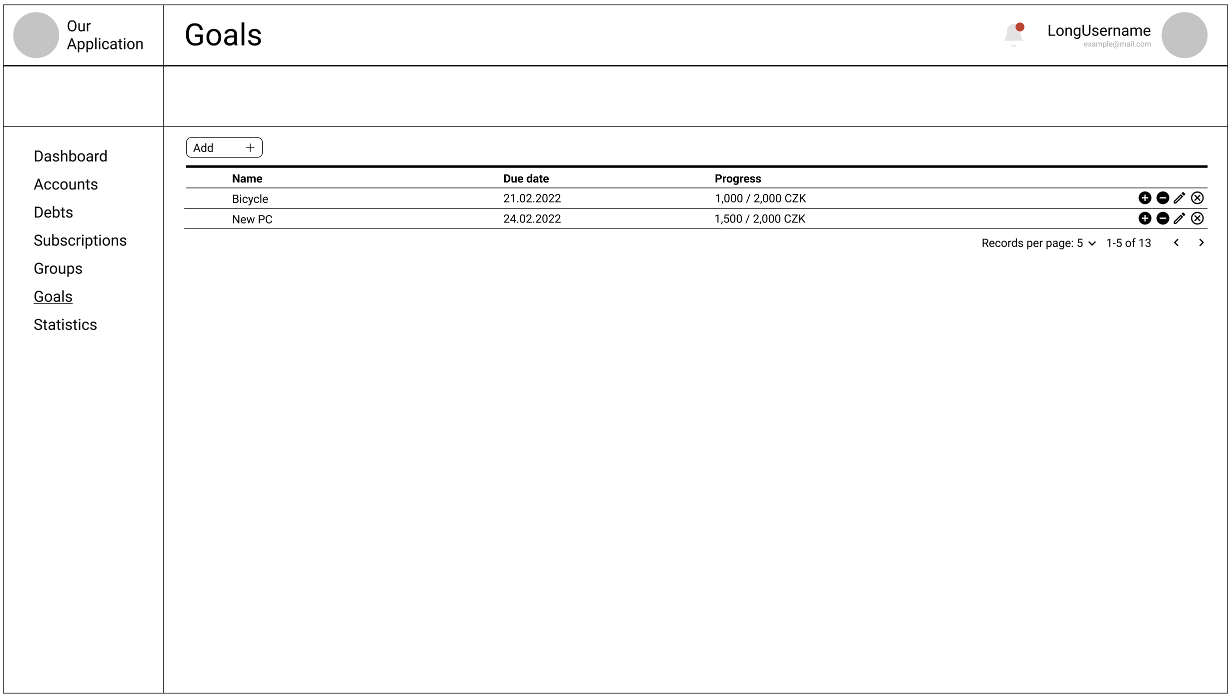
To	Name	Due date	Amount
@username	Beer	21.02.2022	50,000 CZK
Some friend	Pizza	24.02.2022	420 CZK

Records per page: 5 ⌄ 1-5 of 13 ‹ ›

From	Name	Due date	Amount
@prijmjme	vodka))	20.02.2022	4,000 CZK
Friend	Rohlik	24.02.2022	5 CZK

