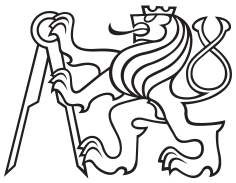
BachelorThesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Web application for management of personal finances

Filip Krul

Supervisor: Ing. Jiří Šebek
Field of study: Open Informatics
Subfield: Software
May 2023

Acknowledgements

I would like to thank my supervisor, Ing. Jiří Šebek, for his guidance, patience and advice.

I would also like to thank my family and friends for their support and encouragement and support during my studies and with this thesis.

Lastly, I would like to thank the communities of all the open-source projects I have used for their work and help with any issues I have encountered.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 21. May 2023

Abstract

This project aims to create a self-hosted web application for managing personal finances. This will include analyzing functional and non-functional requirements, finding an appropriate tech stack, planning the architecture, and implementing the application. The application will use SvelteKit, TypeScript, Prisma, and SQLite. It will then be deployable using Docker.

Keywords: Web application, Svelte, TypeScript, Finance, Software, Prisma

Supervisor: Ing. Jiří Šebek

Abstrakt

Tento projekt má za cíl vytvořit self-hostovatnelnou webovou aplikaci pro správu osobních financí. Toto zahrnuje analýzu funkčních a nefunkčních požadavků, nalezení vhodných technologií a nástrojů, plánování architektury a implementaci aplikace. Aplikace bude využívat SvelteKit, TypeScript, Prisma a SQLite. Poté bude možné ji nasadit pomocí Dockeru.

Klíčová slova: Webová aplikace, Svelte, TypeScript, Finance, Software, Prisma

Překlad názvu: Webová aplikace na správu osobních financí

Contents

Figures

Tables

Chapter 1

Motivation

These days, many people have multiple accounts with multiple banks, and keeping track of all of your finances can be challenging. Especially these days as the global economy slows down and more and more people are trying to find ways to save money. [Wan23]

From my experience, the most efficient way of keeping track of multiple accounts in multiple currencies is by using personal finance management apps. These apps usually help track and categorize payments and generate summaries for the user.

Currently, most existing Personal Finance Management apps are available as offline-only desktop or mobile applications which do not allow data synchronization between devices or as online-only applications which store your data on third-party servers where you have no control over them.

There are a few self-hosted personal finance managers, but none have all the features I would want from one I would consider using long-term or do not work that well from my experience.

Chapter 2

Analysis

Before deciding how the application will be developed and used, it is imperative to decide what it should be capable of doing. This will simplify selecting the most suitable tools job and provide guidance during the development process. Extensive research has been undertaken on multiple existing applications, and input has been sought from friends and family on which features they consider important. The ones chosen are those that have often been mentioned and are within the scope of this project.

2.1 Functional Requirements

2.1.1 User Accounts

Since this will be a self-hosted application, the way user accounts are managed will be quite different from most standard applications and will instead be based on how accounts are implemented in other self-hosted applications I am personally experienced with.

Most large-scale applications require complex user management with email confirmations, account recovery, and two-factor authentication since a single administrator cannot manage thousands upon thousands of users.

Meanwhile, self-hosted applications usually serve a single or sometimes up to a double-digit number of users per instance, which are usually managed by a single administrator. Therefore, most self-hosted applications usually have an administrator account which is created upon the first launch of the application and can create, edit, or delete other accounts. This allows the developer to spend less time on user account management so they can focus on other features.

2.1.2 Manual logging of payments

This will be the main feature of the application. Users can manually log incoming and outgoing payments in specific wallets. Each payment will also carry details such as the transferred amount, user-created categories such as Food or Electronics, date, recipient, and a user-created note.

2.1.3 Sorting and filtering payments

Another feature I plan to implement is the option to filter and sort previous payments. While some apps allow only one way of filtering at a time, it is much more convenient to be able to combine different filters. The filters I plan to implement are: By date, amount of transferred money, and payment category.

2.1.4 Dashboard

The dashboard will be the first thing the user sees, so it will provide a simple summary of important information, such as a run-down on spending during the past month, what are some upcoming payments, and debts the user needs to repay soon.

2.1.5 Multiple wallets per user

Many people have multiple accounts with different currencies, so being able to track them independently is vital.

Sharing one wallet between multiple users will also be possible, so it is easier for them to track shared spending, such as food and housing, if they live together.

Each wallet will also have its own settings for currencies and tags.

2.1.6 IOUs

IOUs ('I Owe You's) will be a way to log outstanding unofficial debts by and to the user. This is a feature I have not seen in any existing application and would appreciate having.

2.2 Non-functional Requirements

2.2.1 Selfhostability and data ownership

The practice of self-hosting involves running an instance of a desired service on one's server [BJV+22]. The main draw of this lies in owning one's data, hereby preventing its sale to third sides for advertising purposes. This is especially important in fields relating to finance. [REC+11]

2.2.2 Easy deployment

A crucial aspect of publishing a self-hosted application is the ease with which the application can be deployed. Preferably, it should be possible to deploy with only a few pre-made terminal commands that are simple to modify to suit the user's needs.

This is commonly done by publishing a docker container, eliminating the need to install prerequisite software on the user's machine and isolating the software from outside problems.

2.2.3 Performance

Experiencing an unresponsive or slow website can be one of the most frustrating experiences when using a computer or a mobile phone. Theciteore, prioritizing performance is crucial, and ensuring the app remains responsive even on slow devices or with slow internet will be one of the key objectives [ZFL+19]. This focus on performance also includes ensuring the app works well even when the database contains large amounts of data.

2.2.4 Security

For a finance-tracking application, security is a crucial aspect [GW05]. Self-hosted applications have an advantage in this field because they are only responsible for securing themselves, and the user is capable of adding extra security features, such as allowing only HTTPS communication or even hiding the whole service behind a VPN. Running in a docker container can also bring significant security benefits [CMDP16].

2.2.5 Usability

Maintaining a simple and consistent user interface is also important, and it ensures that even a user not used to online services can easily understand and navigate the web app. This simplicity should also improve the app's usability for more tech-savvy users. It is also necessary to pay attention to usability standards researched by experts [Joh20], which can often be found integrated into existing UI frameworks or component libraries.

2.2.6 Interoperability

A modern application should be accessible from anywhere and work on most devices. This ensures that users can access and manage their financial information on any device they have at hand. [DP08]

2.3 Use case diagrams

This section shows the necessary use cases that outline the various interactions and functionalities within the web app. The use case diagrams serve as a visual representation to aid in understanding the different functional use cases and show how they relate to each other. Four diagrams depicting different parts of the project were created to make the information more digestible and organized. These diagrams highlight the core features of the web app, as well as the interactions between the users and the system.

The only actor in most of these diagrams is 'User', who symbolizes the generally expected user of this app. In the diagram related to authentication, there is also an Administrator user who can act like a regular user in all other scenarios.

2.3.1 Authentication and user management

The diagram in 2.1 depicts how an administrator can manage users and what authentication-related actions a user can perform.

2.3.2 Dashboard

The diagram in 2.2 depicts how a user can interact with the dashboard, which also serves as the home screen.

2.3.3 Managing IOUs

2.3 depicts the ways a user can interact with his IOUs. Each user has their own set of IOUs which are not shared between users. A user can create or browse IOUs. While browsing them, the user can mark them as already paid, edit them, or even outright delete them.

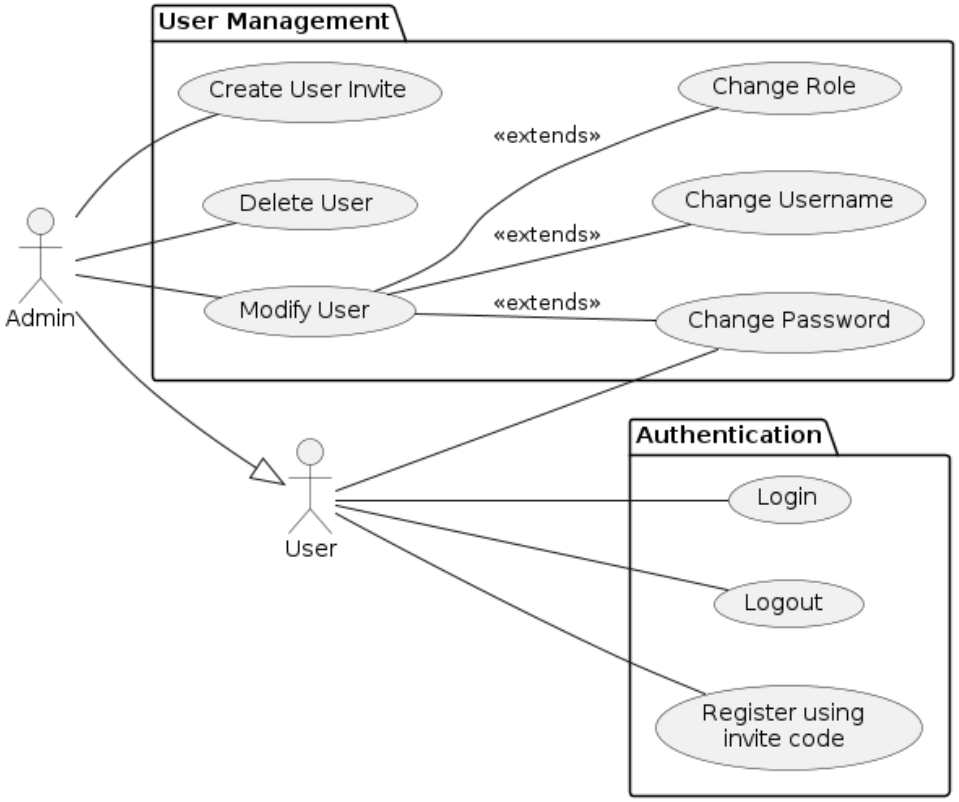


Figure 2.1: Use cases relating to user management

2.3.4 Wallet and transaction

The diagram in 2.4 depicts how a user can interact with a wallet and the payments and categories within the wallet.

2.4 Technology stack

After deciding on the application's functional requirements and use cases, the appropriate tools for implementing those goals must be selected. No specific tools are required to create an application like this one, but selecting the wrong ones can be limiting if they do not do everything needed or add unnecessary work if the way to implement every necessary part is too complex. Therefore, the focus of this section is the aim is to find the best tools that fit the requirements.

