# Mobile App & Backend Integration for Streamlined Library Operations: Approach with Cutting-Edge Technology

**Richard Zilahi**

**Laurea University of Applied Sciences**

# Mobile App & Backend Integration for Streamlined Library Operations: Approach with Cutting-Edge Technology

Richard Zilahi
Business Information Technology
Thesis
November 2023

---

This thesis details the development and implementation of a cutting-edge mobile application using TypeScript and cloud technologies. Employing an agile development methodology complemented by user-centered design principles, the study aimed to create an intuitive and user-friendly mobile app. The research journey encompassed several stages, including requirements gathering, UX research, UI design, back-end development, and cross-platform app implementation, all dedicated to delivering a seamless user experience.

The initial phase involved conducting user research to gather insights, learn about the target user groups, and establishing comprehensive user needs and preferences. These insights were instrumental in crafting a robust UX design plan that aligned the app's interface with user expectations. Iterative design and prototyping identified and addressed usability and interaction challenges, refining the user experience.

Leveraging React Native and Expo's latest SDK, the app's implementation ensured robustness and scalability, enhancing performance, security, and extensibility. Employing Test Driven Development facilitated iterative development cycles, allowing continuous improvement and flexibility in adapting to evolving requirements. Regular feedback loops with stakeholders and users further refined the app's functionality.

Contents

1 Introduction

In today's rapidly evolving digital landscape, mobile applications have become an integral part of our daily lives, providing convenience, connectivity, and entertainment. The development of mobile apps requires a detailed approach, utilizing the latest technologies and methodologies to ensure a seamless user experience. This thesis explores the creation of a mobile application for a free, community driven library in Satakunta, Finland, employing the most modern JavaScript and TypeScript technologies, coupled with an agile development methodology and a user-centered design framework. By leveraging these modern tools and techniques, the study aims to demonstrate the effectiveness of this approach in delivering a highly functional and user-friendly mobile application. Through the integration of interviews, agile development methods, UX design principles, and cutting-edge technologies, this thesis project provides valuable insights into the development process of modern mobile apps, and a working usable cross-platform mobile application.

1.1   Choosing the case

The subject of my thesis work is an operating community driven library, which is located in Satakunta, Finland. I have been participating in the creation of this library. The library is a free library, which is open to everyone. While the library is still in its infancy, it has already gained a lot of popularity in the local community. Since the library is basically a charity project, it has no funding, therefore it soon became clear that the library maintenance work will require some kind of automation, and as well as a log or registry of the books that are being borrowed. While the main focus was on helping the staff personnel of the library, the idea of helping at the same time for the users of the library was also considered. The idea of a mobile application, together with a set of administrative featured was born.

1.2   Defining of the problem

The problem that this thesis work aims to solve it to provide a comprehensive end-to-end solution for a free-to-use community library. The day-to-day operations of the library are currently not automated, there's no log or registry of the books that are being borrowed, and the administrative personnel of the library has no way to keep track of the books. Additionally, there is no framework provided for the visitors of the library to donate their own books, due to the lack of database of what books are available in the library. This is a complex problem, which requires a comprehensive solution, which satisfies the needs of both the administrative personnel of the library, and as well as the visitors of the library. Apart from the technical challenges of the implementation, the whole architecture stack must be easy to maintain, develop further, and do to the free community drive nature of this library, the entire application stack must be very cheap to run.

1.3   Thesis structure

The thesis has two main focus idea: UX / UI related work, and the technical implementation of the mobile application and the backend.

The UX and UI work is discussed in the first part of the thesis. The UX work conducts hypothesis analyses, interviews, user research by personas, and user stories. Providing solutions to the related problems, and creating prototypes throughout multiple iteration of the

mobile application.

The technical implementation of the mobile application and the backend is discussed in the second part of the thesis. This part of the thesis will discuss in great details the development process, the architectural decisions, and codebase and the implementation in general.

The closing part of the thesis will discuss the results of the thesis work, and the potential enhancements that can be done in order to improve the application stack, and the user experience.

2   User Experience Research

The primary objective of this UX research is to gain a comprehensive understanding of the needs, preferences, and challenges faced by users when using the community-driven library. My aim is to ensure that the application aligns with the expectations of the target audience, which primarily comprises community members seeking easy access to information about available books, new books, and their borrowing history. By assessing user information needs, I intend to identify the specific types of content and functionalities that are most valuable to the users of the library. Furthermore, I aim to gauge user satisfaction with the app's existing features and pinpoint any usability issues they may encounter during their interactions. This research will not only enhance the overall user experience but also guide us in prioritizing future feature development.

The purpose of this chapter is to describe the user experience research process, which was conducted to gain a better understanding of the library's target audience and their needs. This chapter will discuss the research methodology, the research process, and the results of the research, which was used to inform the design and development of the mobile application.

For the User Experience research, I utilized the `Design Thinking` process.

2.1   Design Thinking

Design thinking is a non-linear, iterative process that teams use to understand users, challenge assumptions, redefine problems and create innovative solutions to prototype and test. Involving five phases—Empathize, Define, Ideate, Prototype and Test—it is most useful to tackle problems that are ill-defined or unknown.

2.2   Empathize

In this chapter, I will discuss how I gathered insights about the target users, their motivations, frustrations, and unique circumstances. This chapter should focus on conducting, surveys, and usability testing to empathize with the user personas.

2.2.1   Understanding User Context

In this chapter, I explore the foundational understanding of the unique context that defines the lives of retirees in remote areas of Finland. Through extensive research and direct interaction with the target users, I've gained profound insights into their daily challenges and opportunities.

The users reside in small, communities nestled in the vast Finnish wilderness, where physical distance poses a considerable barrier to accessing essential resources like a local library. Geographical isolation and long travel times to the nearest library can be frustrating and physically demanding, particularly for older individuals. This geographical context underscores the critical need for a digital solution that bridges these physical gaps, making it easier and more convenient for retirees to access a vast collection of books and educational materials.

These insights also reveal the significance of preserving local culture and fostering community engagement. The users have a deep sense of pride in their remote communities and wish to

share their local culture and knowledge. They value social connections and desire a platform that allows them to be able to comfortably browser the resources of the community driven library.

The "Empathize" phase has been instrumental in understanding the unique circumstances of the target users, emphasizing the importance of not only convenient access to library resources but also the facilitation of community connections and the preservation of local culture. These insights serve as the cornerstone of the empathetic design approach, ensuring that the subsequent phases of the project align closely with the needs and desires of individuals in remote Finland.

### 2.2.2   User Persona Creation

As part of my UX research for the community library mobile app, I have delved into the critical phase of User Persona Development. This process has enabled me to craft detailed and empathetic user personas, each representing a distinct archetype among my target audience. These personas serve as the embodiment of user characteristics, needs, and preferences, allowing me to better empathize with and design for the real people who will ultimately use my app.

The creation of user personas has proven to be a pivotal step in my UX research for the community library mobile app. These personas, representing distinct archetypes within the target audience, have provided invaluable insights and empathy into the diverse needs, preferences, and motivations of potential users. Each persona brings to the forefront a unique set of goals, frustrations, and aspirations.

Through persona research, I've gained a deeper understanding of the challenges these individuals face in accessing library resources and the opportunities to enhance their library experience through a mobile app. It's evident that convenience, user-friendliness, and community engagement hold paramount importance across all personas. The personas have highlighted the varying skills and experiences of technological proficiency among the target audience, emphasizing the need for a user interface that is not only intuitive but also accessible to users of different skill levels.

In the process of identifying the target audience, I have meticulously crafted user personas that serve as the compass in understanding and catering the needs of the primary users. The target audience especially comprises retired individuals, facing the challenges of isolation. These retirees, ranging in age from 40 to 70, represent a segment of the population that may encounter limited access to conventional library services. Therefore, the primary goal is to bridge this gap by developing a community-driven library mobile app that brings the library to their fingertips, offering a sense of connection and access to a world of literature and community engagement.

The group of my target audience, was guided by the unique challenges they face. Living in remote locations, these individuals may find it challenging to access libraries nearby them, making them an ideal demographic for a digital library solution. By addressing their specific needs and preferences, I aspire to empower them with a user-friendly app that facilitates book browsing, access to personal borrowing history, and seamless community engagement—all without the need for complex sign-up or login procedures. The research and

design efforts are tightly aligned with these user personas, ensuring that every aspect of the mobile app is tailored to enhance the library experience for this distinct audience.

The persona research results have not only provided a user-centered foundation for design and development but have also sparked inspiration for feature prioritization. For instance, the desire for advanced search functionality and efficient book tracking, as expressed by personas like Seppo, suggests the importance of focusing on robust search and user library management features. Simultaneously, personas like Aino underscore the significance of simplicity in the usage of the app, to make it as simple as possible. By aligning my development efforts with the personas' goals and addressing their frustrations, I aim to create a mobile app that not only meets but exceeds the expectations of the users, ultimately enriching their library experience.
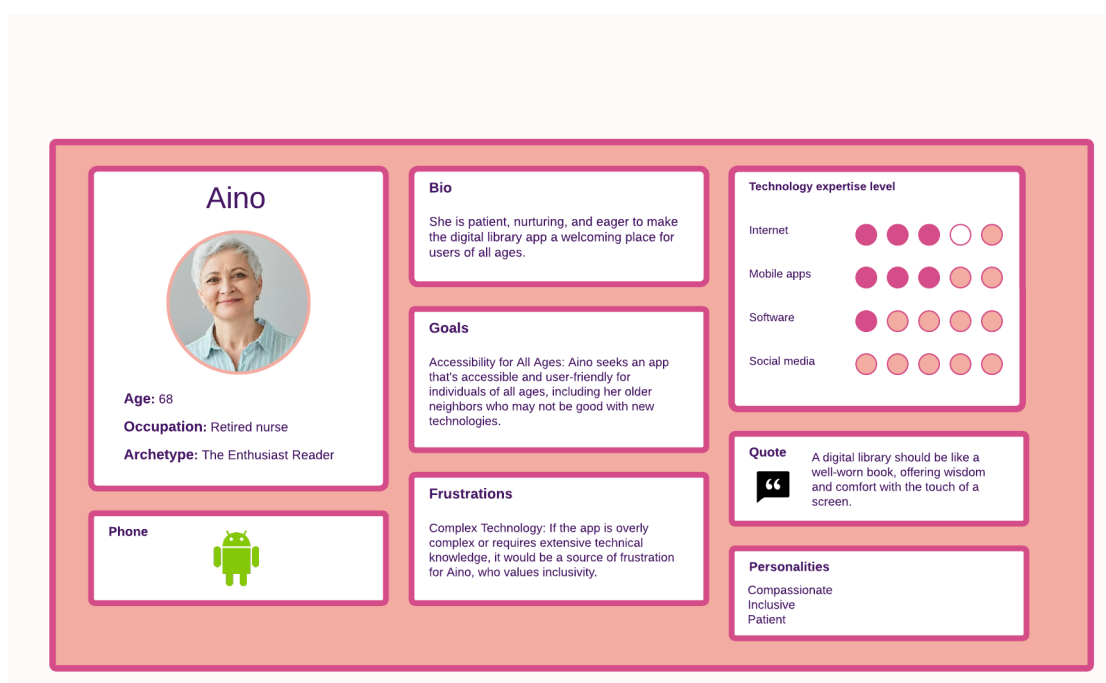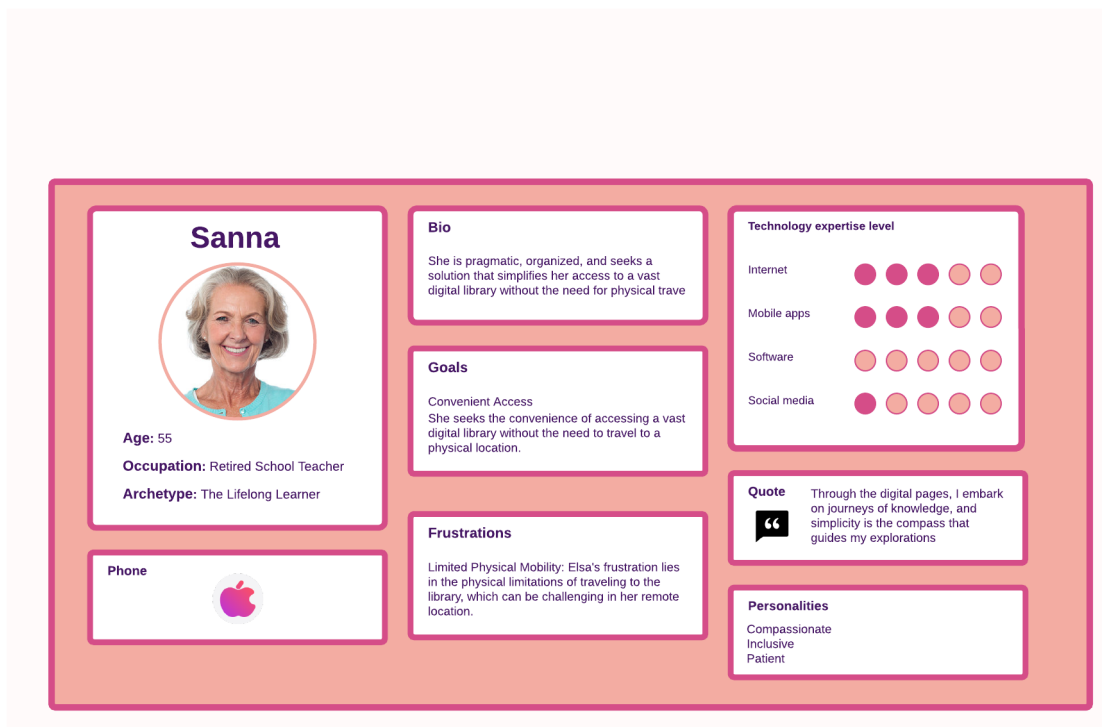


Figure 1: Persona - Aino

**Sanna**

**Age:** 55

**Occupation:** Retired School Teacher

**Archetype:** The Lifelong Learner

**Phone**

**Bio**

She is pragmatic, organized, and seeks a solution that simplifies her access to a vast digital library without the need for physical trave

**Goals**

Convenient Access
She seeks the convenience of accessing a vast digital library without the need to travel to a physical location.