Records per page: 5 ⌄ 1-5 of 13 ‹ ›

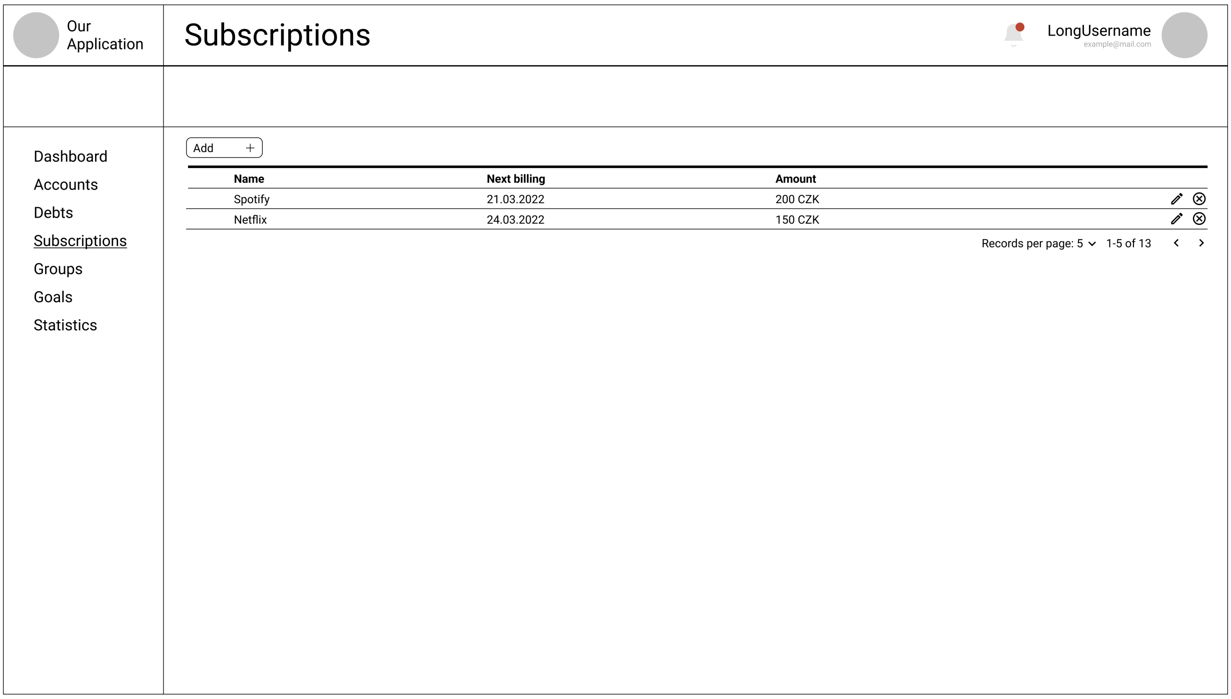
Figure A.10: Debt list page



Name	Due date	Progress
Bicycle	21.02.2022	1,000 / 2,000 CZK
New PC	24.02.2022	1,500 / 2,000 CZK