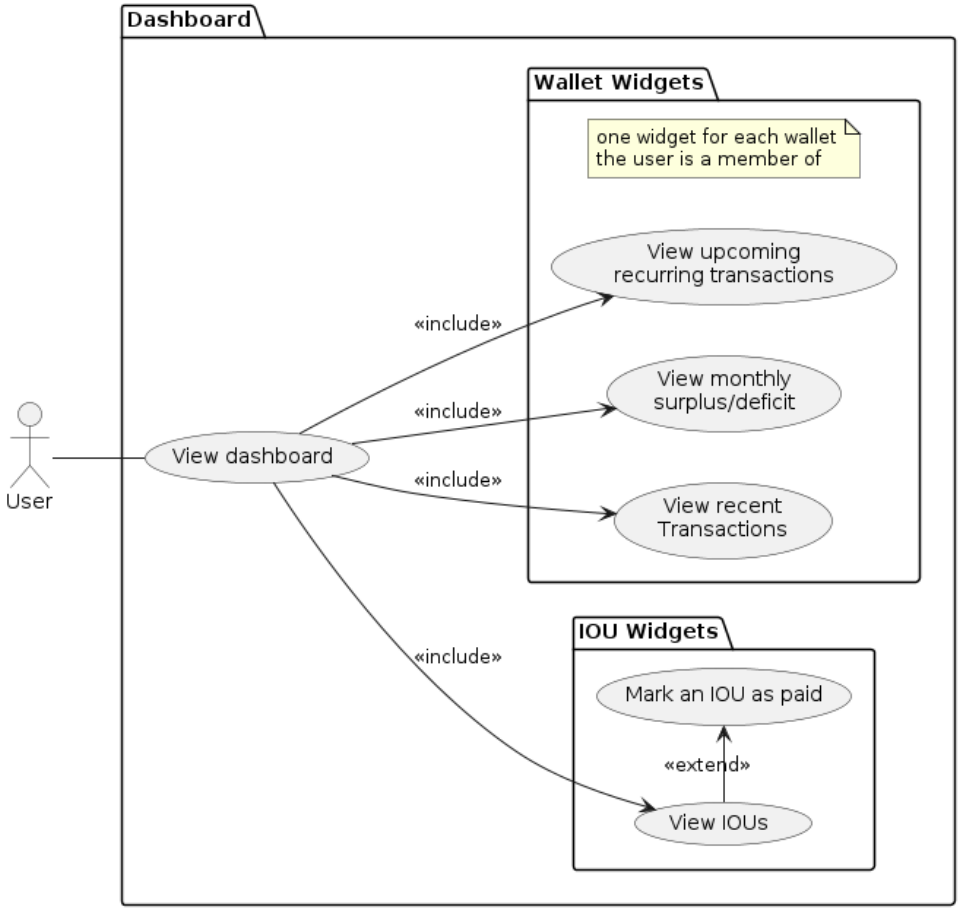


Figure 2.2: Use cases relating to the dashboard

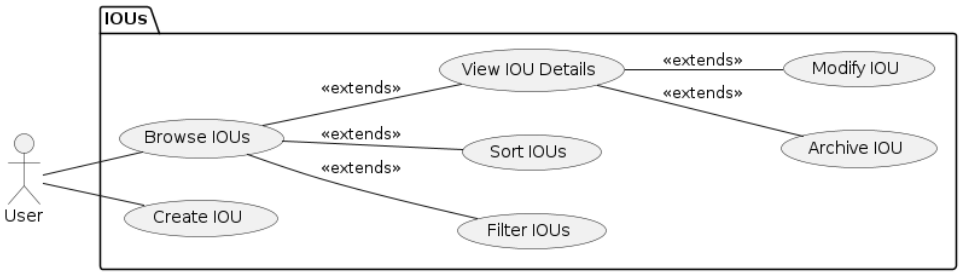


Figure 2.3: Use cases relating to IOUs

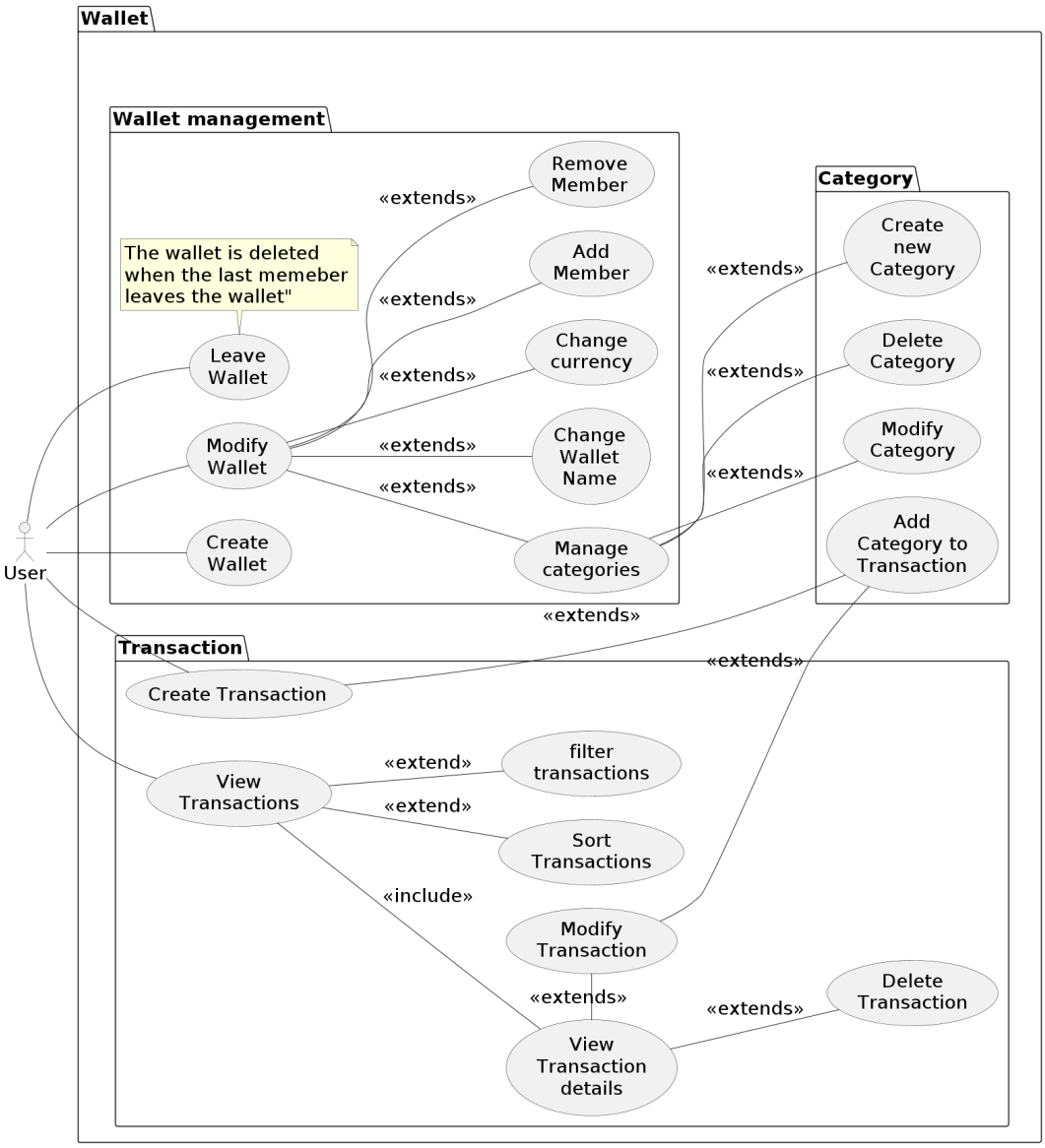


Figure 2.4: Use cases relating to wallets and transactions

2.4.1 Type of application

The main choice to make is how the user will interact with the back end. The main options are mobile, desktop, or web apps, each with its own advantages and disadvantages.

Desktop application

I have the most experience with desktop applications, so I would have the easiest time programming it and could use everything I have learned in school.

Another advantage of desktop applications is that they are quite easy for users to control since the screen is larger, which can be achieved by opening a web app on a computer.

Sadly, they also come with several disadvantages, such as not being portable and desktop UI frameworks not being as developed and modern as mobile and web UI frameworks.

Creating a program that works on macOS without owning an Apple device is also almost impossible. While it is possible to create OS-agnostic desktop applications using a framework like Electron, that seems unnecessary as it is just a more complex web application with few benefits and many negatives for this application. [SA20]

Mobile application

Mobile applications are significantly more portable than desktop applications. However, I do not have as much experience with them as with desktop applications, so programming them would be extra work.

They also have the disadvantage of being limited to a small screen, so doing more complex tasks is harder than on a desktop. Another disadvantage they share with desktop applications is the requirement to own an Apple device to be able to develop for iOS. [Bal21]

Web application

Web applications are as portable as mobile applications and can be accessed on computers with larger screens. The downside of using a web application is that they require a constant internet connection to be accessed, but this can be remedied by making the web app into a PWA (Progressive Web Apps), which are becoming increasingly popular.

Combination of the above options

Another option is to have both desktop and mobile applications to cover more platforms, but this would require significantly more development and testing time since that would require planning, implementing, and testing two applications instead of just one.

Some frameworks, such as Flutter, do allow compilation for multiple platforms, but even that requires platform-specific code, so it would still be more complex than focusing on one platform. [And22]

The choice

In the end, a web application seems like the best choice. The reason for this is the possibility of making a single program while allowing it to run on almost all platforms. The fact that they require a constant internet connection is not much of a problem in 2023 because most developed countries have incredibly high mobile internet coverage. It also allows the app to work on any Apple device with minimal extra work, even though I do not own any Apple device myself.

Another reason for this choice is that I would like to gain experience with web development, as the consensus around the internet is that web applications are the future of UI, so I would like to gain experience developing them while still in school.

2.4.2 Front end

Given the decision to make a web-based application, one must decide which front-end framework to write it in. This choice will also affect the choice of back-end language since some back-end frameworks are built to work with specific front-end frameworks, such as Vue.js with Nuxt or Svelte with SvelteKit.

Flutter

Flutter is a multi-platform framework developed by Google [tea19]. Its code is written in Dart – a strongly-typed object-oriented language also developed by Google to support Flutter [Tea23]. This allows it to feature features made specifically for writing component-based applications. However, Dart's focus on Flutter limits its general usefulness, resulting in significantly fewer community-created libraries compared to JavaScript.

One of the main benefits of coding in Flutter is that it can be compiled for Desktop, Mobile, and Web. While this sounds great, the usability is limited since different platforms often require different components, which results in a messy codebase [And22].

Another important thing about Flutter is its integration with Google's Firebase [Fir23]. While this helps create easy-to-scale applications, it also goes against the principles of self-hosting since Firebase cannot be hosted locally.

Vanilla JavaScript

Plain JavaScript is the simplest way to make websites. It is comparatively easy to learn since all other JavaScript frameworks also require the knowledge of plain JavaScript.

Another advantage of plain JavaScript is its speed. Since there is no virtual DOM or any abstraction layers, it can be significantly faster than a framework like React. This makes the application faster to load and allows it to be more responsive.

The disadvantage of this simplicity is that it lacks a structure of a more advanced framework, making it harder to maintain a larger project. Another disadvantage is the lack of features which are the main draw of frameworks such as React or Svelte.

Vue.js

Vue is a JavaScript framework that uses HTML-like syntax for structuring websites [You19b]. It has a large community of developers and is generally considered easy to learn.

One of its disadvantages is that it is compiled at runtime, which may cause performance issues on slower devices. It can also increase the bundle size sent to the user's browser, increasing loading times and data consumption [Xu21].

Svelte

Svelte is a modern component-based JavaScript framework focused on lightweight and performant websites. It achieves its lightweight bundles and great speed by not using a virtual DOM and building the site at compile time.

It has a pleasant syntax that greatly resembles standard JavaScript, HTML, and CSS and is written in separate .svelte files for each component which makes the project easier to navigate. It is also very concise compared to other frameworks [Har19]. This ease of use is also one of the main reasons it is the front-end framework developers are the happiest using as of 2023 [SG].

Its main disadvantage is how new it is. This means that the community around it is smaller than other more popular frameworks, and there are not as many third-party libraries.

React

React is currently the biggest JavaScript framework with a massive community of developers behind it [SG]. Its key feature is its virtual DOM (Document

Object Model). The virtual DOM is an in-memory representation of the website's actual DOM. This allows it to minimize the number of operations performed on the actual DOM but brings with it additional overhead and limits flexibility since directly editing the actual DOM can cause errors.

It is also component-based, which can significantly simplify development if the website features components that often repeat across different pages.

Another significant factor to consider when working with React is the need to use .jsx (JavaScript XML) files instead of standard JavaScript and HTML. This allows one to write HTML-like code in JavaScript-like files, but the syntax seems unwieldy and hard to work with.

The choice

In the end, the decision fell on Svelte. Its community might be smaller, but it is still quite helpful and friendly. Its component-based approach is also appealing since it simplifies individual components and makes them easier to reuse.

Another reason for choosing Svelte is how lightweight and fast it is. This will help make the app more responsive while allowing it to run on older devices. Related to this, its ability to run in a browser with disabled JavaScript will please the privacy-focused group, which has a significant overlap with the self-host community.

The fact that it is quite modern and does not have that many libraries also does not present an issue considering this project's scope compared to major for-profit websites. Its modernity is a plus since it has many features not yet implemented in older frameworks.

2.4.3 Back end

Choosing the correct back-end language and framework is one of the most important choices to make when building a web application. The decision affects development time, maintenance difficulty, and scalability.

SvelteKit (NodeJS framework)

SvelteKit is a Node.js framework for building applications on top of Svelte that also provides server-side functions, can work as a simple Rest API, directory-based routing, and allows for more advanced things such as full or partial SSR (Server Side Rendering). This allows SvelteKit to build a full-stack application with a single back-end and related front-end frameworks. [Sve23a]

It uses Vite to build the site server-side for the first load and then intercepts all updates and hydrates the client side as required. Vite also enables the use of Vitest – one of the fastest JavaScript unit-testing libraries. [You19a]

It is also extremely new, with the 1.0 version being released on December 14th, 2022, after spending years in beta. This means there is not extensive documentation compared to similar frameworks, but it already provides features on long-term roadmaps for competing frameworks.

One main disadvantage is that it is not the best option when building extensive Rest APIs since that is not its primary purpose.

C# (asp.net)

C# is one of my favorite languages, and Asp.net is a well-tested back-end framework created and maintained by Microsoft. This means that it is one of the most stable and tested web back-end platforms on the market and can be used to build application back-end.

It has a large community of users and allows for the use of any .net and C# libraries. Therefore, it is suitable for building large projects that require robust, full-featured back-end and enterprise-level projects that require high scalability and easy maintenance and that need to be highly testable.

Java (SpringBoot)

SpringBoot is similarly enterprise-focused like ASP.NET, but it is written in Java, which does not align with my personal preferences.

The main difference between SpringBoot and Asp.net is that SpringBoot is open-source and written in Java, so the whole framework is slightly more advanced than Asp.net. The fact that it is written in Java allows it to run on any system, while asp.net has historically been intended to run on Windows servers.

PHP

There are several PHP frameworks that could be used for this project, but PHP as a whole seems to be quite old compared to modern features of frameworks such as SpringBoot or Express. Some people may consider its age a benefit because it has stood the test of time, but the lack of modern features and aged syntax hurt it too much to be a good choice in a modern project.