**Frustrations**

Limited Physical Mobility: Elsa's frustration lies in the physical limitations of traveling to the library, which can be challenging in her remote location.

**Technology expertise level**

Internet

Mobile apps

Software

Social media

**Quote**

Through the digital pages, I embark on journeys of knowledge, and simplicity is the compass that guides my explorations

**Personalities**

Compassionate
Inclusive
Patient

Figure 2: Persona - Sanna

**Seppo**

**Age:** 62

**Occupation:** Retired Engineer

**Archetype:** The Lifelong Learner

**Phone**

**Bio**

She is pragmatic, organized, and seeks a solution that simplifies her access to a vast digital library without the need for physical trave

**Goals**

Stay mentally active
He sees reading and learning as a way to stay mentally active in retirement, and he wants the app to support his intellectual pursuits.

**Frustrations**

Complex User Interface
He prefers a simple and intuitive interface, so a complex or cluttered design would be a source of frustration.

**Technology expertise level**

Internet

Mobile apps

Software

Social media

**Quote**

Through the digital pages, I embark on journeys of knowledge, and simplicity is the compass that guides my explorations.

**Personalities**
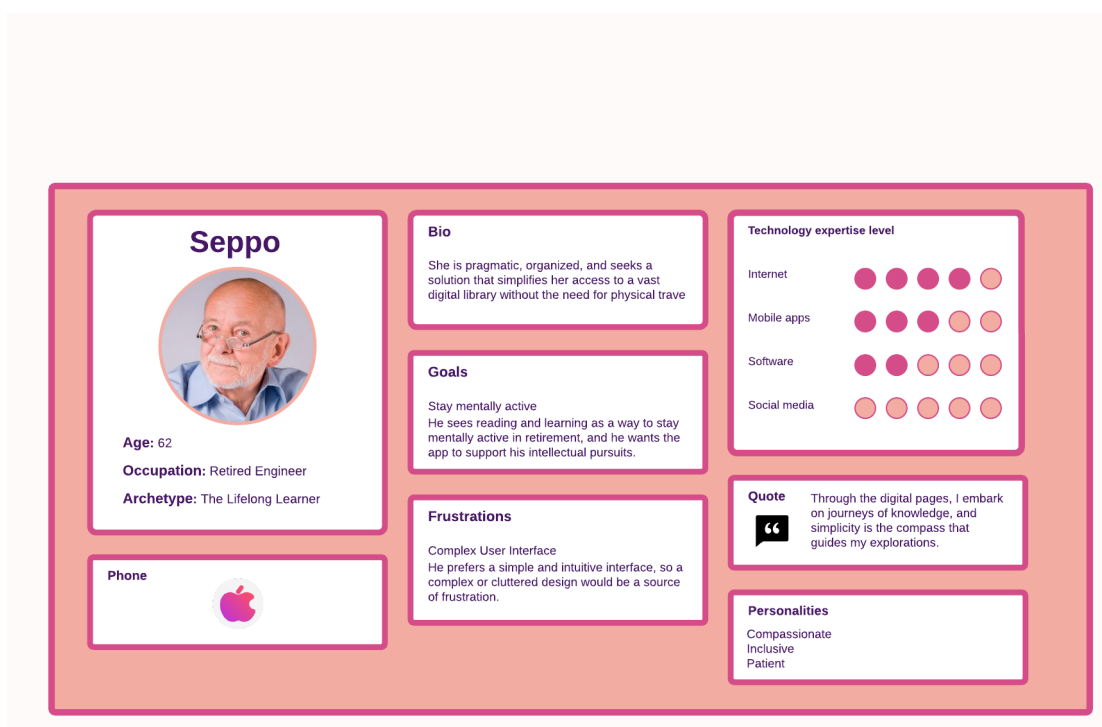
Compassionate
Inclusive
Patient

Figure 3: Persona Seppo

### 2.2.3 User Interviews and Surveys

This chapter explores the methodologies employed to gain a comprehensive understanding of user needs and preferences. User surveys played a central role in empathizing with the retirees' unique circumstances. Through qualitative surveys[1] I gained insights into their

motivations, frustrations, and the experiences they seek in a community library mobile app. These surveys provided a qualitative depth to my understanding. Simultaneously, quantitative data from surveys helped validate trends and preferences identified through the personas, ensuring the understanding was well-rounded and representative of the broader user base. The chapter details the process of conducting surveys, the questions asked, and the insights gleaned from these interactions.

My user survey was designed to be conversational and open-ended, allowing participants to freely express their thoughts and experiences. I aimed to explore how Aino, Seppo, and Sanna typically accessed books and engaged in cultural and intellectual activities in their communities.

The insights I gathered through the surveys were invaluable. Aino, "The Enthusiastic Reader," expressed her passion for reading and her goal of ensuring that the app is accessible and user-friendly for individuals of all ages, especially older neighbors who may not be tech-savvy. Aino's frustration lies in complex technology; she would be dissatisfied with an app that is overly complex or requires extensive technical knowledge.

Seppo, "The Tech-Savvy Explorer," shared his aspiration to stay mentally active in retirement through reading and learning. He emphasized the importance of a simple and intuitive interface in the app. Seppo's frustration stems from a complex user interface, and he seeks a design that is straightforward and clutter-free.

Sanna, "The Lifelong Learner," highlighted her desire for convenient access to a library's vast resources, without the need to travel to a physical location, just to browse books, she might or might not like. Her goal is accessibility, especially for those with limited physical mobility. Sanna's frustration centers on the physical limitations of traveling to the library, which can be challenging in her remote location.

Complementing these surveys provided quantitative data to validate trends and preferences identified through the user personas and surveys. Participants were asked about their reading habits, technological proficiency, and expectations from a digital library app. This quantitative data allowed me to confirm my hypothesis and gain a broader understanding of the user base.

The results of the surveys were synthesized to identify common patterns, preferences, and challenges. For instance, the desire for user-friendliness, advanced search capabilities, and convenient access were universally desired. The app's design needed to be simple and intuitive, avoiding complexity, to meet the preferences of users like Aino and Seppo. Sanna's emphasis on convenience and accessibility was a recurring theme.

The "User Interviews and Surveys" phase served as a vital bridge between understanding the user context and creating user personas. It allowed me to empathize deeply with the lives and aspirations of Aino, Seppo, and Sanna and ensured that my design approach was rooted in their real needs and desires. These insights form the foundation upon which I define the problem statement and set user-centered goals in the subsequent phases of the project.

### 2.2.4 Empathy Maps

In the pursuit of creating a user-centered digital library app the integration of empathy maps played a pivotal role in guiding the design process. These empathy maps served as a tangible

representation of the deep understanding I had gained about the target users, Aino, Seppo, and Sanna. With a focus on the "Says," "Does," "Thinks," and "Feels" categories, the empathy maps provided a multidimensional view of the personas, allowing to effectively empathize with their unique perspectives and needs.

The empathy maps played a critical role during the "Empathize" phase of the design thinking process. By visualizing the thoughts, emotions, and actions of the personas, I gained valuable insights into their daily lives and aspirations. These visual representations of Aino's, Seppo's, and Sanna's experiences helped to align the design decisions with their specific preferences and frustrations.

For Aino I could see her passion for reading, her inclination towards inclusivity, and her apprehension regarding complex technology. This insight informed the decisions to prioritize user-friendliness and accessibility in the app's design, ensuring that it could be comfortably used by individuals of all ages.
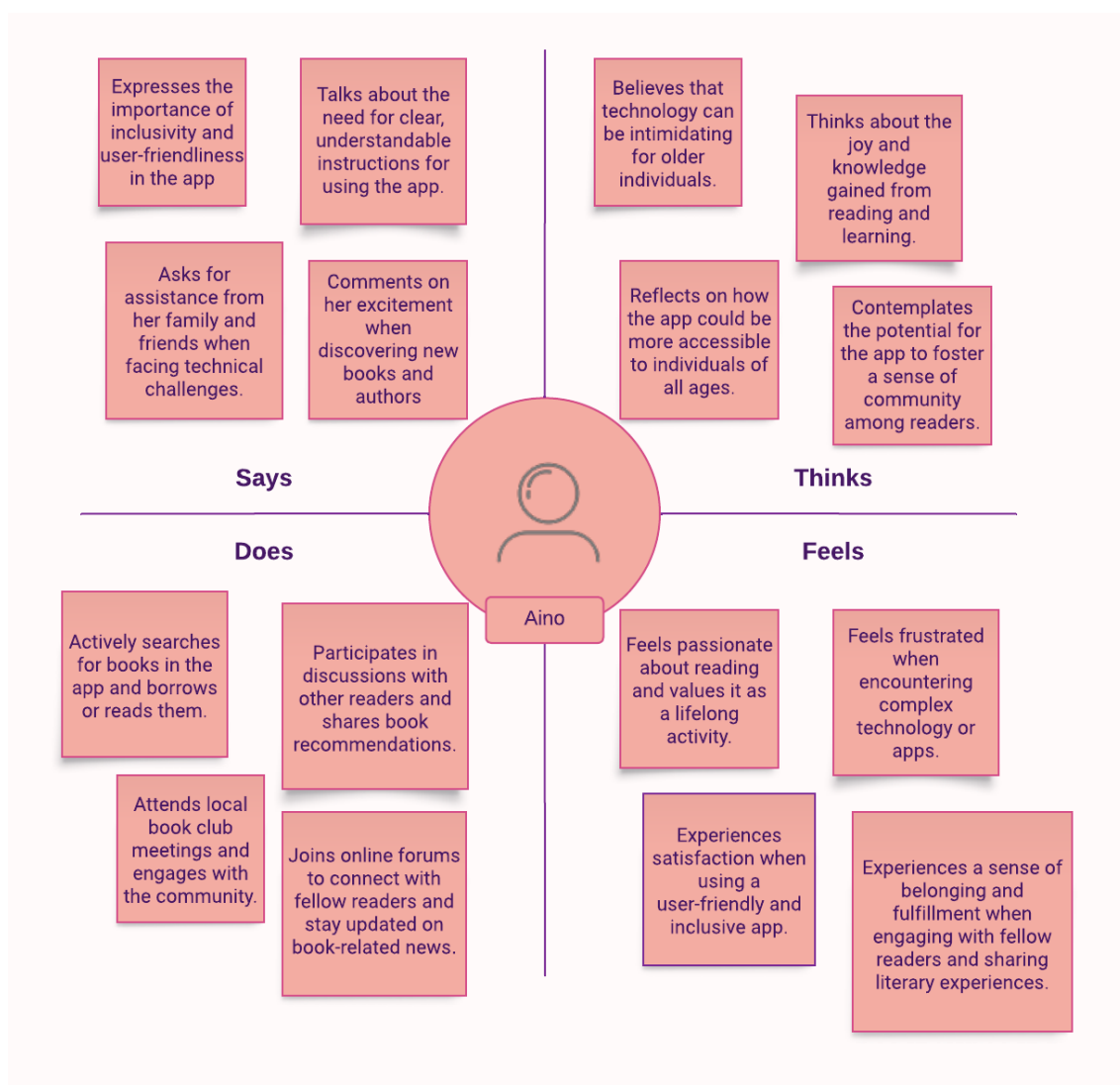