Records per page: 5 ⌄ 1-5 of 13 ‹ ›

Figure A.11: Goal list page



Name	Next billing	Amount
Spotify	21.03.2022	200 CZK
Netflix	24.03.2022	150 CZK

Figure A.12: Subscription list page

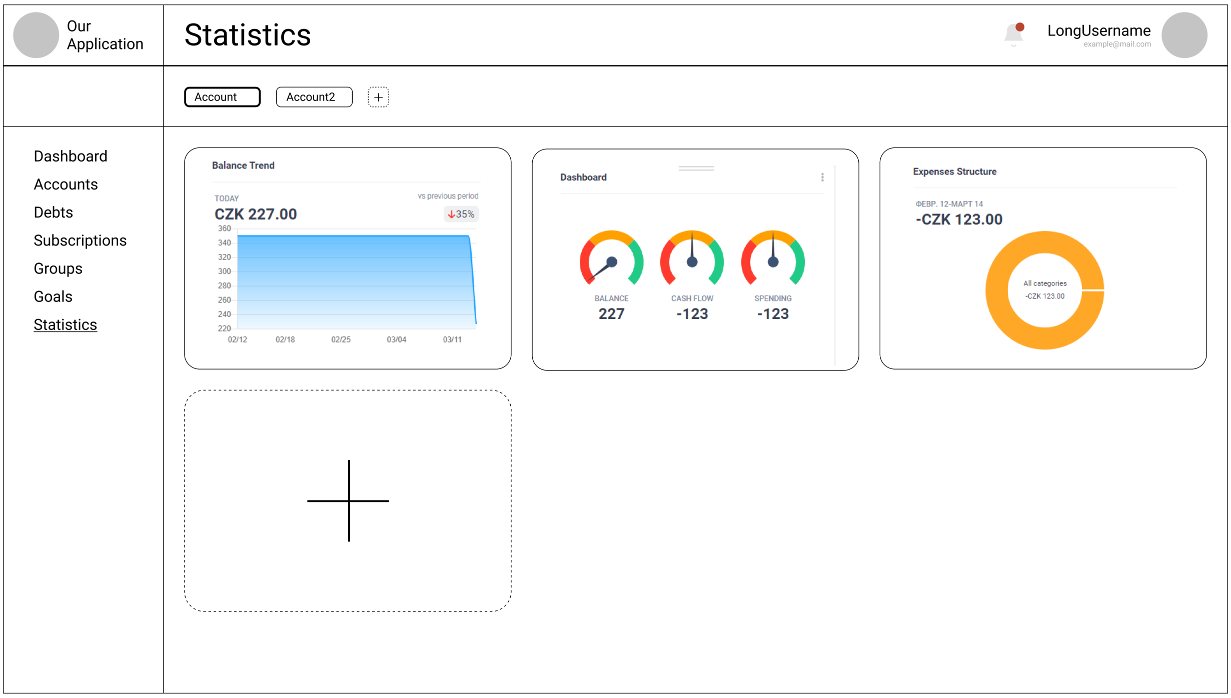
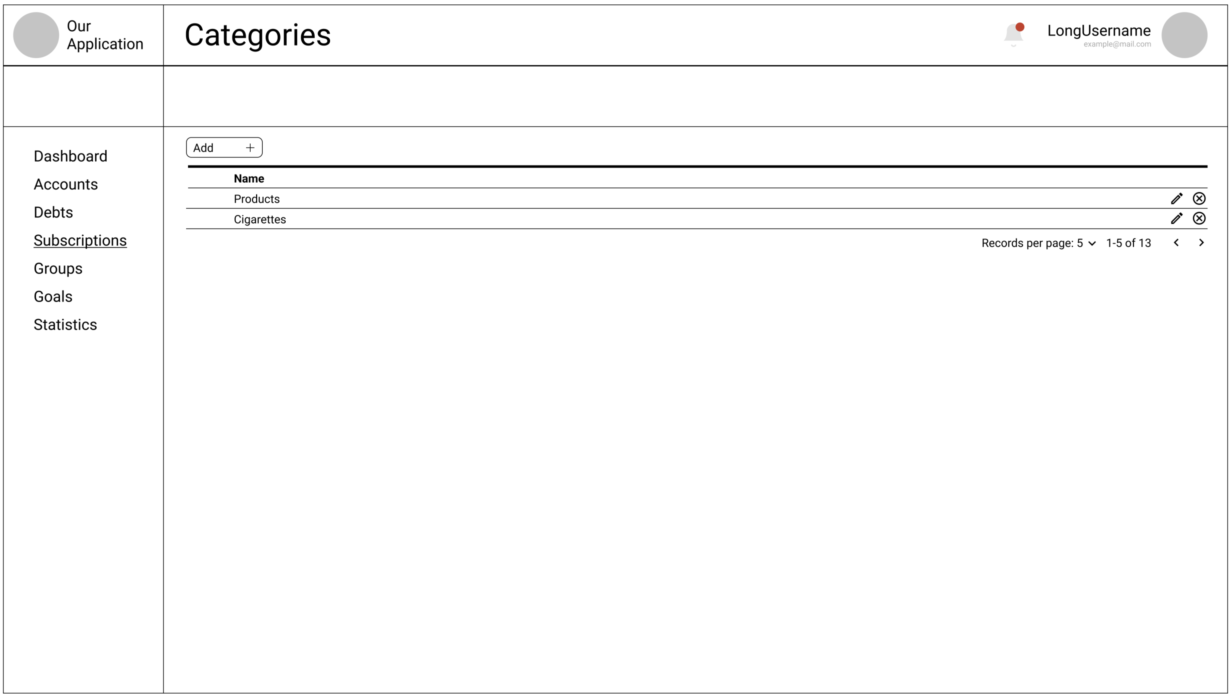


Figure A.13: Statistics page



Name	
Products	✎ ⊗
Cigarettes	✎ ⊗

Figure A.14: Categories page

Acronyms

API Application Program Interface. 2

CLI Command Line Interface. 7

CSS Cascading Style Sheets. 7

HTML HyperText Markup Language. 7

HTTP HyperText Transfer Protocol. 7

JSX JavaScript Syntax Extension. 7

JWT JSON Web Token. 25

SASS Syntactically Awesome Style Sheets. 43

SCSS Sassy CSS. 43

SFC Single File Component. 39

SPA Single-Page Application. 27

UI User Interface. 7

UX User Experience. 27

XSS Cross-Site Scripting. 11

Bibliography

[1] Nov. 8, 2019. URL: https://www.theseus.fi/bitstream/handle/10024/261970/Thesis-Elar-Saks.pdf?sequence=2 (visited on 04/23/2022).

[2] Apr. 30, 2022. URL: https://iopscience.iop.org/article/10.1088/1757-899X/801/1/012136/pdf (visited on 04/30/2022).

[3] *@angular/core vs react vs vue — npm trends*. Apr. 23, 2022. URL: https://www.npmtrends.com/@angular/core-vs-react-vs-vue (visited on 04/23/2022).

[4] *8 Benefits of Angular for Your Project — LIGHT-IT*. Apr. 23, 2022. URL: https://light-it.net/blog/8-advantages-of-angular-for-businesses-and-developers/ (visited on 04/23/2022).