Node.js (Express)

Express is a popular JavaScript framework that includes features such as routing, template rendering, good integration with other Node.js libraries, and can act as a middleware for client requests.

It is also lightweight and easy to use, making it an excellent option for beginners. Sadly, it does not provide any extra integration with Svelte and does not provide almost any extra features compared to SvelteKit.

The choice

In the end, SvelteKit is the obvious option. It provides all the features I need with Svelte-specific quality-of-life features such as Form Actions, allowing the whole web app to be built in a single codebase.

There are some drawbacks, such as not being an amazing API framework compared to a dedicated endpoint framework such as Express in Node.js or FastApi in Python. However, that does not matter for this project since the plan is to create only a Svelte-based website and use form actions for most client-server communication.

2.4.4 Data storage

The last piece of the proverbial puzzle is figuring out how to store user data. Since this will be a smaller-scale project with users in only dozens per instance and the number of logged payments in thousands instead of millions, our approach to storing data may differ from an enterprise-level project.

Database or something else?

The most obvious option is storing the data in a full-fledged database, but with the small scale of our project, storing all the data in a massive JSON file or an Excel database might be feasible. Their main disadvantage is that they slow down proportionally to how many entries are stored in them, and there is no telling how much data some users will need to store. It is also considered a bad practice, so a traditional database seems the correct option.

What kind of database

The main decision is between relational non-SQL databases. Considering the general data structure the app will need, a relational database is the obvious choice.

Server-based databases. Server-based databases are usually robust, feature-rich database systems that run in a separate process. It is considered reliable, supports many data types, and allows for the use of server-side scripts that run directly on the database engine. It scales extremely well with large amounts of data and concurrent queries. [Che22]

SQLite. SQLite is a small, lightweight, self-contained database system stored in a single file. There is no separate process that runs the database, which brings its own benefits and negatives. The main effect is that the client (in our case, the SvelteKit server) handles the queries. This means the database file must be stored locally and cannot be accessed over a network. It also does not scale as well as the full-fledged databases mentioned above. [BPP15]

The choice of long-term data storage

While using a JSON file seems viable, it is not a good idea long term, and it is considered a bad practice to store important data in what is essentially a glorified text file. This narrowed the selection to deciding between a Server-based and SQLite database.

While PostgreSQL or MariaDB are better options for most projects, they may not be necessary for a project like this one. The large scalability is unnecessary since there will only be up to a few dozen users per instance, and the advanced features are unnecessary for a project like this one.

On the other hand, SQLite offers an easy setup, is more lightweight, and can be backed up by just copying one file, which is quite helpful for a small, self-hosted application, so it is the best choice in this situation.

2.4.5 Other tools I plan to use

Before starting the project, it is also essential to find additional tools that will make programming and testing easier. A brief and non-exhaustive list of tools and libraries that will simplify the development process has been compiled in this chapter. Of course, many other libraries will be used in this project, but these are the most significant.

TypeScript

TypeScript is a superset of JavaScript that adds static typing to the language. This improves code readability and allows developers to catch errors that typically happen at runtime before compiling the project [Mic23a].

Playwright

Playwright is a modern open-source end-to-end testing library. It works on all platforms, supports all significant rendering engines like Chromium, WebKit, and Firefox, and can emulate mobile browsers [Mic23b].

The main difference between Playwright and Selenium is that Playwright has a more modern and user-friendly API, while Selenium is more mature and has more user-created libraries.

Vite and Vitest

Vite is a fast JavaScript build tool and development server for building web projects. It is considerably lighter and faster than most competitors and provides modern features such as "hot-reload," which allows the developer to make changes to the codebase without recompilation. [You19a]

This choice was obvious because Vite is bundled with SvelteKit by default. Another reason for this choice is Vitest – a JavaScript unit-testing library built specifically for use with Vite. It provides a simple yet fast way to write and run unit tests for web development projects compiled with Vite. The fact that it uses vitest makes it extremely fast compared to other testing libraries.

Prisma ORM

Prisma is an open-source ORM (Object-Relational Mapping) tool written in Node.js [PD23a]. It provides an efficient, type-save way of interacting with a SQL database from a TypeScript project.

Its defining feature is its ability to generate type-safe classes representing structured data in a relational SQL database that can be directly used in a TypeScript project. Prisma also provides a query language called Prisma Client, which allows the developer to query the database without needing to write SQL queries manually.

Chapter 3

Design

3.1 Architecture

There are two common ways of structuring SvelteKit projects. The first is having the front-end written in Svelte, with simple data processing done on the back-end, which then communicates with a more complicated back-end written in another language such as C# or Java, or even a separate Node.js project. These two back-end systems are then connected using an API. While this can be beneficial if there is another use for the exposed API, such as a mobile app or better scaling, there is no need for those features in this project.

The second has a more complicated back-end written in JavaScript or TypeScript, and this back-end is directly called using JavaScript function calls. both of these reside within a single monolithic codebase, it significantly simplifies project management and testing and improves development speed.

For the sake of project analysis, the front-end and back-end should be considered separate entities even though they are stored in the same codebase.

Figure 3.1 also depicts the general structure of the project and elaborating on component and its purpose.

3.2 Front-end

The front end comprises three main parts: the request hook modifies incoming and outgoing requests, the server-side logic that handles data loading and processing, and the client-side front-end that designs individual pages and components. Each of these components receives a more detailed description in the Implementation chapter.

Which screens are necessary

The first step in planning the front end was deciding which pages were necessary to achieve all the goals set in the use case analysis. For this purpose, a screen flow diagram was created. This diagram can be seen in 3.2.

UI planning in Figma

The second step in planning the front end's general look was designing a mockup in Figma. These mockups are not styled since styling and color themes will be provided by a component library during implementation. Instead, the focus was on the general layout of components and pages.

In the final product, the design has also changed because the original design either did not feel great to use, had some accessibility annoyances, or did not look that great with the theme used for the project. Consulting a more experienced UI designer could have avoided these mid-development redesigns.

An example of these mockups can be seen in 3.3, and the whole mockup can be found in the folder provided with this thesis.

3.3 Back-end

The server could also be split into multiple categories, most of which are linked. This separation into categories is not directly implemented in the

code but is more of a categorical analysis of each part's general purpose and behavior.

The first category is the back-end files containing the code interacting with the database. These methods will be called from the different page controllers and will be stored in the `/src/lib/server` directory. SvelteKit ensures that any files written in this directory can not be imported from client-side code. This feature facilitates the writing of code the end user should not have access to without having to deal with securing it manually.

The second category includes different utility files, which include methods relating to loading and managing currencies and data validation. Another type of these files are files containing the definition of custom TypeScript types and errors used throughout the project. While some of these are also imported from the front-end, their inclusion in this section is justified since they will mainly be used in back-end code.

The last category files for implementation instances of different libraries I will be using, like Prisma, Lucia, and Pino. Since both Prisma and Lucia interact with the database directly and have privileges that should not be given to any user, they will also be put in the `/src/lib/server` directory.

The Implementation chapter of this thesis also delved into more detail about the used libraries.

3.3.1 Database

As stated in the technical details planning chapter, this project uses Prisma to connect to a locally stored SQLite database. While some parts of the database structure are dictated by some libraries I am using, the rest have been created with the intent of keeping the database simple while having all the necessary data. The database structure can be seen in 3.4.

The database contains these tables:

- **AuthUser:** Entries in this table represent the users of this application.
- **AuthSession:** Each entry in this table represents a browser session. The sessions are managed by the Lucia library, which also dictates the exact structure of this table.

- **AuthKey:** Entries in this table handle different login methods. While this application supports only login using a username and password, it would make it easier to add other kinds of authentication, such as OAuth or email-based login. It is also required by Lucia.

- **Wallet:** Entries in this table represent individual wallets users can own. A wallet can belong to multiple users and can contain many payments.

- **Payment:** Entries in this table represent individual payments. A payment may have a category but does not have to.

- **Category:** Entries in this table represent a category inside a wallet. They are not shared between wallets, so they can be set up differently in each. The color is for UI only and holds no informational value.

- **RepeatingData:** Each entry in this table must be linked to an existing payment and serves as a way to keep track of repeating payments. They contain just the type of repetition period, such as days, weeks, or months, and how many units of time they repeat.

- **InviteCode:** This is solely utilized for storing the invite code to this application. The entries are automatically deleted a week after they expire. This table is also not referenced from any other table because it is used purely to store which invite codes are valid, which are expired, and which have been used.

3.4 Component diagram

While some may consider creating a component diagram for a monolithic SvelteKit project unnecessary, the decision was made to include it to show how different parts of the project interact. Creating the diagram in figure 3.5 was more challenging than it usually is since this project is not structured the same way most projects in object-oriented languages. Nevertheless, the diagram remains reasonably understandable.

It is also worth mentioning that none of the APIs in this diagram are actual APIs and are shown as such only for illustrative purposes.

3.5 Sequence diagrams

The following diagrams are sequence diagrams. They depict the sequence of operations between different actors and components necessary to achieve a particular goal. These diagrams provide a simple way to understand how different parts of the system interact.

Four diagrams created, each depicting a different processes common in this program. While this is by far not an exhaustive list of processes that this program performs, it is sufficient to illustrate the general interactions and flow of the system. For the sake of brevity, only two are included in this document since the processes are mostly similar and differ mostly in the data passed between the different components, with the other two being available in the folder containing the analysis that came together with this thesis.

The first diagram shown in figure 3.6 depicts the process of a user logging into the system. This is a simple procedure, but it shows the system's common components and how they work with each other.

The second diagram shown in figure 3.7 depicts another process relating to how the system authenticates the user based on a stored session cookie. The "Page Load" in this diagram combines the request hook and the load function of a specific page.