Figure 4: Aino Empathy Map

Seppo demonstrated a thirst for learning and an appreciation for intuitive interfaces. With this understanding, I aimed to simplify the app's user interface and provide advanced search

capabilities, ensuring a seamless experience for users like Seppo.
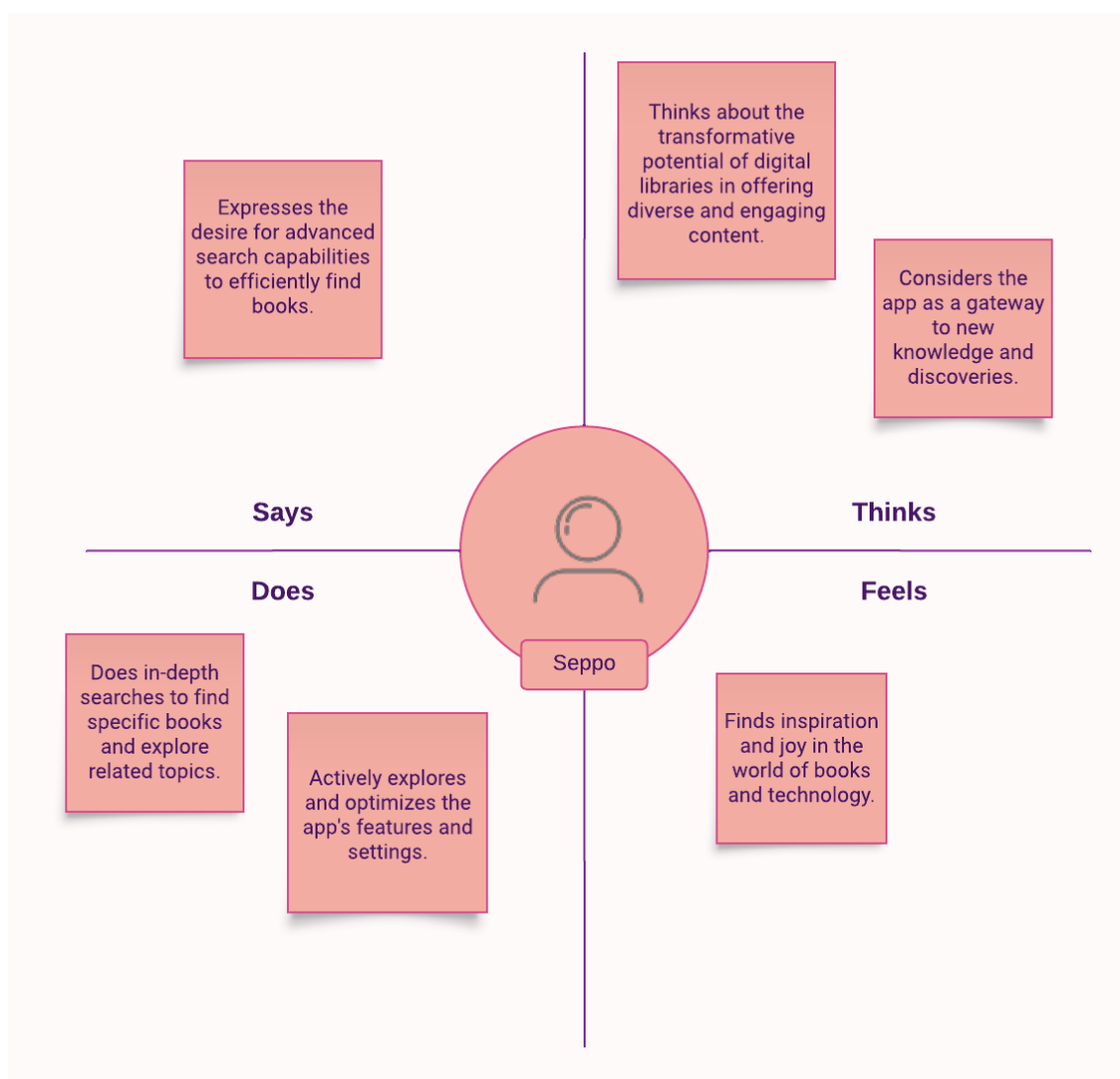


Figure 5: Seppo Empathy Map

Sanna emphasized the importance of convenient access and inclusivity, particularly for individuals with limited physical mobility. Her empathy map underscored the significance of designing an app that allows users to access a wealth of educational materials from the comfort of their homes.
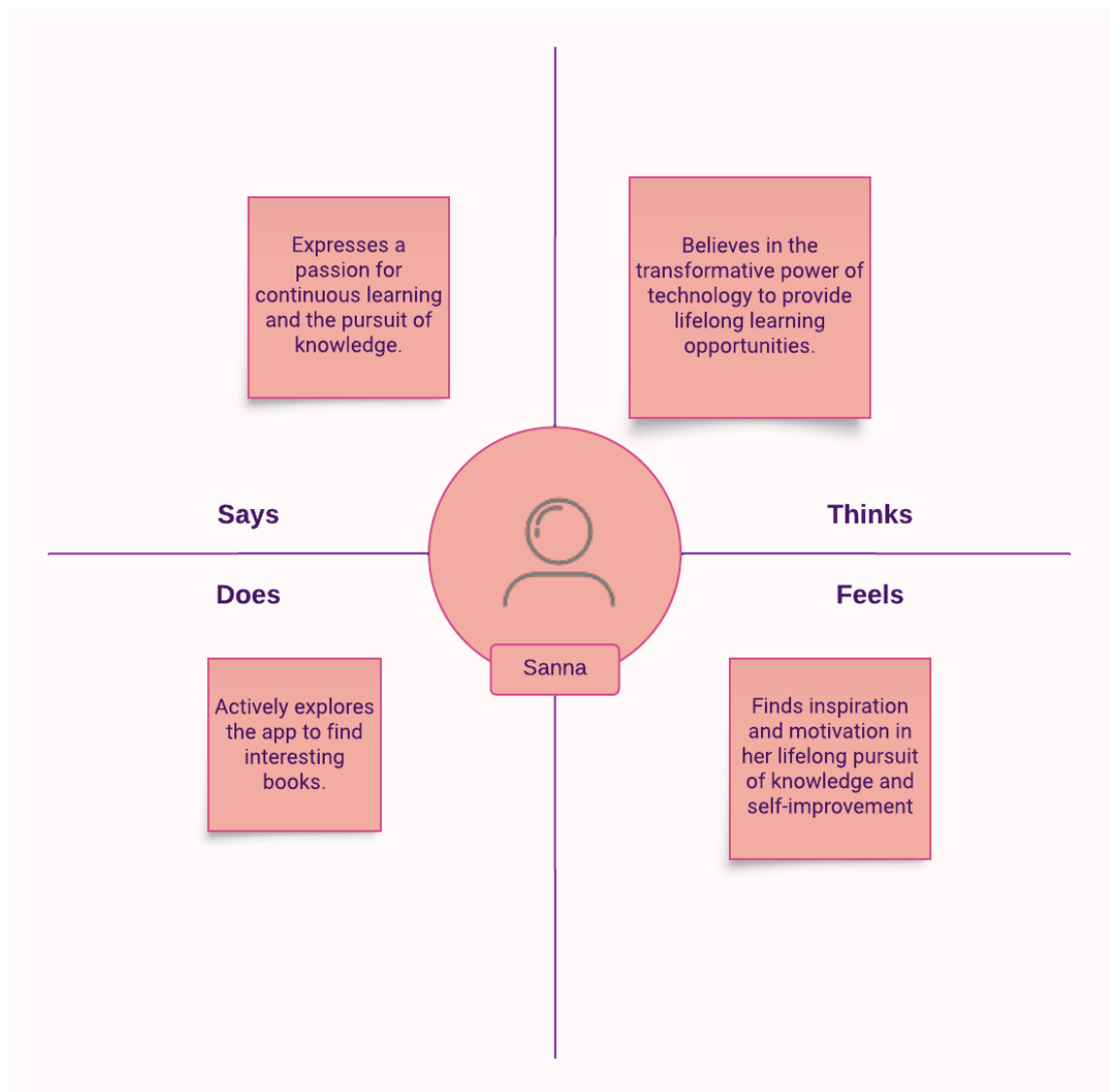
Figure 6: Seppo Empathy Map

Incorporating these empathy maps into the design process allowed us to make informed, user-centered decisions at every stage. They served as a constant reminder of the real people who would be using the digital library app, ensuring that the design choices aligned with their goals and values. Ultimately, the empathy maps were a powerful tool in creating a digital library app that genuinely caters to the needs and desires of the target users.

2.2.5   Defining the Research Objectives

The "Empathize" phase, as a vital component of the design thinking process, served as the bridge to understanding the users deeply. However, to channel the explorations effectively, it was compulsory to define clear research objectives. These objectives acted as compass, guiding the investigations and ensuring that the ultimate target has remained aligned with the project's goals.

First and foremost, the research objectives were centered on unraveling the unmet needs and desires of the target users, Aino, Seppo, and Sanna. The outstanding objective was to identify the primary challenges they faced in accessing books and participating in intellectual pursuits.

Understanding these challenges would pave the way for the mobile app to become a meaningful solution.

A secondary objective was to unearth any preconceived notions or misconceptions the users might have about digital library apps. By identifying any barriers that could be stemming from misconceptions, I could develop strategies to overcome resistance to adopting new technology.

Moreover, the research objectives extended to recognizing the expectations of the users. I aimed to uncover what Aino, Seppo, and Sanna anticipated from a digital library app. This involved exploring their ideas about user-friendliness, accessibility, content availability, and community engagement. The goal was to bridge the gap between user expectations and app functionality.

Additionally, I aimed to delve into the potential emotional and social factors at play in their lives. This involved understanding how reading and learning enriched their lives and provided opportunities for personal growth and community connection. This objective would help us design an app that not only satisfied functional needs but also resonated with the emotional aspects of their experiences.

Defining the research objectives was an essential step in the "Empathize" phase. These objectives focused the research efforts and ensured that every exploration and insight gathering activity was purpose-driven. By addressing these objectives, I aimed to create a digital library app that truly responded to the unmet needs and aspirations of the users, enabling them to embrace a world of knowledge and community engagement from the comfort of their homes.

### 2.2.6  Summary of Empathize Phase

In the "Empathize Phase" of the design thinking journey, I ventured into the world of potential users, retirees living in the remote corners of Finland. This phase was the gateway to comprehending their unique needs and desires, ensuring that the subsequent stages of the project would be deeply rooted in their realities. To accomplish this, I employed a multifaceted approach, including user interviews, surveys, observations, and the creation of empathy maps.

User surveys became the cornerstones of the research during this phase. Engaging in one-on-one conversations with potential users allowed me to glean profound insights. Through these interactions, Aino, vividly conveyed her fervent desire for inclusivity and user-friendliness in the app. Her insistence on clear, understandable instructions highlighted the importance of making the app accessible to individuals of all ages, especially older users like herself. Meanwhile, Seppo consistently reiterated his preference for simplicity and intuitiveness in the app's interface. Advanced search capabilities were a top priority for him. Sanna emphasized the significance of convenient, digital access to the library, eliminating the need for physically traveling to a distant library location.

Observation and the creation of empathy maps provided an additional layer of insight. I witnessed the daily routines, struggles, and moments of joy experienced by retirees in remote regions. This immersive experience allowed me to better appreciate their challenges and triumphs. The empathy maps, in particular, were instrumental in visualizing the personas' thoughts, actions, emotions, and expressions. These visual representations of the personas

served as constant reminders of the real people who would be engaging with the app.

In summary, the "Empathize Phase" laid a robust foundation for the design thinking process. It unearthed profound insights into the users' lives, ensuring that the design choices remained user-centered throughout the project. The firsthand perspectives of Aino, Seppo, and Sanna, brought to life through interviews, surveys, observations, and empathy maps, were instrumental in guiding the approach and fostering a deep sense of empathy for the target users. This phase was instrumental in creating a digital library app that truly caters to the needs and aspirations of it's users.

## 2.3 Defining the Problem

In the journey to create a user-centered library app, I have traversed the foundational "Empathize" phase, gaining profound insights into the lives, aspirations, and unique circumstances of the target users. With a clear understanding of their needs and desires, I now embark on the "Defining the Problem" phase, a pivotal stage in the design thinking process. This chapter marks our transition from empathizing with the users to crafting a precise problem statement that encapsulates the core challenges I aim to address. Here, I delve into the intricacies of articulating a problem that resonates with the experiences and frustrations of the users, setting the stage for the ideation and solution development to come.

The essence of "Defining the Problem" lies in narrowing down our focus from a broad understanding of user needs to a specific issue that can be resolved through the library app. As I proceed, I will explore the intricacies of problem definition and the methodologies employed to ensure its clarity and relevance. The problem definition sets the direction for the design and development phases, guiding our efforts toward creating an app that genuinely serves the the community. In this chapter, I will explore the process of problem articulation, user story development, and the crafting of a problem statement that describes the central challenge the app seeks to overcome.

### 2.3.1 Problem Definition

The problem comes from the lack of accessible and user-friendly digital platform, which offers browsing the library's resources, tailored to the unique needs of the target users. Aino, seeks an app that is inclusive and accommodating for individuals of all ages, including her older neighbors who may not possess advanced technological skills. Seppo, values simplicity in an app's user interface and desires advanced search capabilities, which are currently beyond his reach. Sanna, yearns for the convenience of accessing a vast digital library without the need for arduous journeys to physical locations. Her frustration arises from the physical limitations of traveling to a library.

The problem definition revolves around crafting an app that effectively addresses these user needs and frustrations. It's the challenge of creating a platform that not only provides easy access to a rich repository of books but also ensures that the app's user interface is intuitive, even for those who are not having deep technical knowledge. Moreover, it's about developing a solution that mitigates the need for physical mobility, ensuring that Sanna and others like her can engage with the app from the comfort of their homes. The problem statement is, therefore, rooted in the goal of creating a bridge to connect these users with the world of knowledge and with the joy of reading, fostering a sense of community, and eliminating the

digital divergence they currently face.