[5] *Angular - Lifecycle hooks*. Apr. 22, 2022. URL: https://angular.io/guide/lifecycle-hooks (visited on 04/24/2022).

[6] *Angular - What is Angular?* Apr. 19, 2022. URL: https://angular.io/guide/what-is-angular (visited on 04/19/2022).

[7] *Angular vs React vs Vue: Which Framework to Choose in 2022*. Jan. 10, 2019. URL: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/ (visited on 04/23/2022).

[8] *API Reference — Vue.js*. May 5, 2022. URL: https://vuejs.org/api/ (visited on 05/05/2022).

[9] *Arrow function expressions - JavaScript — MDN*. Apr. 19, 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions (visited on 05/01/2022).

[10] Nick Babich. *5 Essential UX Rules for Dialog Design — by Nick Babich — UX Planet*. May 12, 2020. URL: https://uxplanet.org/5-essential-ux-rules-for-dialog-design-4de258c22116 (visited on 04/20/2022).

[11] *Benefits Of Using Personal Finance Apps. (English)*. August 2019, visited 27-03-2022. URL: https://www.finsmes.com/2019/08/benefits-of-using-personal-finance-apps.html.

[12] *Color Palette — Quasar Framework*. Apr. 8, 2022. URL: https://quasar.dev/style/color-palette (visited on 04/27/2022).

[13] *Cross Site Scripting (XSS) Software Attack — OWASP Foundation*. Apr. 20, 2022. URL: https://owasp.org/www-community/attacks/xss/ (visited on 04/24/2022).

[14] *Decorator*. Apr. 30, 2022. URL: https://refactoring.guru/design-patterns/decorator (visited on 05/01/2022).

[15] *Dialog Plugin — Quasar Framework*. Apr. 8, 2022. URL: https://quasar.dev/quasar-plugins/dialog (visited on 04/28/2022).

[16] Chameera Dulanga. *Quasar vs. Vuetify vs. Bootstrap Vue: Choosing the Right Vue.js UI Library — by Chameera Dulanga — Bits and Pieces*. Nov. 21, 2020. URL: https://blog.bitsrc.io/quasar-vs-vutify-vs-bootstrap-vue-choosing-the-right-vuejs-ui-library-cf566f61bc4 (visited on 04/24/2022).

[17] *Guide to UML diagramming and database modeling. (English)*. September 24, 2019, visited 27-03-2022. URL: https://www.microsoft.com/en-US/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling.

[18] *How to Use Environment Variables in Vue.js - DEV Community*. Sept. 2, 2019. URL: https://dev.to/ratracegrad/how-to-use-environment-variables-in-vue-js-4ko7 (visited on 04/27/2022).

[19] *HTTP Methods - REST API Tutorial*. May 24, 2018. URL: https://restfulapi.net/http-methods/ (visited on 05/01/2022).

[20] *Introducing JSX – React*. Apr. 24, 2022. URL: https://reactjs.org/docs/introducing-jsx.html (visited on 04/24/2022).

[21] *Issues · angular/angular*. Apr. 23, 2022. URL: https://github.com/angular/angular/issues (visited on 04/23/2022).

[22] *Issues · facebook/react*. Apr. 23, 2022. URL: https://github.com/facebook/react/issues (visited on 04/23/2022).

[23] Steve Krug. *Don't Make Me Think: A Common Sense Approach to the Web (2nd Edition)*. USA: New Riders Publishing, 2005. ISBN: 0321344758.

[24] *Layout — Quasar Framework*. Apr. 8, 2022. URL: https://quasar.dev/layout/layout (visited on 04/27/2022).

[25] *Lifecycle Hooks — Vue.js*. Apr. 24, 2022. URL: https://vuejs.org/guide/essentials/lifecycle.html (visited on 04/24/2022).

[26] R Malan and D Bredemeyer. *Functional Requirements and Use Cases*. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.436.4773&amp;rep=rep1&amp;type=pdf.

[27] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[28] *Object.freeze() - JavaScript — MDN*. July 20, 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze (visited on 05/01/2022).

[29] Zachary Orona-Calvert. *Compare the Top 3 Style Guides and Set Them Up With ESLint — by Zachary Orona-Calvert — Better Programming*. Apr. 15, 2020. URL: https://betterprogramming.pub/comparing-the-top-three-style-guides-and-setting-them-up-with-eslint-98ea0d2fc5b7 (visited on 04/25/2022).