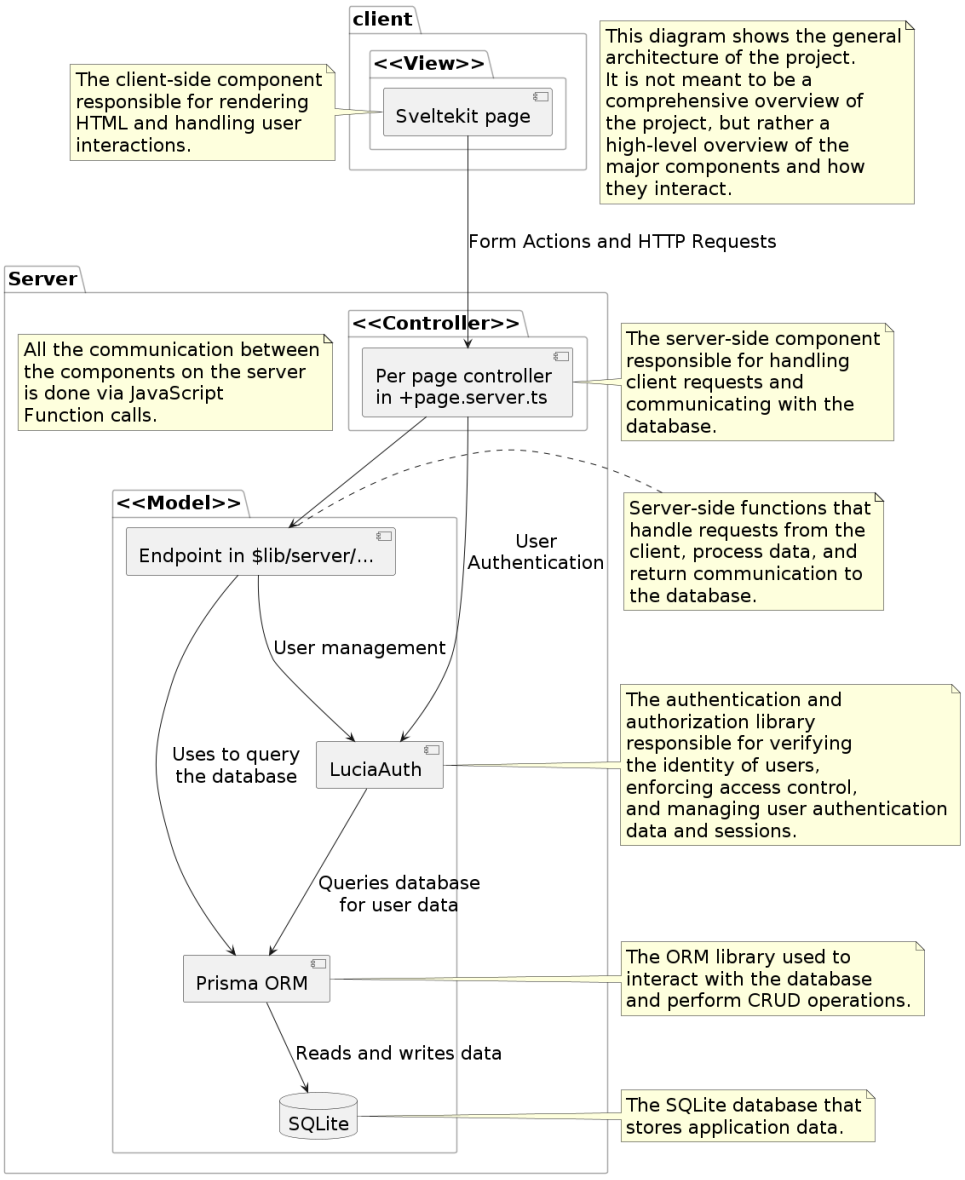


Figure 3.1: Diagram depicting the architecture of the program

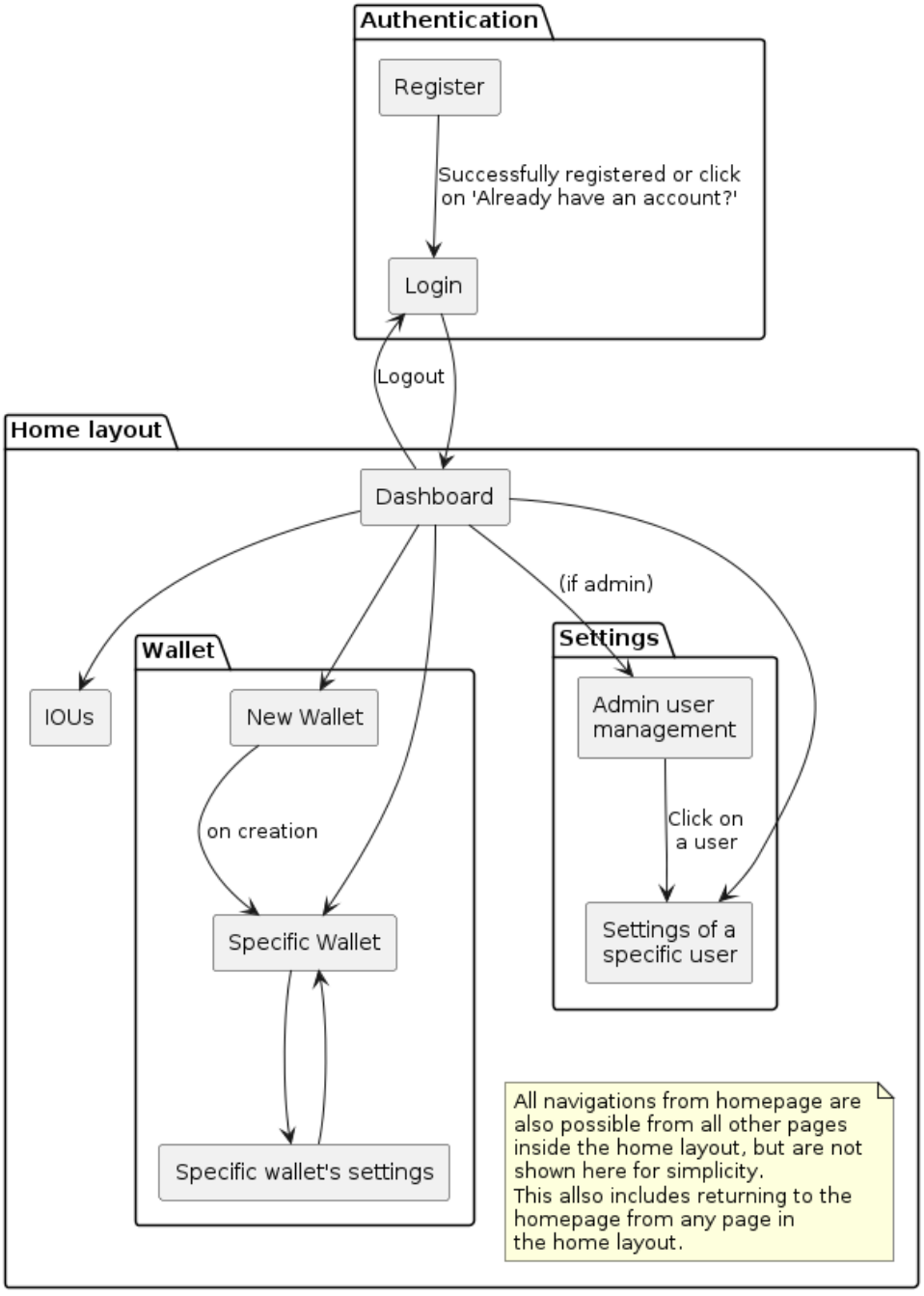


Figure 3.2: Screen flow diagram

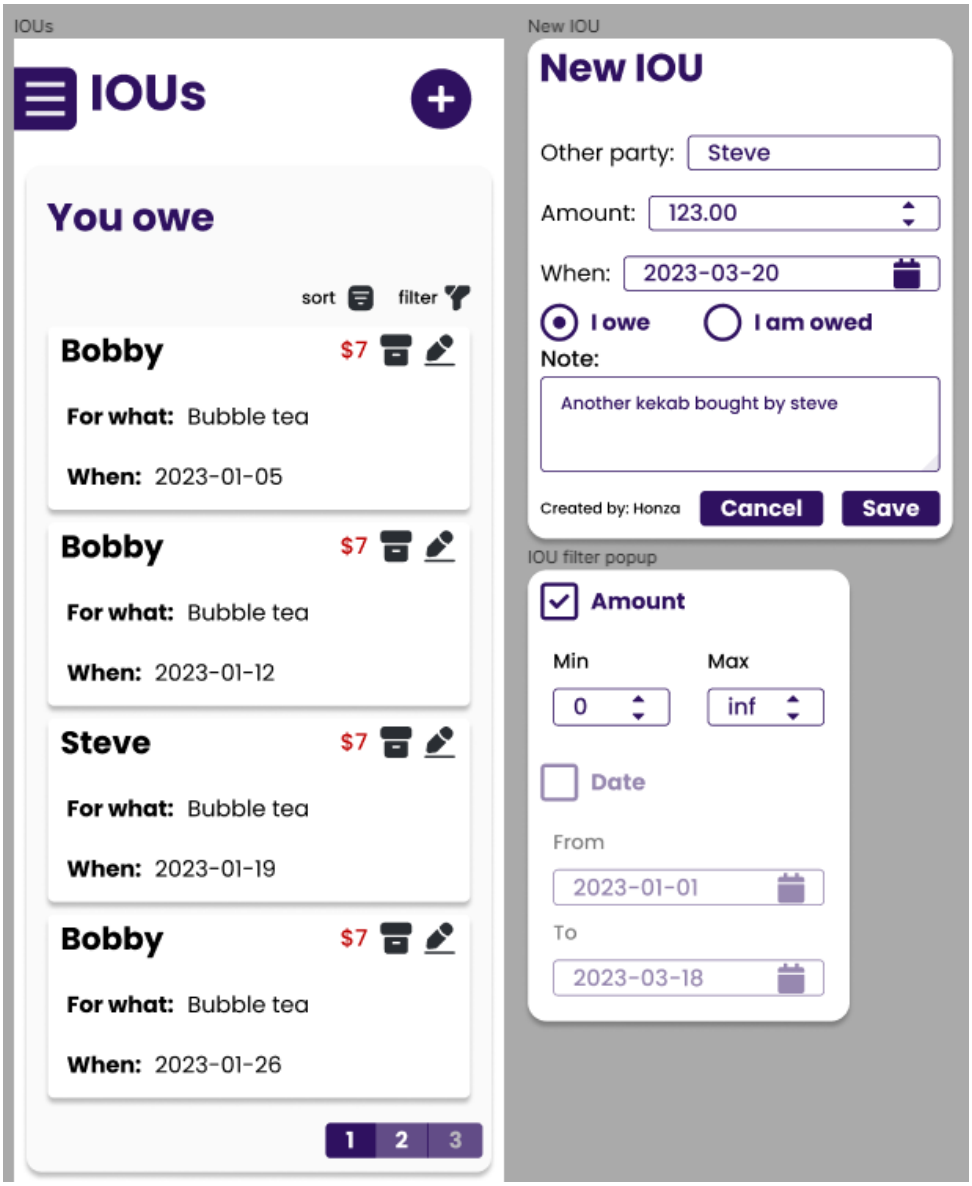


Figure 3.3: Figma mockup of the dashboard

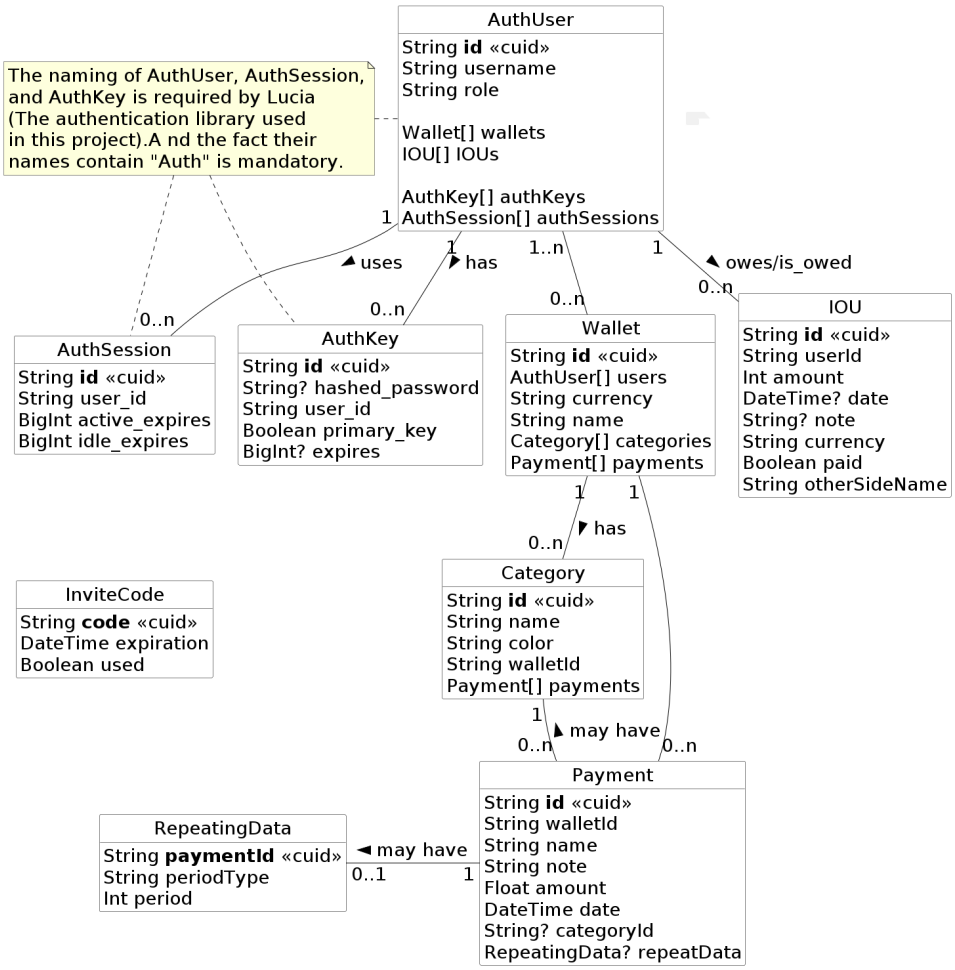


Figure 3.4: Database structure diagram

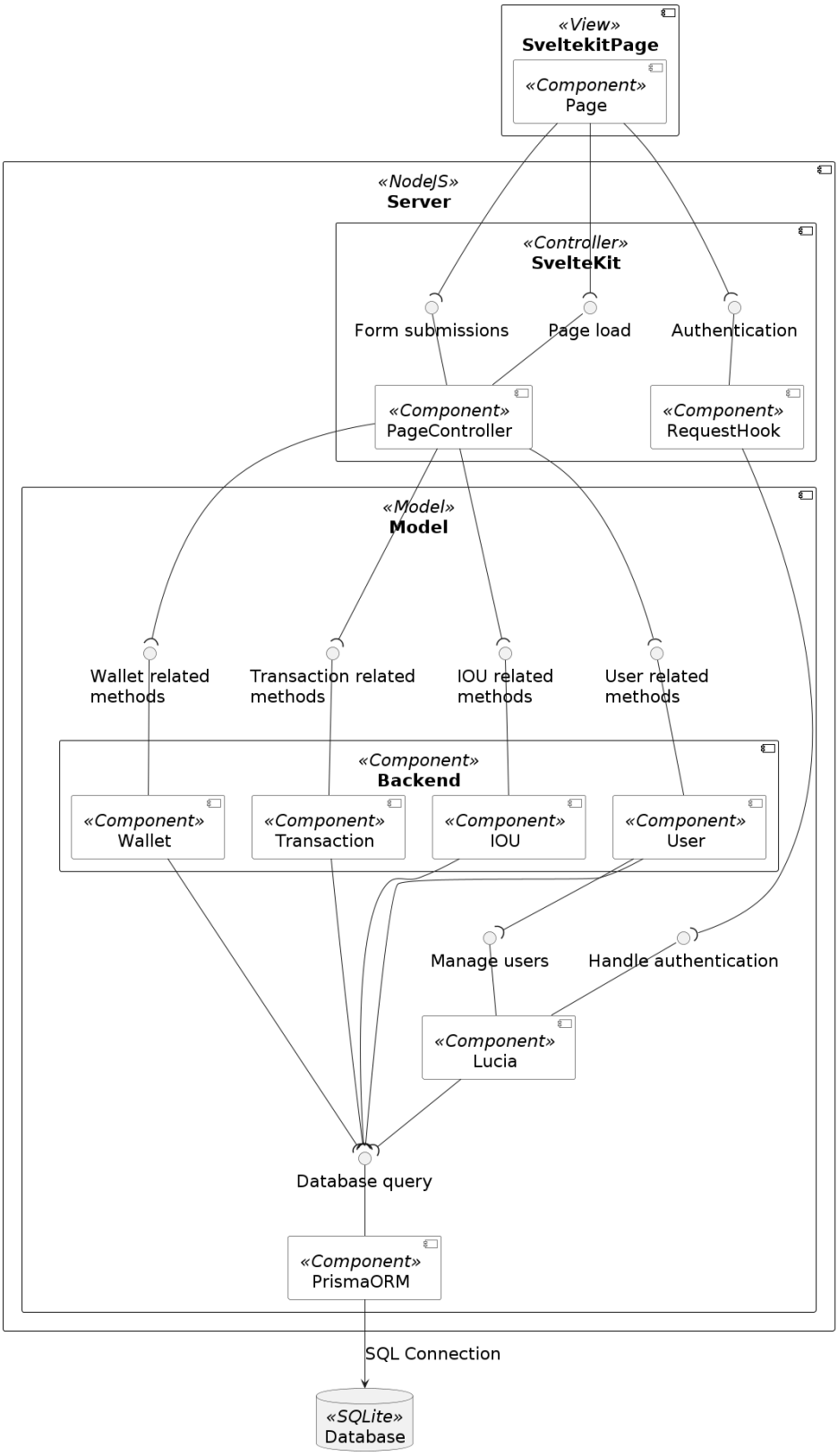


Figure 3.5: Component diagram

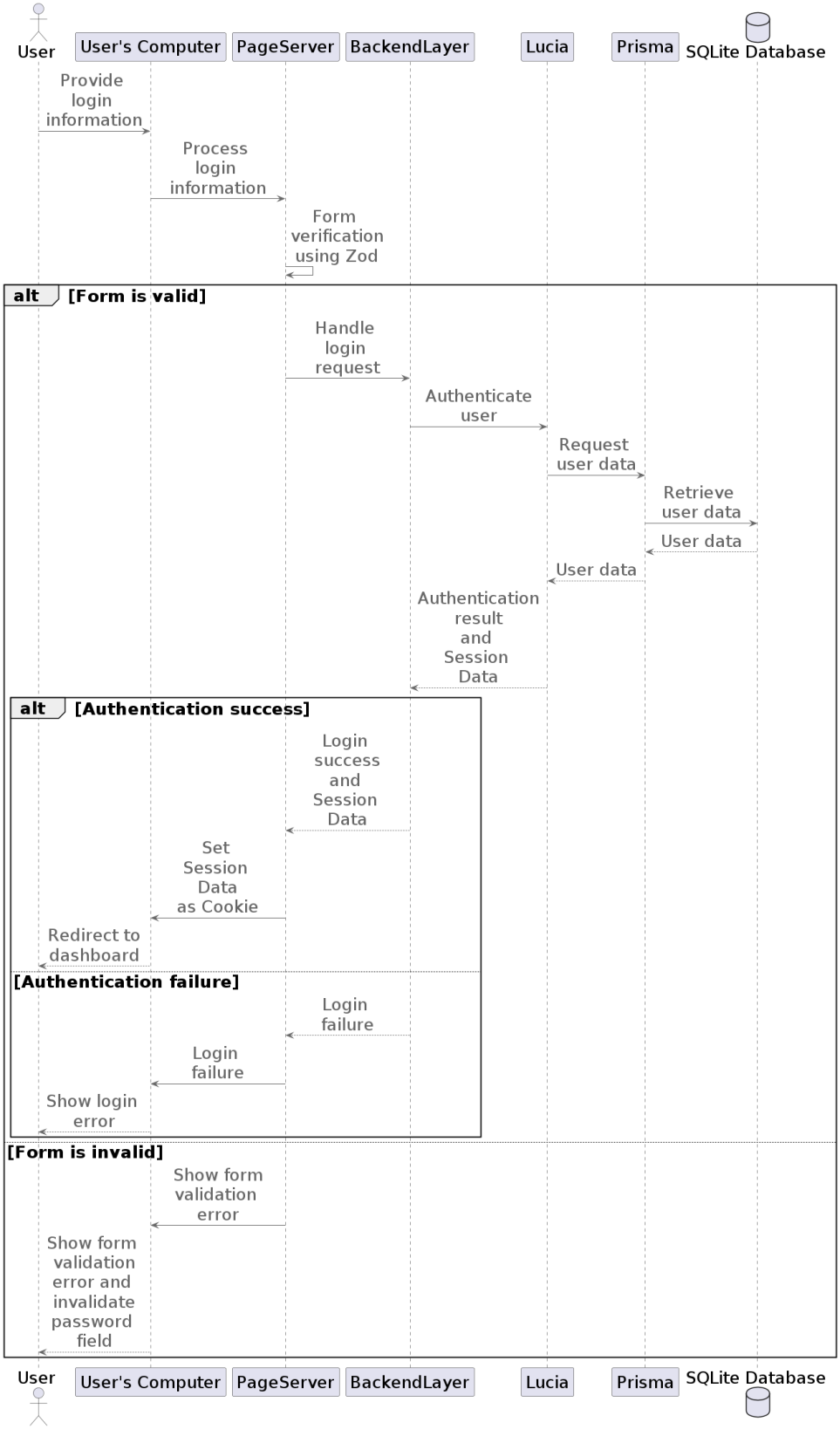


Figure 3.6: Sequence diagram of the login process

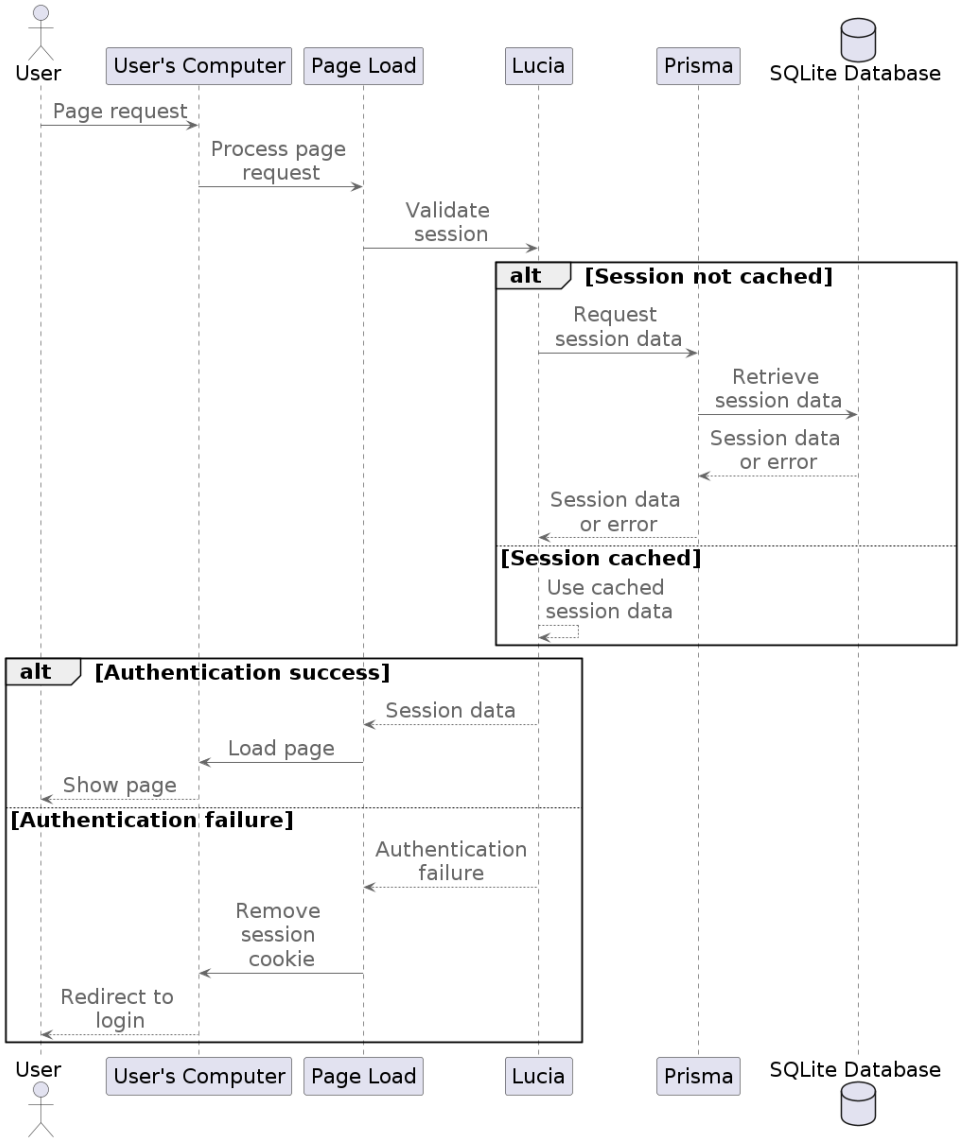


Figure 3.7: Sequence diagram of the page load process

Chapter 4

Implementation

4.1 The attempt at implementation without the need for client-side JS

One of SvelteKit's most interesting features is its ability to work without client-side JavaScript [Sve23b]. While this can be useful in some circumstances, such as for a privacy-focused user or Search Engine Optimization, it is not strictly necessary here because the users who host the application can access the source code and check it manually if they feel it is necessary. Search Engine Optimization also is not a priority because this web app should not be accessible from the broader web if appropriately deployed, so sites like Google will not need to index it.

Although efforts have been made trying to minimize reliance on client-side JavaScript, specific features, like reactive select inputs, require it for optimal functionality. As a result, not all features can be accessed when client-side JavaScript is blocked.

4.2 Front-end

The front end is a crucial aspect of any application, as it directly impacts the user experience. While I acknowledge that I am not a proficient designer, I was able to handle the simpler design tasks. For more complex design requirements, I leveraged the availability of template or component frameworks that have emerged in recent years [YJJ+19]. These frameworks helped facilitate the creation of visually appealing interfaces while ensuring consistency in style throughout the application.

One significant advantage that contributed to maintaining design consistency was the fact that Svelte is a component-based framework. With Svelte, reusable components can be created in separate files and utilized with their associated design, reactivity, and other related code in different components and pages [Sve23a]. The topic of Svelte components and their benefits will be discussed in more detail in the dedicated chapter.