## 2.3.2   Brainstorming

The next phase of the UX research process is to brainstorm ideas for the library app. This phase is critical to the design thinking process, as it allows me to explore a wide range of potential solutions to the problem statement. The brainstorming process is a creative exercise that encourages me to think outside the box and consider all possible solutions, no matter how unconventional they may seem. This phase is also an opportunity to collaborate with others and gain new perspectives on the problem statement.

I used this phase to brainstorm potential features, functionalities, and design elements for the library app. I explored ideas for the app's user interface, including the layout, color scheme, and typography. I also considered the app's functionality, including the search capabilities, book tracking, and community engagement features. The brainstorming process allowed me to think creatively and explore a wide range of ideas, ultimately leading to the development of a robust feature set for the app.
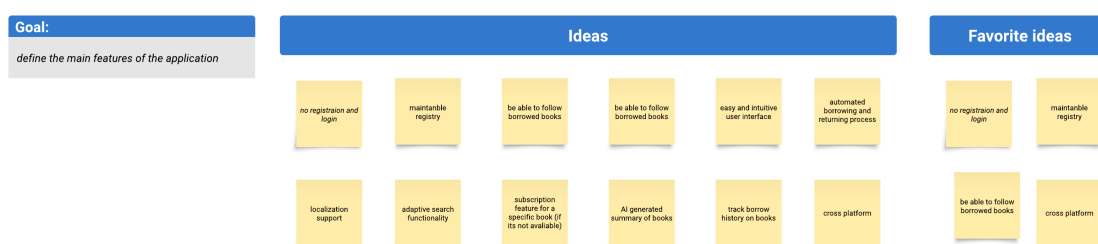


Figure 7: Brainstorming Map

The brainstorming process focused on the outcome of the User Persona research phase, which pinpointing the main pinpoints of the users. This includes the simple user interface, the advanced search capabilities, and the overall convenient access to the library.

To keep the application simple, there is no signup or login process. The user can start using the application right away, without any hassle. This way the registration, the need of remembering their account, and password, the user confirmation, and the email verification is eliminated.

The next main thing is the search capabilities. The user can search for books by title, author, or ISBN. The search results are displayed in a list, where the user can see the book's title, author, and the cover image. The user can also see the book's availability. The user can also filter the results by the book's availability.

Some features are derived from the maintainability of the library. The admin personnel of the library should be aware at all times, which books are currently borrowed, and which books are available. In order to make this happen, I needed to design a process, which is easy to use for the users, but also gives the needed information for the people who are running the book.

This process is done via QR codes. Every book is given a QR code. The users need to read this QR code within the app when they are borrowing. When they are returning the book, they just need to press a simple button in the app, to set the book as returned.

The application needs to be localized, so it's available in both Finnish and English language.

It also needs to be cross-platform, so it's easy to maintain, and it's available for both Android and iOS users.

## 2.4 Prototype: Design and Development

In this chapter of "Developing Prototypes", the ideas generated during the ideation and brainstorming sessions begin to take shape. Here, I will explore the methodologies and strategies employed to bring these creative concepts to life, paving the way for an interactive and tangible representation of the app's future.

The process of developing prototypes is essentially a process of iterative design. It involves creating low-fidelity prototypes that offer a basic representation of the app's key functionalities. These early-stage prototypes are crucial for testing the feasibility of ideas and ensuring that the envisioned features align with our problem definition and user personas.

The goal is to identify any usability issues or challenges early in the process, ensuring that the final product is intuitive and user-friendly. User testing plays a pivotal role in this phase, as we seek feedback from individuals who represent the target users. Their input is invaluable in fine-tuning the prototypes.

As I develop and refine these prototypes, I maintain the user-centered perspective. Each feature and functionality is scrutinized against the user personas and their goals. Aino's need for accessibility, Seppo's preference for simplicity, and Sanna's desire for convenience are all taken into account. This ensures that our prototypes evolve in a manner that directly addresses the users' unique needs and frustrations.

The "Developing Prototypes" phase is not merely about creating a blueprint for the app; it's about crafting a dynamic and interactive representation of the digital library experience I aim to provide. In this chapter, I will delve into the methods employed to breathe life into our ideas, the collaborative approach adopted, and the role of continuous user testing in shaping the final product. As we move from prototypes to the final design, this phase ensures that the target users are at the forefront of the design decisions made.

### 2.4.1 Wireframe

Firstly, I created a paper based wireframe, to start crafting the ideas based on the conducted researches.

Paper-based wireframes are a form of low-fidelity prototyping in the field of UX design. They are physical representations of a user interface or a product's design, created using paper, sketches, or other easily modifiable materials. Paper-based wireframes are often used in the early stages of product development to quickly and cost-effectively test and refine design concepts before moving on to more high-fidelity digital wireframes or final development. They are not meant to replicate the final product's visual design or functionality but rather to capture the basic structure, layout, and user flow. This simplicity makes them ideal for early-stage testing and ideation.

After drawing the wireframe first on paper, I've digitalized them using an online tool called

excalidraw[2].

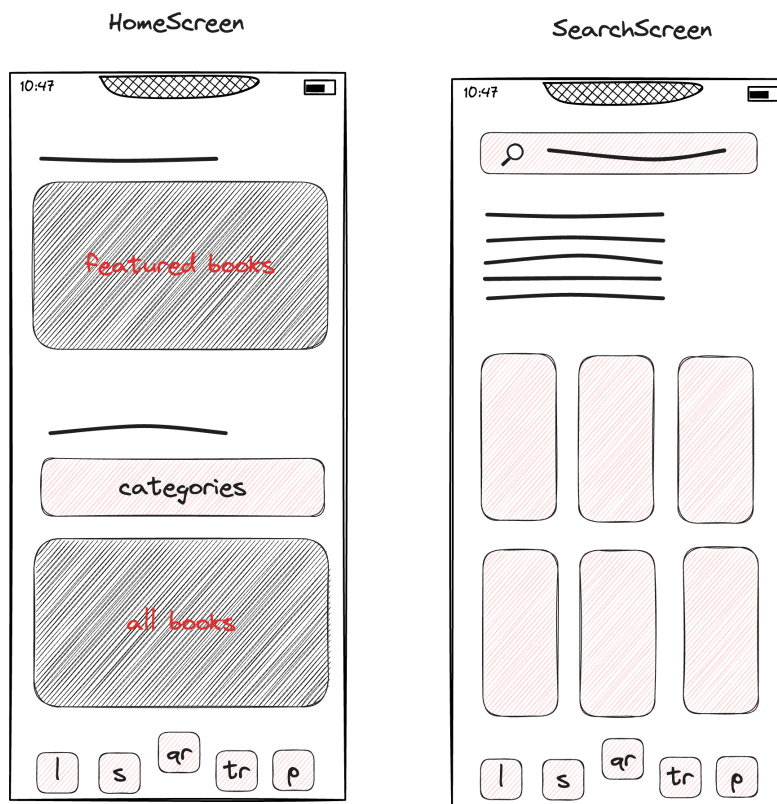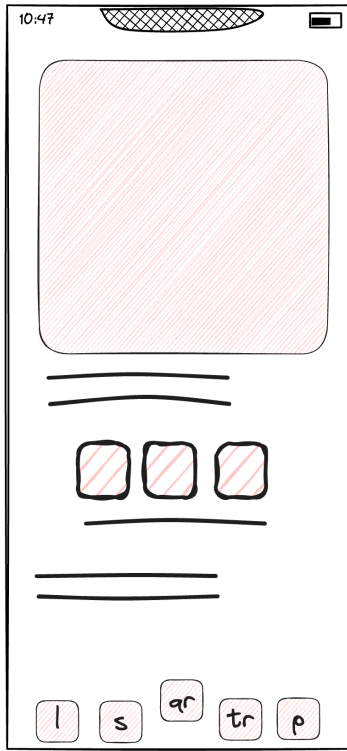The wireframe showcases the layout, and the main user flow of the application.



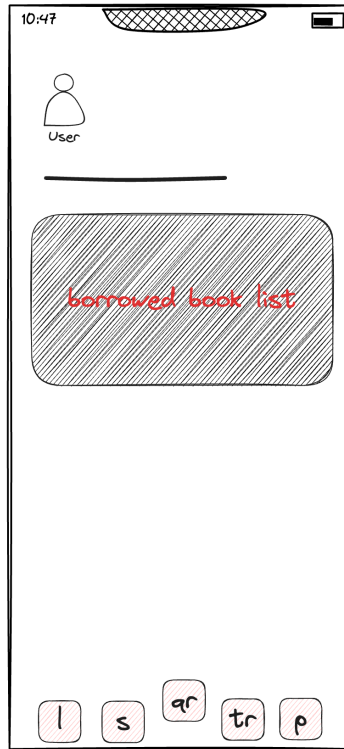Figure 8: Prototype

Book Details Screen

ProfileScreen

borrowed book list

Figure 9: Prototype

QR Scan Screen
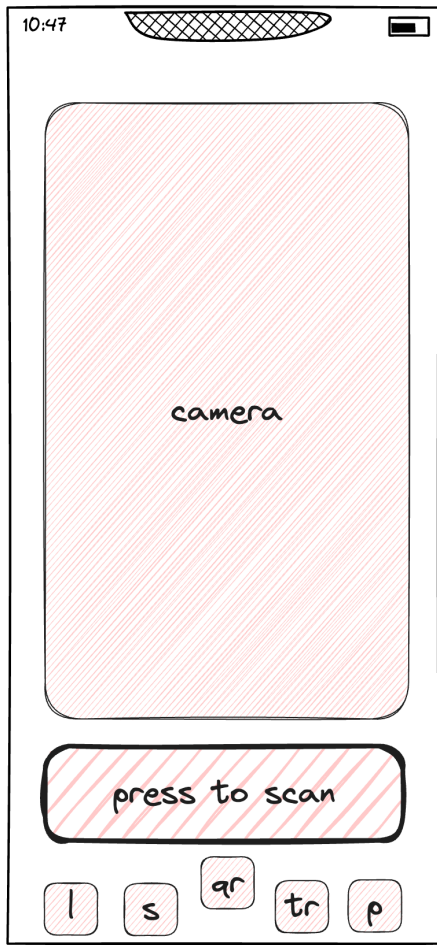


Figure 10: Prototype
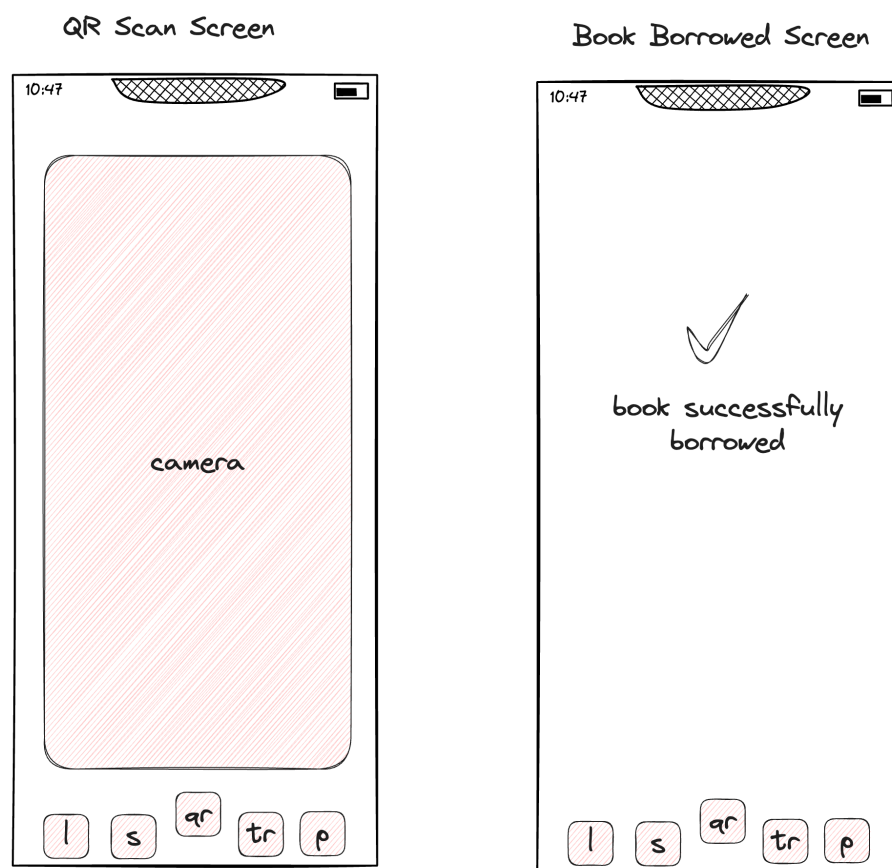
QR Scan Screen          Book Borrowed Screen



Figure 11: Prototype

It demonstrates the main screens of the application, the navigation, and the relation between screens, and gives an overall idea of the direction the UI interface should take.

The main focus I had in mind when designing the prototypes and wireframes was simplicity, and user-friendliness, as this was the main pinpoint of the users. The user interactions are kept to the minimal, and the main features of the application is highlighted the most the users. Therefore, the search, and the book borrow features.

The home screen should offer a few featured books, but also an intuitive way for the user to be able to quickly browse through recommended books.

The profile screen lists the borrow history of the user, but at the same time it also clearly highlights which are the books that are currently borrowed by them.