[30] *Personal Finance Mobile App Market Trend Analysis to 2031*. May 5, 2022. URL: https://www.factmr.com/report/personal-finance-mobile-app-market (visited on 05/05/2022).

[31] *Quasar CLI — Quasar Framework*. Apr. 8, 2022. URL: https://quasar.dev/start/quasar-cli (visited on 04/21/2022).

[32] *Sass Introduction*. Apr. 27, 2022. URL: https://www.w3schools.com/sass/sass_intro.php (visited on 04/27/2022).

[33] Ivan Sedakov. *Finance Management Application (Backend)*. 2022.

[34] *SFC CSS Features — Vue.js*. Apr. 27, 2022. URL: https://vuejs.org/api/sfc-css-features.html#scoped-css (visited on 04/27/2022).

[35] *SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms — MDN*. Oct. 8, 2021. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (visited on 04/30/2022).

[36] *Template Refs — Vue.js*. Apr. 24, 2022. URL: https://vuejs.org/guide/essentials/template-refs.html (visited on 04/24/2022).

[37] *Template Syntax — Vue.js*. Apr. 24, 2022. URL: https://vuejs.org/guide/essentials/template-syntax.html (visited on 04/24/2022).

[38] *The Brutal Lifecycle of JavaScript Frameworks - Stack Overflow Blog*. Jan. 11, 2018. URL: https://stackoverflow.blog/2018/01/11/brutal-lifecycle-javascript-frameworks/ (visited on 04/19/2022).

[39] *The Most Popular Front-end Frameworks in 2022 - Stack Diary*. Feb. 19, 2022. URL: https://stackdiary.com/front-end-frameworks/#top-front-end-frameworks-for-2022 (visited on 04/19/2022).

[40] *UI Frameworks - How They Can Make Your Life Easier in 2017*. Sept. 24, 2017. URL: https://iconic-solutions.com/ui-frameworks/ (visited on 05/01/2022).

[41] *UML basics: The component diagram. (English)*. 15 Dec 2004, visited 27-03-2022. URL: https://softwareresearch.net/fileadmin/src/docs/teaching/WS13/SE/UML_basics-_The_component_diagram.pdf.

[42] *Virtual DOM and Internals – React*. Apr. 24, 2022. URL: https://reactjs.org/docs/faq-internals.html (visited on 04/24/2022).

[43] *Vue Forum*. Apr. 23, 2022. URL: https://forum.vuejs.org/ (visited on 04/23/2022).

[44] *vuejs/core: Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web*. Apr. 23, 2022. URL: https://github.com/vuejs/core (visited on 04/23/2022).

[45] *vuejs/vue: Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web*. Apr. 23, 2022. URL: https://github.com/vuejs/vue (visited on 04/23/2022).

[46] *What is Cordova — Quasar Framework*. Apr. 8, 2022. URL: https://quasar.dev/quasar-cli-vite/developing-cordova-apps/introduction (visited on 04/21/2022).

[47] *What is Front End Testing? Tools, Frameworks*. June 23, 2020. URL: https://www.guru99.com/frontend-testing.html (visited on 04/30/2022).

[48] *What is Vue.js? The Pros and Cons of Vue.js Framework*. Aug. 25, 2021. URL: https://www.spaceo.ca/blog/vue-js-pros-and-cons/ (visited on 04/23/2022).

[49] *Where To Get Support – React*. Apr. 23, 2022. URL: https://reactjs.org/community/support.html (visited on 04/23/2022).

[50] *Why Internationalization is Critical in Your App Development - IoT Software Blog - Geisel Software*. Apr. 27, 2022. URL: https://geisel.software/content/why-internationalization-critical-your-app-development (visited on 04/27/2022).

[51] Shane P Williams. *Can icons harm usability and when should you use them? — by Shane P Williams — UX Collective*. Apr. 14, 2019. URL: https://uxdesign.cc/when-should-i-be-using-icons-63e7448202c4 (visited on 04/20/2022).

[52] *Yarn vs NPM: A Comprehensive Comparison 7-Point Comparison*. Nov. 4, 2021. URL: https://phoenixnap.com/kb/yarn-vs-npm (visited on 04/25/2022).

[53] Kristina Zucchi. *Why Financial Literacy Is So Important. (English)*. January 2021, visited 27-03-2022. URL: https://www.investopedia.com/articles/investing/100615/why-financial-literacy-and-education-so-important.asp.