Overall, the focus on the front-end design aimed to enhance the user experience, and while I may not possess advanced design skills, the used frameworks and libraries helped create a visually appealing and consistent user experience.

4.2.1 Tailwind

Using tailwind proved to be a crucial choice that greatly improved the front-end development experience. It is an extremely popular CSS that allows developers to write CSS styles directly rather than relying solely on HTML classes. This approach offers greater control and customization options, enabling the creation of unique and visually appealing designs without having to bother writing custom CSS classes from scratch.

One notable feature of Tailwind is its ability to maintain design and sizing consistency through pre-defined sizes. For instance, using classes like `rounded` and `rounded-xl` translates to specific border-radius values, ensuring a consistent style throughout the application. While seemingly simple, this feature significantly contributes to a cohesive design approach [Lab23].

Last but certainly not least important feature is Tailwind's Responsive

design feature. This allows the developer to easily declare Tailwind classes that only apply after screen width reaches a specific threshold. For example, a class prepended with md: would only apply on screens with a width higher than 768 pixels.

The difference between a piece of code that uses Tailwind (see listing 4.1) and code that does not use Tailwind (see listing 4.2) can be seen by comparing the following two code snippets:

```
<h1 class="flex gap-2 items-center w-full text-xl lg:text-3xl">
    <Icon width="1.5em" height="1.5em" icon="home" />
    Welcome!
</h1>
```

Listing 4.1: HTML with Tailwind

```
<h1 class="my-heading">
    <Icon width="1.5em" height="1.5em" icon="home" />
    Welcome!
</h1>
<style>
    .my-heading {
        display: flex;
        align-items: center;
        width: 100%;
        gap: 0.5rem;
        border-radius: 0.25rem;
        font-size: 1.25rem;
        /* 20px */
        line-height: 1.75rem;
        /* 28px */
    }


    @media (min-width: 1024px) {
        .my-heading {
            font-size: 1.875rem;
            /* 30px */
            line-height: 2.25rem;
            /* 36px */
        }
    }

</style>
```

Listing 4.2: HTML witout Tailwind

4.2.2 DaisyUI

The component library of choice is DaisyUI [Saa23]. The main reason for this choice is that it also uses Tailwind, which integrates well with tailwinds elsewhere.