In the middle of the bottom tab bar, there is a floating button, which clearly outstands from the rest of the buttons in the Tabbar navigator. The book borrow screen, which is accessibly by this floating button, is where the borrowing flow takes place. Users can use their device's camera to scan the barcode of a book they want to borrow. The screen provides real-time feedback and confirmation, making the borrowing process seamless and user-friendly.

After running the wireframes through a few user tests, I've decided to make some changes to the wireframe, to make it more user-friendly. I've come to a realization, that placing an extra user interaction between the QR code scanning, and the book borrowing is confusing. Therefore, I decided, to iterate this idea, and making the borrowing process happen

immediately after the QR code had been scanned successfully. This way, the user doesn't need to press an extra button to confirm the borrowing, and made the entire process way more user-friendly.

## 2.4.2   Prototype

After, the wireframe was finalized, I've started to work on the high fidelity prototype. The high fidelity prototype is a more detailed version of the wireframe. It's a more detailed representation of the application, which is closer to the final product.

The first version of the prototype contains a more detailed mockup, and draws clear visual representation of the hierarchy of the screens in the application. This already carries the first idea of the layout and the arrangements of the UI elements across screens.

The other main purpose of the prototype was, to carefully design the navigation between the screens. This was a challenging part, because it had to be kept minimal, intuitive, easy to follow. Designing and implementing navigation stack in mobile application are always challenging, to avoid deep nesting the screens, but still keep the navigation as simple as possible.

During the prototyping session, the target, in the context of the navigation was to every main feature of the application has to be one navigation away. The secondary features could be available maximum two navigation away.
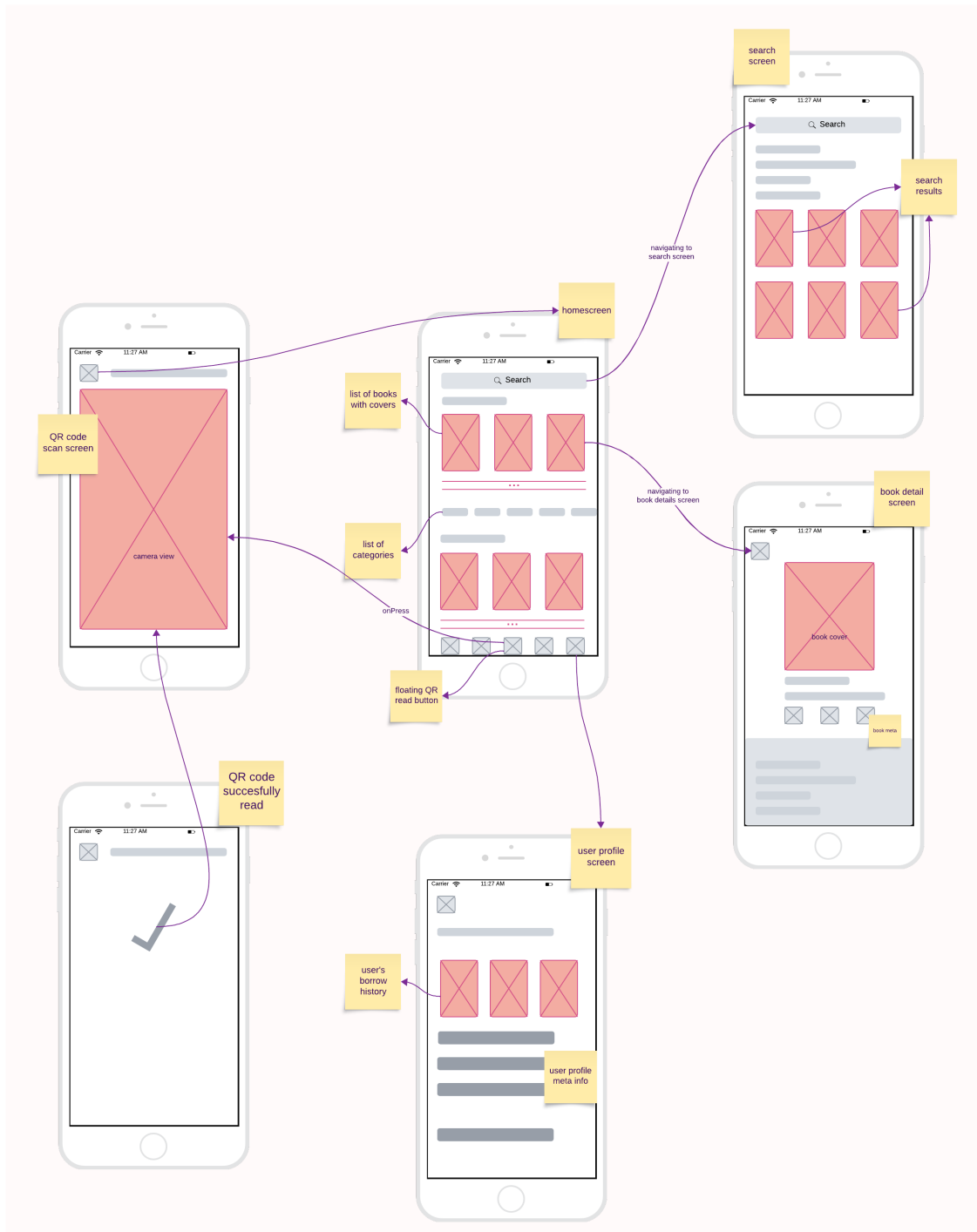
Figure 12: Wireframe

The above prototype is a blueprint representation of the application. It demonstrates the ideal navigation, of keeping all the main features only one level deep in the navigation stack.

These main screens (and their features) are:

- Home Screen
- Search Screen
- Profile Screen

- Borrow Screen

- Single Book Screen

The second level screens, containing secondary features, are:

- User's active borrowed book list

- User's borrowing history

### 2.4.3  UI Design

Following the wireframe and prototype, the next step was to design the User Interface of the application.

One of the key principles driving the UI design was user-centeredness. I continuously referred back to the user personas, to ensure that the design aligned with their goals and preferences. The UI was carefully crafted to be clean, intuitive, and easy to use, meeting the unique needs of the target audience. Elements like font sizes, color schemes, and button placements were chosen with the intention of enhancing accessibility for individuals of all ages.

Establishing a clear visual hierarchy was crucial for guiding users through the app effortlessly. I employed various design elements, such as contrasting colors, typography, and layout, to emphasize important features and content. This not only enhances the app's aesthetics but also aids in making it easy to navigate, as suggested by the user personas. The design ensures that users can quickly identify and access key functionalities like book searches, profile settings, and borrowing history.

The UI design incorporated interactive elements that mimic real-world interactions, making the app more engaging and user-friendly. Features like swipe gestures for book browsing, interactive buttons for borrowing, and smooth transitions between screens create an enjoyable user experience. These interactive elements were designed with convenience in mind, aiming to provide an efficient and enjoyable experience for all users.

Selecting the right color scheme is an important aspect of UI design, as it has a profound impact on user perception and experience. In the library app, the light color scheme with orange as the dominant color plays a pivotal role in setting the tone and creating a visually appealing interface. The choice of colors goes beyond aesthetics; it influences user emotions, usability, and brand identity. Orange, often associated with enthusiasm, warmth, and creativity, aligns with the app's goal of fostering a welcoming and engaging environment for the users.

| #FF9900 | #FDCE33 | #FFFFFF | #333333 | #828282 |
|---------|---------|---------|---------|---------|

Figure 13:  Color Scheme

The light color scheme with its use of orange is carefully selected to invoke specific emotions and interactions. Orange, known for its energetic and uplifting qualities, encourages users to feel positive and motivated. It enhances the readability of text, making it easy for users to engage with content, which is crucial for an app primarily focused on reading and learning. Additionally, the contrast between light backgrounds and orange accents aids in guiding users through the app's functionalities and creating a visually coherent and user-friendly experience. The color scheme ensures that the app aligns with the goals of inclusivity, simplicity, and convenience, catering to the needs and preferences of your target audience, and reinforcing a memorable and engaging user experience.

After defining the color scheme, I've started to design the UI elements of the application.

The UI design of the app is the embodiment of a user-centered approach, carefully crafted to cater to the specific needs and expectations of the target audience. The design exudes simplicity and intuitiveness, with a clean, uncluttered interface that encourages effortless navigation. As users like Aino, Seppo, and Sanna engage with the app, they'll find that every element and interaction is thoughtfully designed to align with their goals, whether it's convenient book browsing, easy-to-use search features, or accessible profile settings.
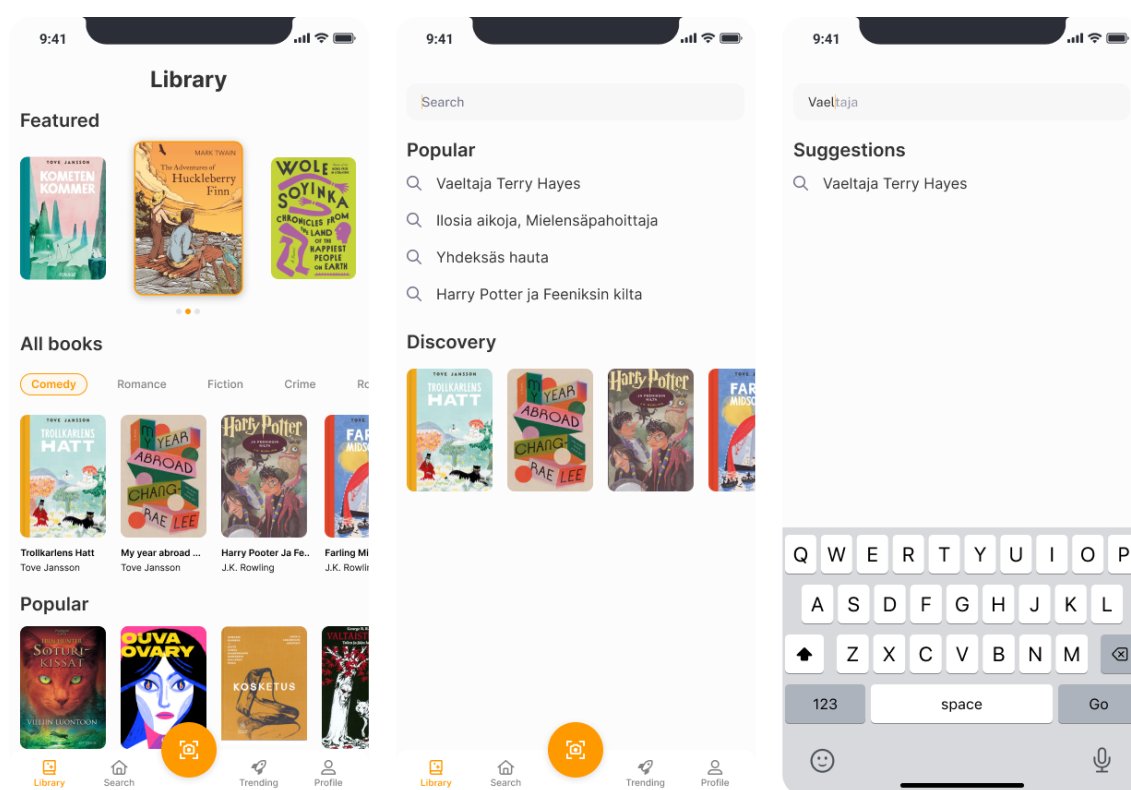


Figure 14: UI Design

Establishing a clear visual hierarchy is fundamental to guiding users seamlessly through the app. Typography is chosen with legibility in mind, ensuring that content is easily readable for users of all ages. Essential features are highlighted with contrasting colors, emphasizing their importance and aiding users in quickly identifying and accessing them. The result is a visually coherent design that supports efficient navigation and readability, contributing to an enjoyable and frustration-free experience.
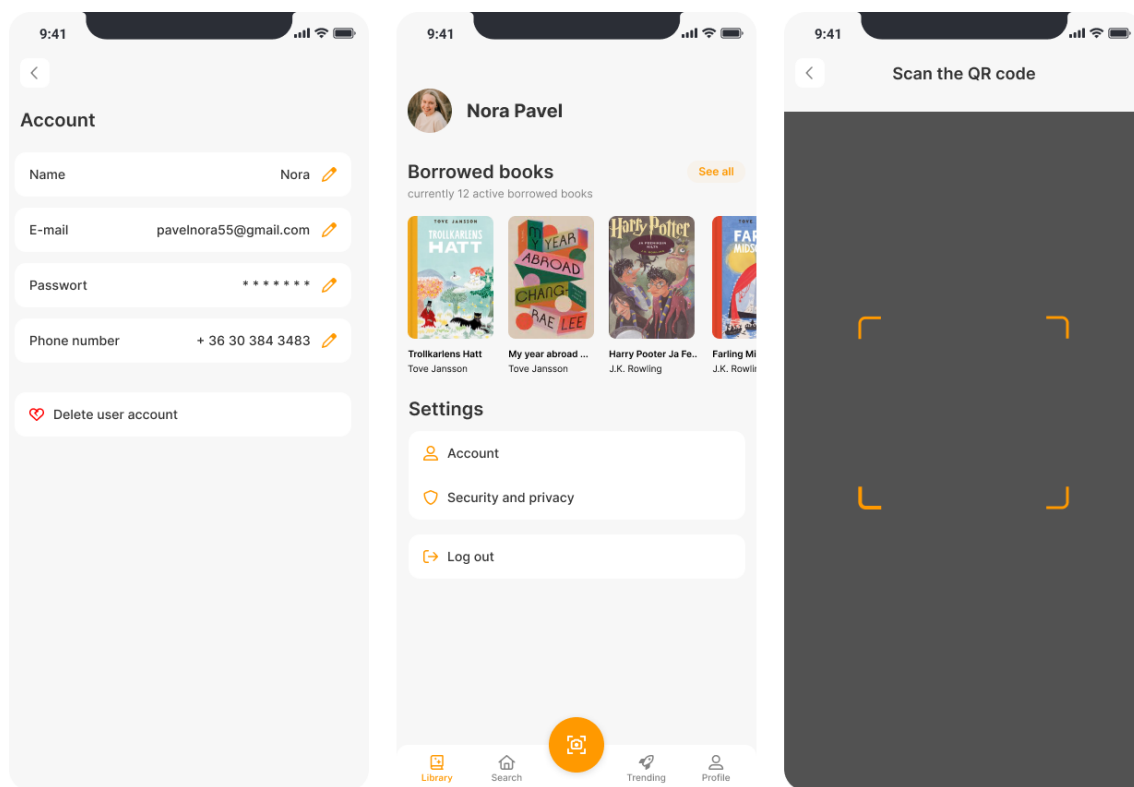
Figure 15: UI Design

Throughout the UI design process, an iterative approach was embraced, with continuous feedback loops from user testing and evaluations. This methodology helped to refine the design iteratively, ensuring that any usability issues were addressed proactively. The UI design reflects a commitment to aligning with user goals and preferences, providing a seamless transition from frustration to fulfillment.
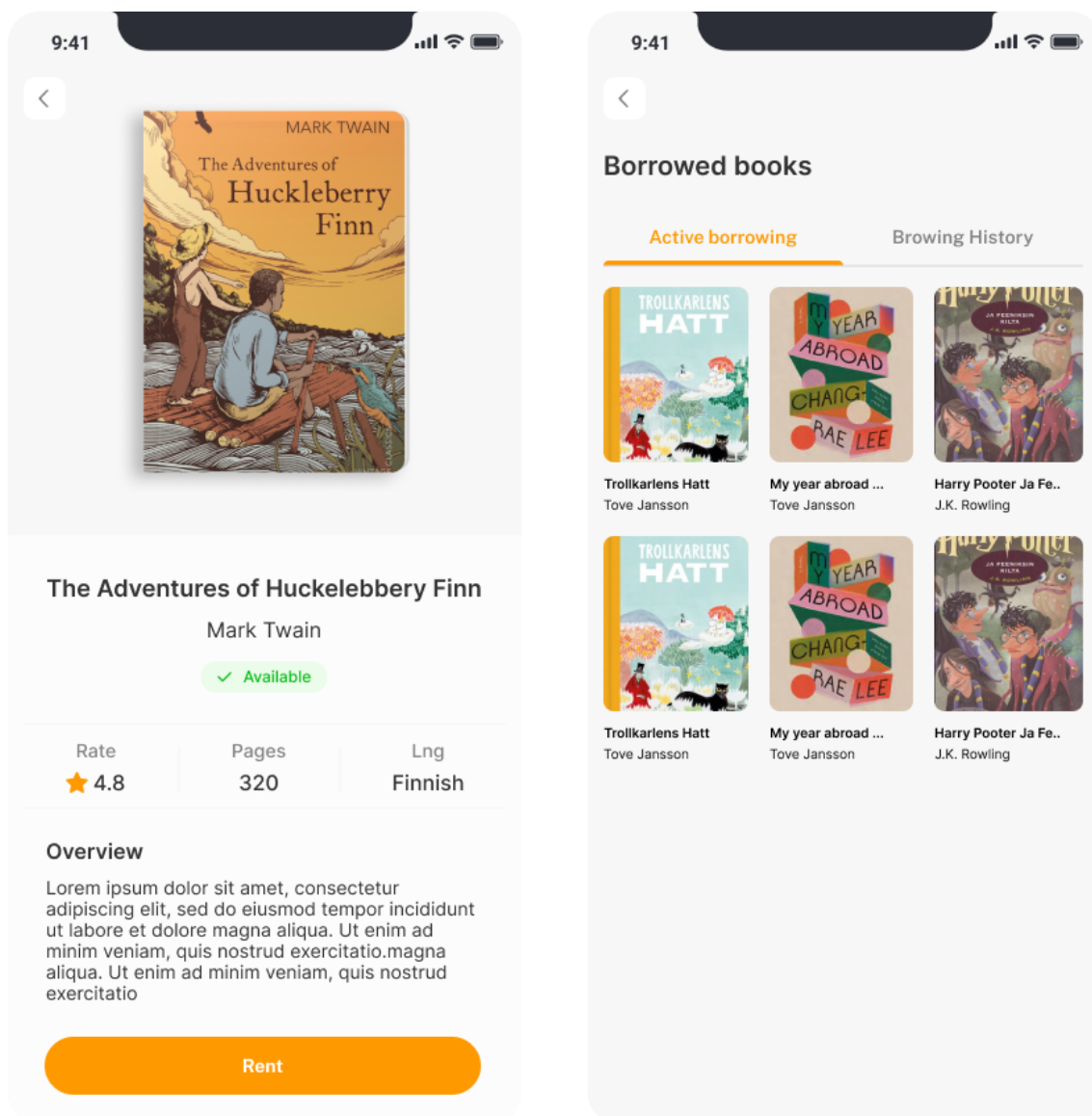
Figure 16: UI Design

In summary, the UI design for app is a testament to a user-centered, visually appealing, and accessible approach. The design's visual hierarchy, interactive elements, mobile responsiveness, and iterative refinement are all part of a strategy that ensures the app's usability and user experience exceed the expectations of the users.

---

[1]excalidraw.com

[2]en.wikipedia.org/wiki/Qualitative_research

3 Development

In this section I am going to describe in detail the end-to-end development environment, for both the server side and the mobile application, including the chosen technologies, architectural decisions deployment processes, and the related services I used during the engineering work.

3.1 Development Environment

The development environment is a crucial part of the development process. It is the place where the application is built, tested, and deployed.

In my thesis work, I have utilized a `monorepo`[3], using `pnpm`[4], and `nx`[5].

3.1.1 Project Structure

The project and the repository structure follows the `monorepository` approach.

There are two main folders in the repository, `apps` and `packages`. The `apps` folder contains standalone applications, and the `packages` folder contains the shared modules and pluggable configs (such as `ESLint` ), and other utils, like common types, and interfaces.

```
├── apps     monorepo entry point for every application
│   ├── mobile-client    the React Native application
│   │   ├── src    source code of the React Native application
│   │   └── test    test files of the React Native application
│   └── server    the NestJJ application
│       ├── documentation    NestJS documentation
│       ├── src    source code of the NestJS application
│       └── test    test files of the NestJS application
└── packages    monorepo entry point for packages
    ├── eslint-config    pluggable ESLint configs
    └── pikkukirjasto-types    Shared types for the application
```

3.1.2 Monorepo

A monorepo is a single repository that contains multiple projects. In my thesis work, I have used a monorepo to contain the mobile application, the backend, the documentation, and other modules which are shared between components and modules of the entire application stack. This approach has several advantages, such as:

The entire application stack can be

- developed together.

- documented together.

- tested together.

- linted together.

- built together.

- deployed together.

- versioned together

- maintained together.

The repository where the entire application stack and source code is located is available at https://github.com/pikkukirjasto/halkeinkiven-pikkukirjasto.

### 3.1.3   nx

`Nx` is a set of extensible dev tools for monorepos, which help with the development, testing, and building of the applications within the monorepo. Using `nx` makes it possible to create hierarchies of applications and libraries, which can be used to create a modular application stack. To decrease cost of having to rebuild and retest the entire application stack, `nx` uses a computation cache, which caches the results of the computations.

### 3.1.4   Workspaces

`pnpm` is a package manager for `JavaScript`. It is a drop-in replacement for `npm`[6], and `yarn`. `pnpm` is a fast, disk space efficient package manager, which uses a single folder for all the projects in the monorepo. This means that the `node_modules` folder is not duplicated for each project, which saves a lot of disk space. `pnpm` has a very fast install time, since it uses hard links to link the dependencies to the `node modules` folder. Besides the fast install time, update time is also fast since it uses a single `node modules` folder for all the projects in the monorepo.

`pnpm` comes with both cold and hot cache, which means that the dependencies are cached between installations. It also uses way less disk space, because the package are reused if they are already installed in the monorepo, for any other package.

The other important feature of `pnpm` is that it supports `workspaces`. Workspaces are a way to setup a monorepo, which this repository is using.

### 3.1.5   Git Repository

The entire monorepo is hosted on `GitHub`, under the pikkukirjasto organization. The repository is available at github.com/pikkukirjasto/halkeinkiven-pikkukirjasto. The repository has a main branch named `master`, which is the main branch of the repository. The `master` branch is protected, and it is not possible to push directly to it.

I've utilized `GitHub`ś built-in issue tracking system to break down the entire engineering work into smaller tasks, which can be tracked easily. When using the issue tracker system in `GitHub`, it is possible to link the issues to the commits, and pull requests, which makes it easier to track down the changes in the repository. It also comes with a labeling feature, which I connected to my `commitlint` setup, to make sure that the commit messages are following the conventional commit specification, and it gives additional sorting feature when listing tickets for releases.

In the repository I've used the following labels:

| Description | Name |
|---|---|
| Something isn't working | 🐛 bug |
| Improvements or additions to documentation | 📖 book |
| This issue or pull request already exists | 👻 ghost |
| New feature or request | 💎 enhancement |
| Good for newcomers | 👶 good first issue |
| Extra attention is needed | 🔍 help wanted |
| This doesn't seem right | ⛔ Invalid |
| Further information is requested | ❓ question |
| This will not be worked on | ⛔ won't fix |
| App's UI systems related tickets | 🎨 UI System |
| Server related tickets | 🔨 backend |
| React Native application related tickets | 📱 client app |
| Repository related tickets | 📌 repository |
| Tickets that won't be done | 🚫 won't do |
| Standalone Desktop App related tickets for Admin personnel | 💻 admin-app |
| Thesis related tickets | 📕 thesis |
| Continuous Integration related tickets | `arrows-clockwise` CI/CD |
| Testing, or testing system related tickets | ♻️ test |
| Release commit or Pull Request | 🎉 release on @master |
| AWS CDK Related Tickets | 🚧 CDK |

Table 1: GitHub Labels

When a commit is made, using the `commitizen`[7]CLI tool, it is possible to select the label which is related to the commit, so the commit message can be easily identified by its corresponding label. For example, this is a list of commits, which were specifically targeted different bugs:

| Commit Message | Commit Hash |
| --- | --- |
| fix(mobile-client): 🐛 eslint | c137be3 |
| fix(server): 🐛 book search API | b814360 |
| fix(typescript): 🐛 bottomtabnavigator type | 866cc6b |
| fix(layout): 🐛 safe area hoc | 54ed5e9 |
| fix(tab-bar): 🐛 wip on animation on bottom-tab | 873f29a |
| fix(eslint): 🐛 module resolver | d4465c5 |
| fix(typescript): 🐛 navigation prop | 5d5b9c5 |
| fix(eslint): 🐛 nested-components | ee643b0 |
| fix(typescript): 🐛 navigation | 08937d2 |

Table 2: Bugfix commits

Similarly for `feature` commits:

| Commit Message | Commit Hash |
| --- | --- |
| feat(mobile-client): ✨ usebooks query | 7e52fa7 |
| feat(mobile-client): ✨ dotenv | 9063958 |
| feat(server): ✨ book | 4a202ec |
| feat(server): ✨ database | 59fb876 |
| feat(localization): ✨ i18n support | 938f0a8 |
| feat(typescript): ✨ lower-level types | ccd3629 |
| feat(server): ✨ borrow | adb7eeb |

Table 3: Feature commits

For the development workflow, I have chosen `GitHub Flow`[8]as the branching strategy. The `GitHub Flow` is a lightweight branching strategy. The main purpose of this workflow is to keep the main branch deployable at all times, therefore to keep the mainline clean. To do that, every development work is done in a separate branch, which is branched off from the main branch. When the work is done, a pull request is created, and the changes are reviewed. In this repository, I have decided to use rebase[9]merge strategy. This means that the pull request is rebased on top of the main branch, and then merged into the main branch.