DaisyUI provides many styles for different components, ranging from simple ones like a button to a whole drawer that dynamically opens and closes based on a hidden checkbox and screen size state.

A few components also have sub-variants, such as a `btn` class that can be further specified as `btn-primary`, `btn-secondary`, `btn-ghost`, or others looking slightly different and having a different purpose while allowing the developer to stay consistent.

The look of these components is decided based on a plethora of provided themes that can be easily switched between by changing only one line of HTML. By default, this only switches between basic dark and light themes, but it was quite easy to implement a settings page that allows the user to switch between different available themes, which are then stored in the user's browser and used every time a new page is rendered.

4.2.3 Input validation

The challenge of complex input validation was addressed with the help of a library called Zod [McD23]. This library allows for the definition of validation schemas that can be run in both the front-end and back-end and allow for parsing user inputs from a form and values from URL parameters. One major advantage of using Zod is that the developer can declare error messages that will be shown when some requirements are not satisfied once centrally and show them to the user in different places when needed. An example of one of these schemas can be seen in listing 4.3.

4.2.4 Svelte components

In Svelte, components are the fundamental building blocks of applications. They are essentially JavaScript files containing HTML, usually CSS, and

```
export const PasswordSchema = z.object({
    password: z
        .string()
        .trim()
        .min(1, { message: "Password is required" })
        .min(8, { message: "Password must be at least 8 characters" })
        .regex(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).+$/, {
            message:
                "Password must contain at least one lowercase letter, one uppercase letter, one number, and one special character.",
        }),
});
```

Listing 4.3: Zod schema for password validation

JavaScript code stored in a `.svelte` file to define a reusable UI piece. The structure of a Svelte component is usually as follows (The reason for the of *usually* is that we don't use the `<script>` tag since this project uses Tailwind). How `.svelte` files are structured can be seen in listing 4.4.

```
<script>
    // JavaScript code goes here
</script>

<!-- HTML code goes here -->

<style>
    /* CSS code goes here */
</style>
```

Listing 4.4: Structure of a Svelte component

This allows each component to have its own scope, meaning that code and styles defined in one component will not affect any other part of the code unless specifically defined.

After creating a component, one can easily use it in another svelte file can be seen in listing 4.5.

Components also allow the use of external variables, changing which would propagate the new value to the parent I received it from and to any children components I passed it through.

```
<script>
    import Component from "./Component.svelte";
</script>

<!-- Some HTML code -->
<Component />
<!-- Rest of HTML code -->
```

Listing 4.5: Structure of a Svelte component

This brings us to the topic of reactivity, which will not be discussed here in great detail since the Svelte documentation explains it well, but the simple version is that any variable defined with `let` in the script block can be changed by any JavaScript action I used to from normal JavaScript+HTML combination. [Sve23a]

An example of a reactive component with a button that shows the number of times it has been clicked can be seen in the code snipped in listing 4.6.

```
<script>
    let count = 0;
</script>

<button
    on:click={() => {
        count += 1;
    }}
>
    Clicked {count}
    {count === 1 ? "time" : "times"}
</button>
```

Listing 4.6: Structure of a Svelte component

4.2.5 Page loading using SvelteKit

In SvelteKit, there are a few steps to rendering a page. While some of these, such as cookie handling and SSR, happen on the server, they should still consider a part of the front-end development since they mostly relate to the user experience. The major elements of developing the front end with SvelteKit that will be highlighted are request handling, routing, layouts, and rendering individual pages. Additionally, an overview of how Svelte handles

server-side rendering will be discussed.

Request handling

SvelteKit handles most of the requests without the need for developer interference but still allows the developer to intercept and work with parts of it. This is done in the `src/hooks.server.ts` file. This file contains the handle method, which runs whenever the server receives a request. This function allows us to read cookies sent in the request, modify the `locals` variable or even modify the response sent to the client or chain multiple functions that modify the response in their own ways.

The main two uses for this file in this project involve handling the site's theming by reading the `theme` cookie stored in the user's browser and modifying the response HTML and injecting Lucia authentication functionality to the `locals` variable, which is a variable accessible from any subsequent part of the page loading process.

Routing

SvelteKit uses a system called a file-based router, which means that the routes users access are mapped directly to the locations where the files for specific pages are stored on the file system. For the sake of explanation, let us say the domain we are using is `page.com`.

The base route is `src/routes`, which maps directly to page.com. Creating a directory called `src/routes/home` would create a route to `page.com/routes`. This is an intuitive way to handle routes, which greatly simplifies the development process.

Of course, SvelteKit also supports more advanced routing, such as hidden routes and slugs, but it is recommended to read the SvelteKit documentation to learn more about those.

Layouts

Layouts are another SvelteKit feature I greatly appreciate. They are a way to create a layout that is applied to every page in a route and its children. For example, a layout file in `src/routes/auth/` would apply to not only `page.com/auth` but also `page.com/auth/login` and `page.com/auth/register`.

These layouts are defined in a `+layout.svelte` file, which can load data from an accompanying `+layout.ts` or `+layout.server.ts` files which load data on the client and server sides, respectively. The great thing about these layout files is that any data that is loaded in them is also accessible from subsequent layout and page files, which allows the developer to load data only once and use it in different parts of the rendering pipeline.

Individual pages

Svelte stores each page in its own route, with each page being defined in a `+page.svelte` file for the design and simpler logic of the page, and `+page.ts` and `+page.server.ts` for data loading and more complex logic. The important part of the `+page.server.ts` file is the fact that none of the code stored in it is ever executed on the client machine, which allows us to write more sensitive code without the need to separate the server-side from client-side code manually.

Server-Side Rendering (SSR)

By default, SvelteKit renders the first load of a page on the server, which speeds up the initial build of the HTML code. After that, any changes to the page are made using a process called hydration which modifies only the changed parts of the page. Combining these two features results in fast initial load times and smaller packet sizes required for page changes and navigation.

State management

SvelteKit offers a streamlined approach to state management, ensuring data accessibility and reactivity across components. The state is often propagated between components via properties, similar to passing variables to functions. This mechanism is suitable for parent-child component relationships where a parent component sends data to a child component through its properties. A useful feature that Svelte offers is two-way bindings on properties using the `bind:` keyword.

However, for complex state management scenarios, SvelteKit leverages stores. These stores are essentially singletons that hold the application state. A store is an object with a subscribe method that updates components whenever the store's value changes. This approach ensures that state data is available globally and synchronizes changes across all listening components.

In addition to global stores, SvelteKit provides context stores for more granular state management. Context stores allow components to create a context and set values. Any nested child components can retrieve and use those retrieve or set values using its key. This is particularly useful when components deeply nested in the hierarchy need to access data without passing it down through multiple layers of components.

4.3 Back-end

The back-end is the second major part of any software project, as it is the engine that powers all the front-end interactions and ensures the seamless functioning of the software. Despite not having a strong background in front-end design, my experience with back-end is much greater, which greatly reduced the work required to complete this part of the project compared to the front-end.

SvelteKit, a framework rapidly rising in popularity, was deployed to manage routing and communication between the front-end and back-end. The usage of SvelteKit has greatly sped up the process of creating dynamic, server-rendered pages and managing routes and endpoints. The usage of such a framework not only simplified the back-end building process but also maintained a consistent approach to the application's development.

4.3.1 Prisma

Probably the most important back-end library is Prisma, a multi-language ORM (Object-Relational Mapping) library that allows the developer to define a schema based on which a whole database structure is generated. When this structure is created, Prisma also invokes one of its generators which can generate type-safe objects for the desired language, which was TypeScript in the case of this project [PD23a].

This generated client allows the developer to directly use types generated based on the database model in the project's code without having to define it manually. The generated structure also allows for creating queries in a format much more like standard JavaScript compared to SQL.

An example of a part of the project's database schema to illustrate how Prisma schema files are structured can be seen in listing 4.7.

```
1  model Wallet {
2      id String @id @default(cuid())
3      name       String
4      users      AuthUser[]
5      currency   String
6      categories Category[]
7      payments   Payment[]
8
9      @@map("wallet")
10 }
11
12 model Category {
13     id String @id @default(cuid())
14     name  String
15     color String
16     wallet   Wallet @relation(fields: [walletId], ↩
           references: [id], onDelete: Cascade)
17     walletId String
18     payments Payment[]
19
20     @@unique([walletId, name])
21     @@map("category")
22 }
```

Listing 4.7: Prisma schema example

An example of a TypeScript code containing a Prisma query that disassociates a user from a wallet can be seen in listing 4.8.

```
const wallet = await prisma.wallet.update({
    where: {
        id: walletId,
    },
    data: {
        users: {
            disconnect: [{ id: userId }],
        },
    },
});
```

Listing 4.8: Prisma query example

A SQL query that would achieve the same result as listing 4.8 can be found in listing 4.9.

```
UPDATE wallet
SET users_id = NULL
WHERE id = < walletId >
    AND users_id = < userId >;
```

Listing 4.9: SQL query example

Another feature of Prisma I appreciated is Prisma Studio [PD23b], which provides a web-based GUI for managing and exploring the data in the database. Prisma Studio enables the monitoring of data in the database in real-time, which is extremely useful while debugging query problems and checking if queries were executed properly.

4.3.2 Lucia

Lucia is a modern, simple, and flexible library designed for managing users and sessions. It helps abstract the relationship between the application that the user and the session management side of the database. It also provides an easy-to-use function for handling cookies, creating, verifying, refreshing sessions, and other quality-of-life features. [Pil23]

While it is quite bare-bones compared to some alternatives, it does everything a smaller project needs without unnecessary bloat or complications. This simplicity both improves the speed and simplifies the required setup.

It also has middleware made specifically for SvelteKit that hooks up to the `locals` variable mentioned in the Front-end section, and it can directly hook up to a Prisma client if the database schema follows certain rules defined in the Lucia documentation.

One issue encountered with this library was that version 1.0 came out while this project was already in development, and the migration required a few major breaking changes, such as renaming parts of the database schema and re-implementing the SvelteKit adapter. This problem is acceptable, as the documentation stated that the library was still in beta when it was first added to the project. [Pil23]

4.3.3 Rest of the back-end

The back-end also contains many utility functions, custom error types, and a pre-made list of currencies and functions relating to them.

4.4 Deployment

A well-made self-hosted application also requires a user-friendly deployment method. While SvelteKit offers different adapters for building applications for production, such as static adapters, which build the project into a static site, and platform-specific adapters, like an adapter for Vercel or Cloudflare pages that work natively on those specific platforms, opting for the Node adapter enables running and hosting the application using NodeJS which is ideal for a self-hosted application.

4.4.1 Dockerization

These days, most self-hosted applications offer a docker image or a pre-made way to build the docker container that would run the application without the need to install any prerequisite software or libraries directly on the host machine. [Inc23]

A Dockerfile has been created to build the required environment to run the

server. It copies the code and configuration files from the host filesystem and installs `pnpm`, which it then uses to install all required JavaScript libraries. It also creates a data folder the user can mount to the host filesystem when running the container. This folder is then used to store the SQLite database file, which can easily be backed up or moved to another machine. After that, it sets up all necessary Environment variables to run the file and prepares the script that compiles and runs the program.

This separate script is necessary because the generated Prisma client relies on an existing database. The script contains four main steps. The first creates a database if the user does not already have one, the second generates the Prisma client, the third builds the application, and the last runs it.

4.4.2 How to run an instance

1. Install docker. The instructions on how to install it on a specific OS can be found here: `https://docs.docker.com/get-docker/`

2. Open the folder containing the code in a terminal.

3. Run `docker build -t pfm_image .`, or replace `pfm_image` with desired image name.

4. Run `docker run -d -v path/to/data:/data -p 3000:4173 -name personal-finance-manager pfm_image` while replacing `path/to/data` with the path to the directory you want to store your data in and replacing `pfm_image` with the name of the docker image you created in step 3.

   - If one wants to change the port, replace the first 3000 with the desired port.
   - If one wants to change the name of the container, replace personal-finance-manager with the desired name.

5. The server should now be running on `localhost:3000` (or the port selected in step 4). The default login is admin, and the default password is Admin123. It is recommended to change this immediately, or to create a new account, set it as an administrator, and delete the default account.

4.5 Security

Security is a crucial part of any application, especially one that handles sensitive data like personal finance management. This section will cover the security measures taken to protect the application from unauthorized access and common attacks.

4.5.1 Authorization

The standard authentication-based security is managed by intercepting all requests to the server using a hook, checking if the requested URL is for a page other than /register or /login, and then utilizing Lucia to verify the session cookie from the request, allowing only requests from users with valid sessions. Using a hook stops the request before reaching any load function that could cause problems. This both prevents an unauthorized user from accessing any page other than the login and register pages, and also prevents data submission from unauthorized users.