`git merge` and `git rebase` do the same thing, they bring the contents of two branches together. However, both of these commands execute this change, in entirely different ways. The main benefit of using rebase is to keep the commit history clean, and linear. This makes it easier to track down bugs, and to revert changes if necessary. By using rebase strategy it will be much easier to follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate the git history of the project.
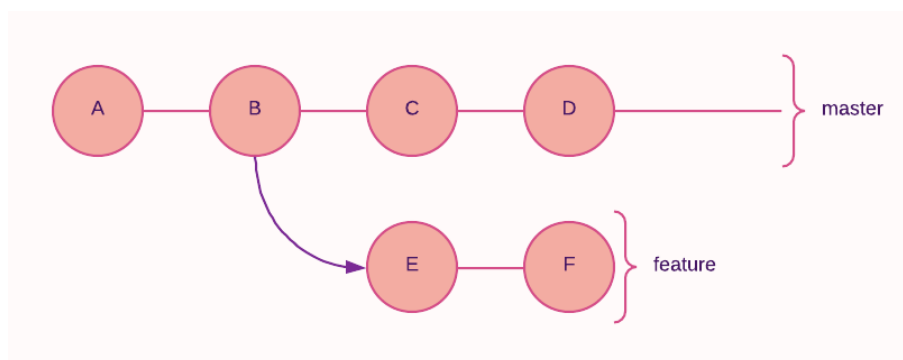


Figure 17: Before Git Rebase
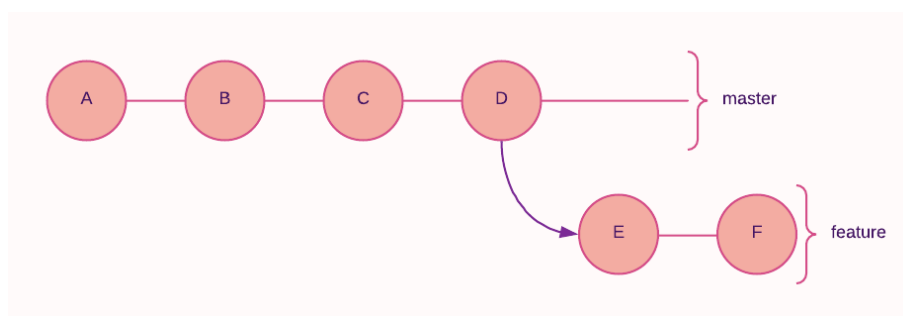


Figure 18: After Git Rebase

The commit messages that are being pushed into the git repository has to follow a specific set of rules, in order for the semantic release to be able to determine the next version of the application. The commit messages have to follow the `conventional commits`[10]specification. In this repository, I have used `commitlint`[11]to enforce the commit message format. Whenever a new commit is created, to prepare commit message git hook is being called, which prompts the user a CLI tool, to help format the commit message.

```
? Select the type of change that you're committing: Use arrow keys or type to search
❯ feat:     A new feature
  fix:      A bug fix
  docs:     Documentation only changes
  style:    Changes that do not affect the meaning of the code
  refactor: A code change that neither fixes a bug nor adds a feature
  perf:     A code change that improves performance
  test:     Adding missing tests or correcting existing tests
(Move up and down to reveal more choices)
```

Figure 19: Committizen CLI in action

Commitlint is a command line interface tool, which provides an interactive was of crafting commit messages, and it makes sure the result satisfies the conventional commit specification.

A good commit message is made up of four parts, the type, the scope, a subject (or a very short description) and a longer description. The type describes the kind of change that the commit is providing. The scope describes the part of the application that is being affected by the change. The subject describes the change itself, and it should be written in the imperative mood. The commit message should also be written in the present tense, and it should not end with a period.
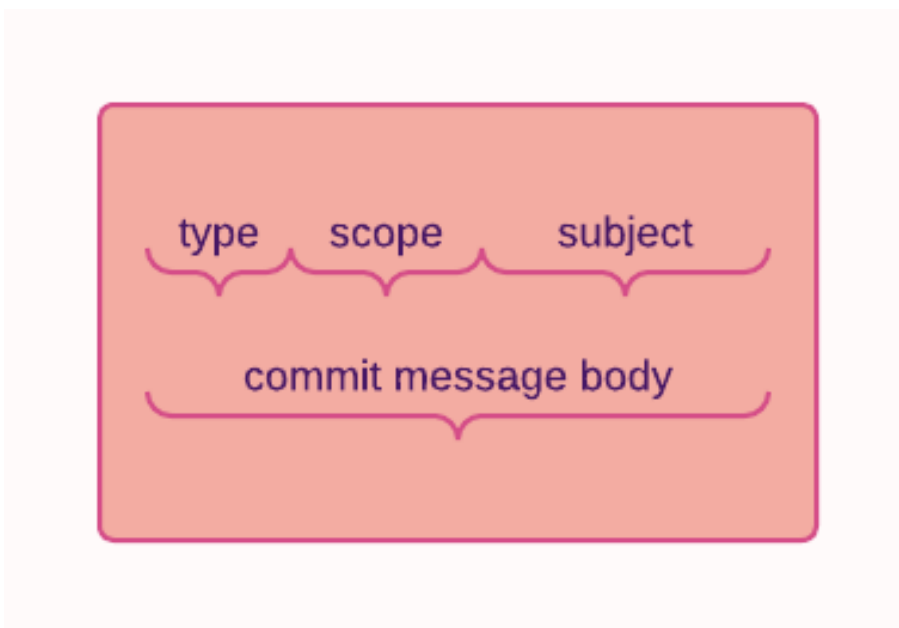


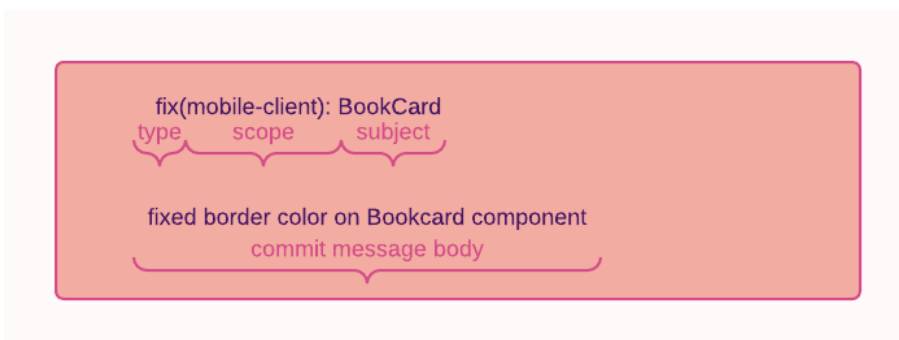Figure 20: Commit message tempalte



Figure 21: Commit message example

I have created an own set of scopes which suites the nature of this application, containing all the scopes which can determine a specific set of changes in the repository.

```
module.exports = {
  prompt: {
    scopes: [
      "repository",
      "mobile-client",
      "typescript",
      "server",
      "chore",
      "thesis",
    ],
  },
};
```
Listing 1: Commit Message Format

### 3.1.6  ESLint

`ESLint` is a static code analysis tool for `JavaScript` and `TypeScript`. It is a tool that helps developers to find problems in their code, and to enforce a specific coding style. This repository, both the frontend and the backend applications are using a custom `ESLint` configuration, which is placed under the `packages/eslint-config` folder. This configuration is behaved as an `npm` package, it has exported files defined in it `package.json` file.

```
"files:": [
    "plugins",
    "rules",
    "react.js",
    "nestjs.js"
],
```
Listing 2: Files exported from eslint-config package.json file

These are te files, which can be later on used to extend the `ESLint` configuration.

```
{
    "extends": "pikkukirjasto/react",
}
```
Listing 3: Configuring ESLint for the frontend application

Similarly, for the backend application:

```
{
    "extends": "pikkukirjasto/nestjs",
}
```
Listing 4: Configuring ESLint for the backend application

While both the frontend and the backend applications are written in `TypeScript`, they are running on different runtimes, therefore they require different `ESLint` configurations.

As the frontend application is written in `React`, the `ESLint` configuration is defined `eslintconfig/react`. The backend application is using the `NestJS` framework, so it requires a slightly different `ESLint` configuration, which is defined in `eslint-config/nestjs`.

These are pluggable `ESLint` configurations, so they can share some specific set of rules and plugins, which can be applied on both applications. The shared configurations are placed under the `eslint-config/rules/base.js` file, while the plugins which are shared between the configurations are placed under the `eslint-config/plugins/common.ts` file.

---

[3]en.wikipedia.org/wiki/Monorepo
[4]pnpm.io
[5]nx.dev
[6]github.com/npm
[7]docs.github.com/en/get-started/quickstart/github-flow
[8]git-scm.com/docs/git-rebase
[9]conventionalcommits.org
[10]commitlint.js.org/

## 3.2 Deployment

The application deployment is a crucial part of delivering every software to it's end users. In modern application development, the deployment process is automated, and it is part of the development process.

In this section I will introduce the deployment strategies I applied to release and deliver the application to the users.

### 3.2.1 Github Actions

The entire application, and it's related stacks and services are deployed using `Github` actions.

`Github` actions are workflows that are triggered by events, and they are defined in a `YAML` file within the repository. The main purpose of the `Github` actions is to automate the deployment process, and to make sure that the application is always in a deployable state.

In the `Pikkukirjasto` repository there are multiple features of `Github` actions utilized. It also runs tests, deploys static websites (documentation, codecov, etc.), and it also deploys the application to the production and staging environments.

The main workflow implemented in `Github` action is the `deployment` workflow. The `deployment` workflow is a reusable workflow, which has two different triggers, based on the environment being deployed.

- Pull request merge to the `staging` branch
- Pull request merge to the `release` branch

Reusable workflow is a strategy that is used to create a workflow that can be reused in multiple workflows. Utilizing reusable workflows helps to avoid code duplication in the workflow declaration, and it also makes it easier to maintain the workflows, and create new workflows.

The diagram below demonstrates the in-progress workflow that uses a reusable workflow in `Pikkukirjasto` repository.
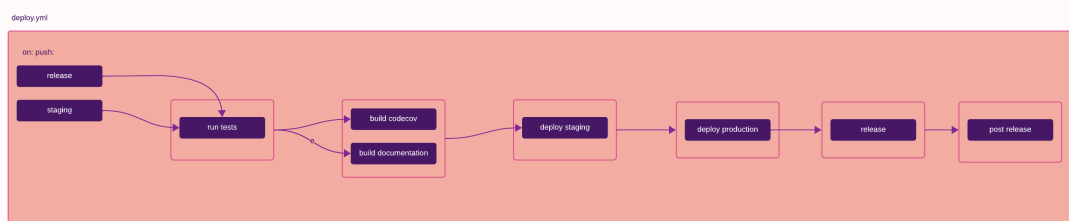


Figure 22: Github Actions

- After each of two build jobs on the left of the diagram completes successfully, a dependent job called `run tests` is run.
- The `deploy` job calls a reusable workflow that contains two jobs:

```
    - building codecov static site

    - building documentation static site
```

- The `deploy staging` job only runs after the previous static site deployment jobs has completed successfully, and it contains a reusable workflow.

- The `deploy production` job only runs after the previous deploy staging job has completed successfully, and the release target is `production`.

- The last two jobs are handling releases, and release related actions, for example creating a new version using `semantic release`, and creating a new release in `Github`, writing into the `changelog`, etc.

- Using a reusable workflow to run deployment jobs allows you to run those jobs for each build without duplicating code in workflows.

In this repository, there are dependent jobs utilized in the deployment pipeline, to make sure the subprocesses of the jobs are only running, if their parent job has completed successfully. One job can have multiple jobs which it depends on.
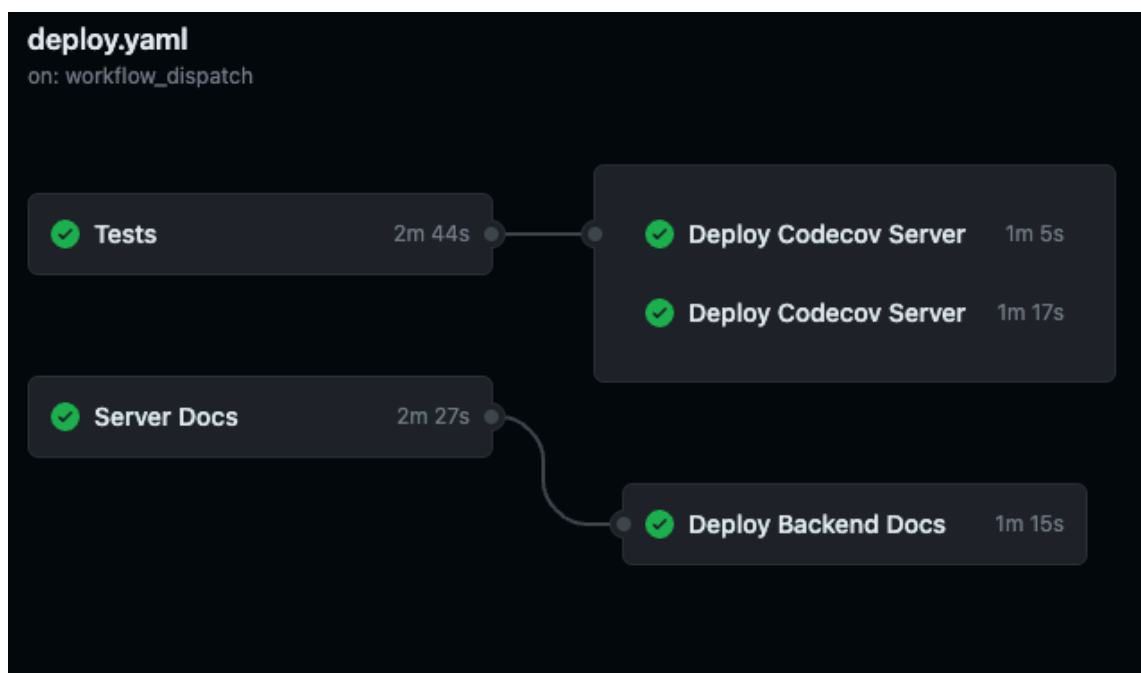


Figure 23: Github Deployment Workflow

The deployment pipeline is made up of multiple jobs, which are running in parallel, and they are dependent on each other. To make it more robust and easy to maintain, they are implemented by different `GitHub Actions`. Every action is responsible for a small part of the pipeline, which can be reused across workflows and jobs. I have utilized an action alongside the workflows, which handles every necessary steps a workflow needs to have, for example checking out the repository, setting up the environment, installing the `pnpm` dependencies, etc.