4.5.2 Protection against common attacks

There is a series of common attacks every website should protect against; a few of the most common ones are SQL injections, cross-site scripting, password guessing, denial of service attacks, and many others [BNK+23].

Password guessing. Brute force attacks like password hashing are handled by Lucia since it handles everything relating to authentication, which includes hashing passwords. Lucia uses the `scrypt` hash from the noble-hashes library to hash passwords [Pil23]. `Scrypt` is a memory-hard hashing algorithm which means that it requires a large amount of memory to compute [CMBH19]. Noble-hashes has also been audited by a professional security firm which raises confidence in this encryption not being broken [pau23].

SQL Injection. SQL injection is a technique where an attacker passes a SQL query into an input field that later gets passed into an actual database

query and executed [BNK+23]. Luckily, Prisma automatically sanitizes all input, so we do not have to worry about this kind of attack.

Input validation. Even though non-SQL-injection attacks relating to input validation do not pose as much threat as SQL injections, validating the user input on the server side is still important since a malicious user can always modify the page's HTML to submit invalid data, potentially disrupting the website. This vulnerability is solved by using Zod schemas to validate submitted data both on the client side when the client clicks the Submit button and then on the server side when the request is received. Since we also use Zod to parse the received data from the request, it ensures that all user data we use on the back end is always validated.

Encryption and DDOS. As this is a self-hosted application, the responsibility of properly encrypting the communication with the server falls to the user who deployed the application. This is usually handled by hiding the service behind a VPN or using a reverse proxy.

While it would be possible to force the end user to use only HTTPS, it would limit the user's choice of how the application is deployed.

The same applies to protection against targeted attacks such as a Denial-of-Service attack. Since the end-user handles the deployment and hosting of the application, the application's development phase can't offer much in terms of preventive measures. A common protection against these kinds of attacks is hiding the app behind a Cloudflare DNS with DDOS protection enabled.

One major advantage of a self-hosted app is its lower likelihood of being targeted, given that it usually only serves a limited number of users, typically in the single or double digits.

4.6 Logging

Logging is the last part of the implementation process I have yet to mention. Logging refers to the practice of recording events, warning of possible problems and full-on errors with relevant details such as the error level, the timestamp of

the error occurring, and an error message. This enables the use of monitoring and troubleshooting in the system. By capturing and storing log data, developers and administrators can gain insights into system behavior, identify potential issues, and analyze patterns.

Logging serves multiple purposes in software development and system maintenance. It helps track the flow of execution, provides a historical record of events, and aids in debugging and diagnosing problems. By including relevant details such as error levels, timestamps, and error messages, logging enables efficient error detection and resolution.

It can also help detect security issues by analyzing logs and checking for suspicious behavior. While this may not be necessary for a self-hosted application like this one since it is a small target, it is always good to consider.

4.6.1 Pino

The logging library I have elected to use is Pino. It is one of the many logging frameworks available for Node.js, and the reason I have chosen it is its speed and easy setup. It includes all the features one may need from a logging library, such as customizable log levels and multiple transport layers allowing different types of output, such as outputting to console, to a file, or even to something like AWS Elasticsearch.

The one I have elected to use is provided by a library called `pino-pretty`, which greatly simplifies the formatting of error messages. A simple setup shown in listing 4.10 allows for a sufficient logging setup with minimal effort.

```
1  import pino from "pino";
2
3  export const logger = pino({
4      transport: {
5          target: "pino-pretty",
6          options: {
7              colorize: true,
8              translateTime: true,
9              ignore: "pid,hostname",
10             levelFirst: true,
11         },
12     },
13 }) as pino.Logger;
```

Listing 4.10: Pino logger setup

Chapter 5

Testing

Creating a web app entails various stages, such as designing, programming, and launching the app. However, one crucial and frequently overlooked element of the development journey is testing. Verifying the web app's functionality and dependability is essential, as it directly influences user experience. In this section, we will delve into two testing methods for web app development: Unit Testing and End-to-End testing. Since they serve distinct purposes, both will be discussed in individual sections.

5.1 Unit testing

Unit Testing focuses on testing the application at the method level. Unit tests are usually quite short and should be performed quickly and often. This form of testing enables developers to identify and resolve issues at the code level before they turn into more significant problems. It also facilitates the maintenance and refactoring of code, since it instantly reveals any issues created created by the change as soon as the file is saved.

5.1.1 Vitest

Vitest is a modern and fast JavaScript testing framework designed specifically for Vite, providing an efficient and streamlined testing pipeline for web appli-

cations developed using TypeScript [Cap23]. The main reason I chose Vitest is the fact that SvelteKit projects come with a complete Vite configuration file, which Vitest can use to simplify setup, such as configuring test files, and helping prevent problems with things specific to SvelteKit such as aliasing `$lib/...` to `/src/lib/...` to prevent messy relative paths.

Another great feature of Vitest is its integration with Visual Studio Code, which offers a watch mode that significantly simplifies catching errors during development. This capability leads to quicker error resolution and increased productivity, allowing developers to focus on refining their code and testing strategies.

5.1.2 Vitest-Mock-Extended

Another library that helped with unit testing is vitest-mock-extended, a fork of a popular library called jest-mock-extended that was modified to work better with Vitest.

While unmodified Vitest allows for mocking objects, vitest-mock-extended makes the process significantly easier and more intuitive. Vitest-mock-extended simplifies the process of creating type-safe mocks for interfaces, argument types, and return types, allowing developers to test their web applications with greater confidence. Moreover, vitest-mock-extended offers features such as adding a `.calledWith()` extension to mocked objects and functions for argument-specific expectations and deep mocking capabilities.

5.1.3 Unit testing in this application

In this application, the focus of unit testing is mainly on acceptance testing of utility functions and functions that do not have side effects. Acceptance testing focuses on testing if the tested functions return the expected values. Examples of tested functions include Zod schemas which are used to parse user inputs, functions related to currency processing, and utility functions for modifying URLs and dates. This can be seen in listing 5.1.

The reason behind this approach is that unit tests are meant to test isolated pieces of code, and isolating SvelteKit's pages for testing poses a challenge. This is because they rely on the framework and the browser, and are not

designed to operate outside of a browser environment. This means that unit tests for pages would have to be written in a way that would mock the browser and the framework, which would be a lot of work for little gain, since this application also has end-to-end tests that test the pages in a browser-like environment.

Prisma models face the same challenges, as almost all problems they could cause happen with the database connection or a badly written query, which would not be detected by most unit tests as these mocked tests usually mock the return value of the query rather than the query itself. While there are some unit tests that do test the models, they are mostly there to test the general structure of the functions that utilize the models.

Fortunately, both the SvelteKit code and database-related code are covered by end-to-end tests, explored further in the following section.

```
describe("formatCurrency", () => {
    test("formats the amount with the currency symbol and decimal digits", () => {
        const currency: Currency = {
            symbol: "$",
            decimal_digits: 2,
        } as Currency;
        const amount = 1234.56789;
        const formatted = formatCurrency(amount, currency);
        expect(formatted).toBe("$1234.57");
    });

    test("Format with weird currency symbol and name", () => {
        const currency = {
            symbol: "~",
            decimal_digits: 3,
        } as Currency;
        const amount = 1234.56789;
        const formatted = formatCurrency(amount, currency);
        expect(formatted).toBe("~1234.568");
    });
});
```

Listing 5.1: Unit tests for the `formatCurrency` function

5.2 End to End testing

End to End testing, or E2E testing for short, is the polar opposite of unit testing as it covers the System level of testing. Whereas unit testing aims to test the smallest possible portions of code, such as individual functions, E2E testing focuses on testing large chunks of the project in a production-like environment. This effectively tests everything from the front-end styling, page rendering, and server back-end to the fact that everything works properly when used with an actual database.

5.2.1 Playwright

For this project, the end-to-end testing framework is Playwright. Playwright is a modern alternative to more traditional frameworks like Selenium, offering a more straightforward and more efficient approach to E2E testing [Mic23b]. Developed by Microsoft, Playwright is a Node.js library that enables developers to automate browser actions, facilitating the testing of web applications across multiple browsers, including Chromium, Firefox, WebKit, and their mobile versions with both headed and headless modes.

Some of the features that separate Playwright from similar frameworks such as Selenium:

- More user-friendly syntax: Playwright's API is designed to be more intuitive and easier to use than Selenium. The syntax is clear and concise, making it simple to understand and implement test scenarios. This simpler syntax reduces the learning curve for developers and allows them to write tests more efficiently [Kha22].
- The lack of implicit waits: Playwright automatically waits for elements to be available, visible, or stable before performing actions, eliminating the need for manual waits. This results in more reliable, faster tests, reduces flakiness, ensures that elements are ready before actions are performed, and decreases the chances of timing-related issues.
- Input recording and Locator finding: Playwright has a built-in feature to record user interactions, automatically generating test scripts. Although this feature sometimes records less optimal locators, such as 'First label on the page' rather than 'Label with text "ABC",' it still provides a valuable starting point for writing tests. Developers can refine the

generated scripts and locators to improve their accuracy and reliability, resulting in faster test creation.

How playwright tests work

Playwright tests simulate user interactions with the web application in a controlled environment. These tests are designed to mimic real-world user experiences and validate the application's functionality across various browsers and platforms. They start by initializing a browser instance, then navigating to the desired URL and performing user actions such as clicking buttons, filling out forms, and navigating between pages. Throughout the testing process, assertions are made to verify the expected behavior, ensuring that the application responds accordingly to the simulated user inputs.

Playwright's robust API enables developers to handle complex interactions, such as dealing with asynchronous operations, handling network requests, and managing browser contexts. By automating these browser actions and validating their outcomes, Playwright tests provide comprehensive coverage of the application's functionality, ultimately contributing to a reliable and high-quality user experience.

Changes to code

A few changes needed to be made to the codebase to ensure that end-to-end tests can be performed in this project. One of the major changes required involved the addition of specific checks that allows tests to ignore the invitation key used for user registration while the environment variable `NODE_ENV` is set to `'test'`. This check assures the project is running in test mode, so the skip is not available in production without having to remove the relevant code manually.

Problems with these E2E tests

While there were several problems while working with Playwright, primarily stemming from the fact that Playwright is still quite new compared to other

testing frameworks, these problems resulted in longer time spent solving those problems than is generally necessary.

One of the encountered problems is that these tests are too fast for a combination of SvelteKit with an SQLite database. While the database speeds are sufficient for a self-hosted project like this one, it is still intended for a single or double-digit number of users per instance, with even lower number of concurrent users. The expectation for peak usage by humans are approximitely five writes and ten reads per second, and even that is severely overestimating the peak time usage.

Meanwhile, Playwright tests can easily fill out multiple forms per second, which results in a database write followed by a redirect which can require multiple database queries for each. Playwright also supports running tests in parallel, which can result in dozens of database writes per second, and the measured operations reached over 300 reads per second with eight Playwright tests running simultaneously.

Although this this will not cause problems with the day-to-day usage of this application, it can still cause slower queries which, combined with SvelteKit's partial SSR, can lead to Playwright trying to perform the following steps before the page has been fully submitted and hydrated, thus leading to flakiness. While this problem has been mostly alleviated by adding retries to some operations where the tests often failed, some flakiness remains, as shown in figure 5.1, so adding retries to the test configuration was required.

The other major issue was that Playwright could not find test files that imported functions from other files using Implicit File Extension Resolution (Implicit file extension resolution in TypeScript allows omitting the file extension when importing modules, and TypeScript automatically looks for the appropriate file extension such as .js or .ts in import statements).

While this issue seems minor at first glance, the fact that it simply acts as if the tests that use it do not exist, this problem is not being mentioned anywhere online, and the fix required changing the Vite configuration file resulted in a whole evening spent on this minor issue [You19a].

```
✗  1 [chromium] › auth.test.ts:16:1 › register, login, logout (12.0s)
✓  2 [firefox] › auth.test.ts:16:1 › register, login, logout (3.9s)
✓  3 [chromium] › wallet.test.ts:21:1 › Wallet creation and filtering (13.0s)
✓  4 [firefox] › wallet.test.ts:21:1 › Wallet creation and filtering (28.9s)
✓  5 [firefox] › auth.test.ts:32:1 › register and log in input validation (7.1s)
✓  6 [chromium] › auth.test.ts:16:1 › register, login, logout (retry #1) (1.8s)
✓  7 [webkit] › auth.test.ts:16:1 › register, login, logout (5.0s)
✓  8 [chromium] › wallet.test.ts:107:1 › multi-account wallets (8.8s)
✓  9 [chromium] › auth.test.ts:32:1 › register and log in input validation (6.6s)
✓  10 [webkit] › auth.test.ts:32:1 › register and log in input validation (10.0s)
✓  11 [webkit] › wallet.test.ts:21:1 › Wallet creation and filtering (21.6s)
✓  12 [Mobile Chrome] › auth.test.ts:16:1 › register, login, logout (3.3s)
✓  13 [Mobile Chrome] › auth.test.ts:32:1 › register and log in input validation (7.3s)
✓  14 [Mobile Chrome] › wallet.test.ts:21:1 › Wallet creation and filtering (14.2s)
✓  15 [firefox] › wallet.test.ts:107:1 › multi-account wallets (13.9s)
✓  16 [Mobile Safari] › auth.test.ts:16:1 › register, login, logout (3.3s)
✓  17 [Mobile Safari] › auth.test.ts:32:1 › register and log in input validation (8.5s)
✓  18 [Mobile Chrome] › wallet.test.ts:107:1 › multi-account wallets (11.8s)
✓  19 [webkit] › wallet.test.ts:107:1 › multi-account wallets (13.8s)
✓  20 [Mobile Safari] › wallet.test.ts:21:1 › Wallet creation and filtering (22.3s)
✓  21 [Google Chrome] › auth.test.ts:16:1 › register, login, logout (2.7s)
✓  22 [Google Chrome] › auth.test.ts:32:1 › register and log in input validation (6.1s)
✓  23 [Google Chrome] › wallet.test.ts:21:1 › Wallet creation and filtering (12.4s)
✓  24 [Microsoft Edge] › auth.test.ts:16:1 › register, login, logout (2.3s)
✗  25 [Microsoft Edge] › wallet.test.ts:21:1 › Wallet creation and filtering (30.0s)
✓  26 [Microsoft Edge] › auth.test.ts:32:1 › register and log in input validation (6.0s)
✗  27 [Mobile Safari] › wallet.test.ts:107:1 › multi-account wallets (13.5s)
✓  28 [Google Chrome] › wallet.test.ts:107:1 › multi-account wallets (7.6s)
✓  29 [Mobile Safari] › wallet.test.ts:107:1 › multi-account wallets (retry #1) (14.1s)
✓  30 …osoft Edge] › wallet.test.ts:21:1 › Wallet creation and filtering (retry #1) (10.9s)
✓  31 [Microsoft Edge] › wallet.test.ts:107:1 › multi-account wallets (7.6s)
```

Figure 5.1: Flaky test results

Conclusion

In conclusion, Playwright is a powerful and efficient E2E testing framework, offering significant advantages over traditional tools like Selenium. Its user-friendly syntax, automatic waiting mechanism, and input recording capabilities contribute to a more streamlined testing process, ultimately enhancing the quality and reliability of web applications. While there were some challenges as mentioned above, the process of writing Playwright tests is a much more pleasant experience when compared to other testing frameworks like Selenium.

5.2.2 Test scenarios

To test this project, we have created four individual end-to-end test scenarios for different parts of the program. This document contains the summaries of each scenario, with steps and expected results available in the `scenarios.md` markdown file, which can be found in the analysis folder bundled with this thesis.

While these scenarios are not comprehensive, they cover the most important parts of the project, such as user authentication, user-to-user interactions, wallet management, and input validation.

5.2.3 Basic Authentication Test Scenario

This scenario covers the essential user authentication process, including user registration, login, and logout. By testing these functionalities, we ensure that users can successfully register, log in with their credentials, and securely log out of the platform.

5.2.4 Input Validation Test Scenario

This scenario targets the registration and login forms' input validation and back-end handling of credentials. By testing various input scenarios and back-end handling, we ensure that the forms are properly validated for different inputs and that the back-end responds correctly to both valid and invalid credentials.

5.2.5 Wallet Management Test Scenario

This targets wallet functionality - creating a new wallet, adding categories and transactions and sorting and applying filters to transactions. By testing these features, we ensure that the wallet is created with the correct name and currency, categories and transactions are added accurately, pagination works as intended, and filtering and sorting operations provide the desired results.

5.2.6 Multi-Account Wallet Test Scenario

This scenario focuses on the shared wallet functionality, simulating a multi-user scenario where a wallet is created and shared between two users. By testing the registration, wallet creation, user addition, and transaction management, we ensure that both users are capable of working with the shared wallet seamlessly as would be expected.

Chapter 6

Conclusion

The original aim was to create a simple self-hosted web app that allows users to track their finances, and I believe I have achieved this goal.

To begin, I compiled a list of functional and non-functional requirements and created a list of possible use cases. This process has given me a general idea of the type of application I will be developing and what steps I will need to take to achieve this.

After that, I performed a more in-depth analysis of what tools should be used when developing this kind of application. I have considered if it should be a mobile, desktop, or web application, and I ended up going with web applications for their ability to be accessed on any device. After that, I chose between different JavaScript frameworks and selected Svelte. Thi decision directly lea to me chosing SvelteKit as a back-end framework. I have also created a list of other tools that can simplify the development process.

The next step was planning the general architecture of the program, which included planning the program structure, back-end, and front-end, planning how they would communicate, and creating diagrams to plan the structure and different processes.

The following step was the most important one – creating the software itself. When working on it, I created both the front and back end simultaneously since SvelteKit ties the front and back end of a single page close together than two different pages. Although there were some problems along the way,

I believe the final result works well and looks good.

I have also tested the code during the development using unit tests and after the completion using E2E testing. Both testing methods have been instrumental in catching and ironing out the bugs I have not caught during development in different ways. Overall, both unit and E2E testing have proven invaluable in improving the quality of the final product.

I would also like to mention that even though some problems have been caused by using some very new frameworks and libraries, I do not regret using them as they have significantly improved some processes that would have been more complicated when using older methods.

6.1 Potential future improvements

While this project works as intended, there are always areas that could be improved. The first area of possible improvement is the front end. While it is usable and not-bad looking right now, I cannot say I am a talented UI designer, so external help from an experienced UI designer may be necessary. Moreover, while DaisyUI works well as a starting template, some hard-coded details could be changed to fit the app better. Based on the DaisyUI 3.0 GitHub repository, it seems like most of the problems have been addressed, but the version remains in alpha as of the writing of this thesis.

The second area that could be improved is deployment. The obvious next step for easier deployment is publishing a docker image to Docker Hub and providing a docker-compose file for easier setup. This could also neatly combine with the next possible improvement: creating a Continuous Integration pipeline that automatically builds and publishes docker images when a new version is pushed to the Main/Release branch on GitHub and passes all the unit and E2E tests.

If better scalability is needed in the future, the obvious improvement would be allowing the use of other databases like Postgres since their performance is significantly better than the SQLite database we use currently. This could also help addressed the issue of flaky tests caused by them being too fast.

Appendix A

Bibliography

[And22] Henry Andersson, *A comparison of the performance of an android application developed in native and cross-platform: Using the native android sdk and flutter*, 2022.

[Bal21] Jatinder Singh Bal, *Development of native mobile application*, Journal of Critical Reviews (2021).

[BJV+22] B Balatamoghna, Aditya Jaganath, S Vaideeshwaran, Anish Subramanian, and K Suganthi, *Integrated balancing approach for hosting services with optimal efficiency-self hosting with docker*, Materials Today: Proceedings **62** (2022), 4612–4619.

[BNK+23] Bhuvana Reddy Bhimireddy, Alekhya Nimmagadda, Harshith Kurapati, Leelendra Reddy Gogula, Radhika Rani Chintala, and Vijaya Chandra Jadala, *Web security and web application security: Attacks and prevention*, 2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS), vol. 1, IEEE, 2023, pp. 2095–2096.

[BPP15] ST Bhosale, Tejaswini Patil, and Pooja Patil, *Sqlite: Light database system*, Int. J. Comput. Sci. Mob. Comput **44** (2015), no. 4, 882–885.

[Cap23] Matías Capeletto, *Vitest*, 2023, Accessed: 2023-05-18.

[Che22] Cheng Chen, *Multi-perspective evaluation of relational and graph databases*, Master's thesis, University of Helsinki, 2022.

[CMBH19] Jiwon Choe, Tali Moreshet, R Iris Bahar, and Maurice Herlihy, *Attacking memory-hard scrypt with near-data-processing*, Proceed-

ings of the International Symposium on Memory Systems, 2019, pp. 33–37.

[CMDP16] Theo Combe, Antony Martin, and Roberto Di Pietro, *To docker or not to docker: A security perspective*, IEEE Cloud Computing **3** (2016), no. 5, 54–62.

[DP08] David Dearman and Jeffery S Pierce, *It's on my other computer! computing with multiple devices*, Proceedings of the SIGCHI Conference on Human factors in Computing Systems, 2008, pp. 767–776.

[Fir23] Firebase, *Firebase*, 2023, (accessed on May 20, 2023).

[GW05] William Bradley Glisson and Ray Welland, *Web development evolution: The assimilation of web engineering security*, Third Latin American Web Congress (LA-WEB'2005), IEEE, 2005, pp. 5–pp.

[Har19] Rich Harris, *Write less code*, 4 2019.

[Inc23] Docker Inc., *Docker*, 2023, Accessed: 2023-05-18.

[Joh20] Jeff Johnson, *Designing with the mind in mind: simple guide to understanding user interface design guidelines*, Morgan Kaufmann, 2020.

[Kha22] Arsalan Khalid, *Challenges with automated software testing*, Master's thesis, University of Oslo, 2022.

[Lab23] Tailwind Labs, *Tailwind CSS*, 2023, Accessed: 2023-05-18.

[McD23] Colin McDonnell, *Zod*, 2023, Accessed: 2023-05-18.

[Mic23a] Microsoft, *JavaScript With Syntax For Types.*, 2023, Accessed: 2023-05-18.

[Mic23b] ———, *Playwright*, 2023, Accessed: 2023-05-18.

[pau23] paulmillr, *noble-hashes: Audited & minimal js implementation of hash functions, macs & kdfs*, 2023, Commit dated May 20, 2023.

[PD23a] Inc. Prisma Data, *Prisma Documentation | Concepts, Guides, and Reference*, 2023, Accessed: 2023-05-18.

[PD23b] ———, *Prisma studio*, 2023, Accessed: 2023-05-18.

[Pil23] Pilcrow, *Lucia*, 2023, Accessed: 2023-05-18.

[REC+11] Christopher Riederer, Vijay Erramilli, Augustin Chaintreau, Balachander Krishnamurthy, and Pablo Rodriguez, *For sale: your data: by: you*, Proceedings of the 10th ACM WORKSHOP on Hot Topics in Networks, 2011, pp. 1–6.

[SA20] Gian Luca Scoccia and Marco Autili, *Web frameworks for desktop apps: an exploratory study*, Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020, pp. 1–6.

[Saa23] Pouya Saadeghi, *daisyUI*, 2023, Accessed: 2023-05-18.

[SG] Raphaël Benitte Sacha Greif, *State of JavaScript 2022: Front-end Frameworks*, Accessed: 2023-05-18.

[Sve23a] SvelteKit Foundation, *Sveltekit documentation*, 2023, Accessed: 2023-05-18.

[Sve23b] _____, *Sveltekit documentation - page options*, 2023, Accessed: 2023-05-18.

[tea19] Flutter team, *Flutter - beautiful native apps in record time*, 2019.

[Tea23] Dart Team, *Dart programming language overview*, 2023, Accessed: 2023-05-20.

[Wan23] Yichen Wang, *Inflation surge: Impact of covid-19 pandemic and ukraine conflict*, Highlights in Business, Economics and Management **10** (2023), 393–397.

[Xu21] Wenqing Xu, *Benchmark comparison of javascript frameworks react, vue, angular and svelte*, Hämtad den **5** (2021).

[YJJ+19] Enes Yigitbas, Klementina Josifovska, Ivan Jovanovikj, Ferhat Kalinci, Anthony Anjorin, and Gregor Engels, *Component-based development of adaptive user interfaces*, Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2019, pp. 1–7.

[You19a] Evan You, *Vite*, 2019.

[You19b] _____, *Vue.js*, 2019.

[ZFL+19] Agustin Zuniga, Huber Flores, Eemil Lagerspetz, Petteri Nurmi, Sasu Tarkoma, Pan Hui, and Jukka Manner, *Tortoise or hare? quantifying the effects of performance on mobile app retention*, The World Wide Web Conference, 2019, pp. 2517–2528.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Krul Filip** Personal ID number: **499329**

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:

Web application for management of personal finances

Bachelor’s thesis title in Czech:

Webová aplikace na správu osobních financí

Guidelines:

Bibliography / sources:

1] Pressman, R. S. (1987). Making software engineering happen: A guide for instituting the technology. Prentice-Hall, Inc..
[2] Xu, Wenqing. "Benchmark Comparison of JavaScript Frameworks React, Vue, Angular and Svelte." Hämtad den 5 (2021).
[3] Haque, Mubin Ul, Leonardo Horn Iwaya, and M. Ali Babar. "Challenges in docker development: A large-scale study using stack overflow." Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2020.

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: **21.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **16.02.2025**

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