The `Github Workflows` are located in the `.github/workflows`, while the reusable actions are located in the `.github/actions` directory.

### 3.2.2 Release strategy

The release strategy of the application is connected to the nature of the `monorepo`, and the chosen branching strategy, therefore the chosen release strategy for this application is `semantic release`[12].

`Semantic release` is a tool that automates the versioning and publishing of the application. It uses the commit messages to determine the next version of the application. It also generates a changelog based on the commit messages. Semantic release also creates a git tag for the release, so the versions are easy to follow, and the releases are easy to revert if necessary. The changelog of a specific release contains the commit messages of the commits that are included in the release.
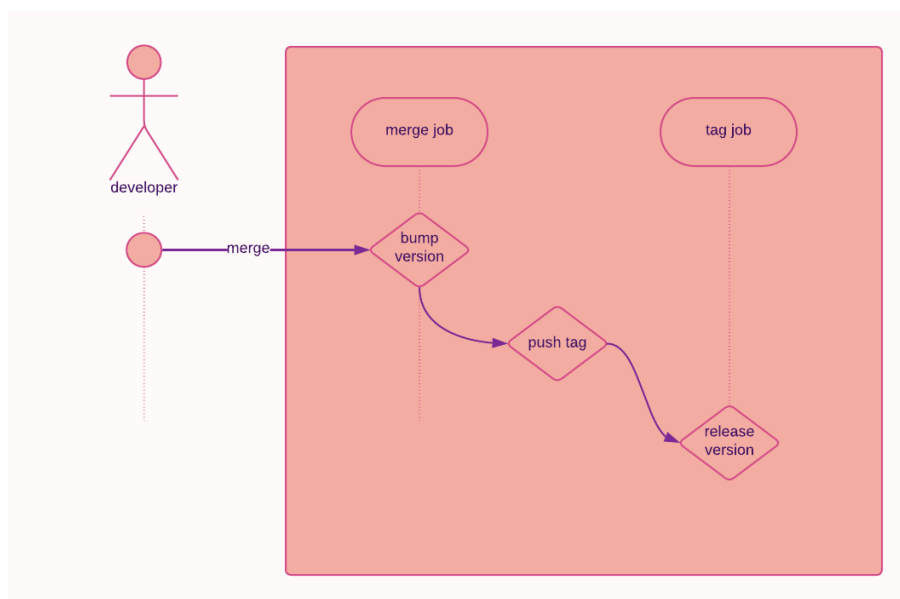


Figure 24: Semantic Release

In this application there are two different dedicated release channels. One for production release, and for staging release. Ultimately, the staging release is also a production candidate, so the semantic release job will tag the release accordingly. The release channels have their own branches.

`Release` branch for production releases, and `staging` branch for staging releases. The semantic release job is configured to run only on these branches. Whenever a new commit (merge or rebase commit) is pushed to the staging branch, semantic release will attempt to create a new staging release. The release branch is configured in a way, that it only accepts merge commits from the staging branch. This means that the staging branch is always ahead of the release branch, and the release branch is always behind of the master branch. This is the reason why the staging branch is always a production candidate.

Both the production, and the staging release creates different versions, where the release channel is visible on the release tag. For the production release, the release tag will be the new version number, for example `1.0.0`. For the staging release, the release tag will be the new version number, with the release channel, for example `1.0.0-staging.0`.

Whenever a commit is pushed to any of the release branches (on either channels), the dedicated GitHub action will create a new release, and it will deploy the application to the production environment, and as well creates a new changelog, which contains the changes the release contains, and the commit messages of the commits that are included in the release.

Semantic Release besides creating a `CHANGELOG` file, it also creates a release within the repository of the application. This release also contains the changelog, and the release tag.
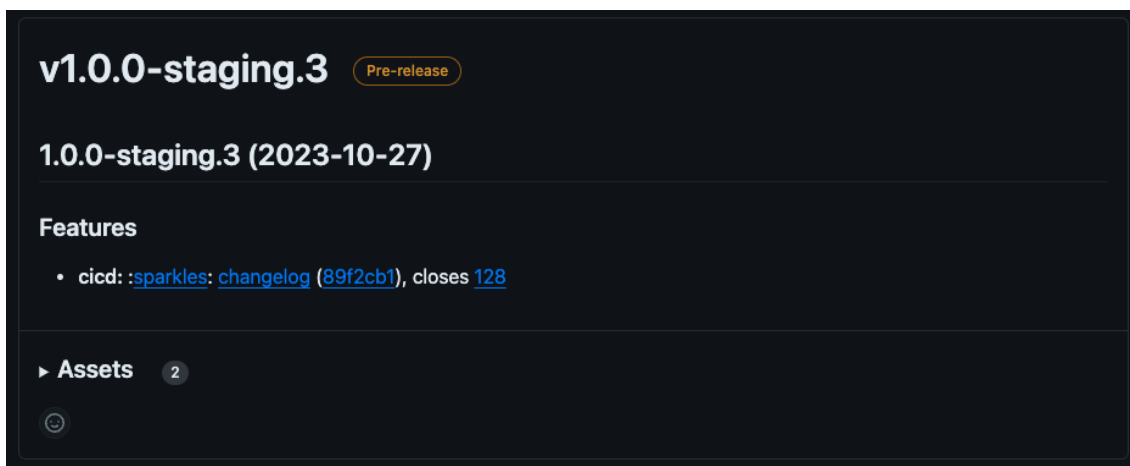


Figure 25: Release in Github

[11]github.com/semantic-release/semantic-release

## 4  Technologies

This application utilizes the most cutting-edge technologies available at the time of writing this thesis. Both the frontend and backend application is written in `TypeScript`, following the `ECMAScript 2020` standard. Both of codebase is strictly typed using `TypeScript`, which means that the code is checked for type errors during the build process, making sure to avoid runtime errors as much as possible.

### 4.1  TypeScript

`TypeScript`[13] is a statically typed superset of `JavaScript`, designed to enhance the development experience by providing strong typing and additional features for large-scale applications. By introducing static types, `TypeScript` enables developers to catch type-related errors during compile-time, reducing the risk of runtime errors and enhancing code reliability.

One of the main features of `TypeScript` is its support for static type checking. With type annotations and inference, developers can explicitly define the types of variables, function parameters, and return values. This not only improves code clarity but also provides intelligent auto-completion and type-aware refactorings in modern IDEs.

`TypeScript` also facilitates the use of modern `ECMAScript` features by allowing developers to target specific `ECMAScript` versions, ensuring compatibility across different environments. This feature is particularly useful when dealing with legacy codebases or targeting specific browser versions.

Furthermore, `TypeScript` offers powerful object-oriented programming capabilities, such as classes, interfaces, and inheritance, enabling developers to build scalable and maintainable codebases. The support for generics allows the creation of reusable and type-safe data structures and algorithms.

To ensure seamless integration with popular `JavaScript` libraries and frameworks, TypeScript supports declaration files (`.d.ts`), which provide type information for existing `JavaScript` code. This feature enables developers to leverage the rich ecosystem of `JavaScript` libraries while still enjoying the benefits of static typing.

In summary, `TypeScript` empowers developers to write more reliable and maintainable code by bringing static typing and advanced language features to the `JavaScript` ecosystem. Its ability to catch errors early, support modern `ECMAScript`, and foster robust object-oriented programming makes it a valuable tool for building large-scale applications and enhancing overall development productivity.

### 4.2  React Native

`React Native`[14] is a cutting-edge framework for building cross-platform mobile applications using `JavaScript` and React. At the core of React Native's architecture lies the concept of the UI thread and the JS thread, which play vital roles in rendering and handling the user interface.

In `React Native`, the UI thread is the main execution thread responsible for rendering the user interface components and responding to user interactions. This thread is separate from the

`JavaScript` thread to ensure that the app's UI remains responsive and doesn't freeze during intensive `JavaScript` computations. By offloading UI rendering to a separate thread, React Native achieves smooth performance and a delightful user experience.

The `JavaScript` thread, on the other hand, is responsible for executing the application's `JavaScript` code. This thread runs the React Native application logic, processes state changes, and handles data manipulation.

It communicates with the UI thread through a bridge, passing instructions and updates to be rendered on the screen. As `JavaScript` execution can sometimes be computationally intensive, `React Native` allows developers to optimize the performance by moving certain tasks to native modules written in `Java` (or `Kotlin`) for `Android` and `Objective-C` (or `Swift`) for `iOS`.

The renderer is a crucial part of `React Native`'s architecture. It acts as a bridge between the `JavaScript` thread and the native components. The renderer interprets the React components and translates them into native views for `iOS` and `Android` platforms. This process enables `React Native` to provide a truly native user interface experience while allowing developers to write code in `JavaScript`.

One of the key advantages of `React Native` is its ability to use a single codebase to target both `iOS` and `Android` platforms, thanks to its cross-platform nature. Developers can write the UI components and business logic once in `JavaScript` and then rely on the `React Native` renderer to handle the translation into native UI elements. This drastically reduces development time and effort, allowing for rapid iteration and code sharing across platforms.

Additionally, `React Native` embraces the concept of native modules, enabling developers to access native `APIs` and functionalities that may not be available out of the box in `React Native`'s core components. By creating native modules, developers can bridge the gap between the `JavaScript` and native environments, making it possible to leverage platform-specific features and third-party libraries seamlessly.

In conclusion, `React Native` is a groundbreaking framework that leverages the power of the UI thread and the `JavaScript` thread to deliver high-performance cross-platform mobile applications. The separation of concerns between these threads, along with the renderer and native modules, empowers developers to create engaging and responsive user interfaces while maximizing code reuse and productivity.

With its ever-growing community and continuous updates, `React Native` remains a top choice for mobile app development, enabling developers to build innovative, feature-rich, and performant applications on both `iOS` and `Android` platforms.

### 4.2.1  Expo

`Expo`[15] can be considered as a superset of `React Native`. It provides an `SDK` of many useful tools and libraries, which are not available in the bare `React Native` framework. Using `Expo` simplifies a lot of the development process, and makes it easier to build and deploy cross-platform mobile applications.

The `Expo EAS` (`Expo Application Services`) is a comprehensive set of tools and services

designed to streamline the development and deployment of mobile applications. It offers a unified platform for creating cross-platform apps using `JavaScript`, `React Native`, and other popular web technologies. `EAS` extends Expo's capabilities by providing advanced features for building, testing, and publishing apps, making the development process more efficient and accessible. With `EAS`, developers can seamlessly manage the entire app lifecycle, from coding and testing to building and distribution, while also benefiting from enhanced performance optimizations, real-time updates, and simplified maintenance. This integrated approach simplifies the complexities of mobile app development, allowing developers to focus on creating exceptional user experiences without getting bogged down by intricate technical details.
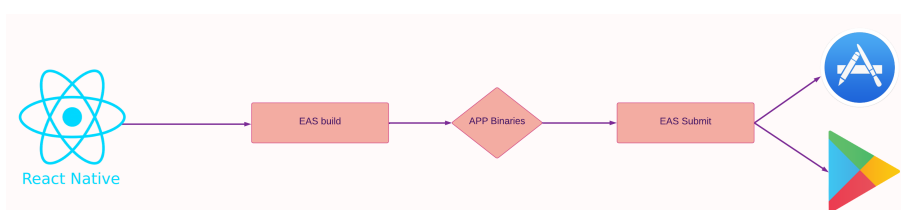


Figure 26: Expo EAS

## 4.3 NestJS

`NestJS`[16] is a powerful and modern `Node.js`[17] framework that has gained significant traction in the development community due to its robustness and support for building scalable, maintainable, and performant server-side applications. At the core of `NestJS` lies its extensive use of decorators, which play a crucial role in defining and structuring the application.

Decorators in `NestJS` are used to mark classes, methods, and properties, enabling the framework to identify and treat them as special entities. One of the fundamental decorators in `NestJS` is the `@Module`, which defines a module and serves as the building block of the application's architecture. Modules encapsulate related functionalities, allowing for a highly modular and organized codebase. Through the `@Module`, developers can import and export other modules, facilitate dependency injection, and manage the overall application flow.

In addition to modules, `NestJS` leverages the power of decorators to define controllers, the heart of handling incoming requests and generating responses. Controllers are adorned with the `@Controller` decorator, and individual routes within a controller are defined using the `@Get`, `@Post`, `@Put`, and other `HTTP` method decorators. By leveraging these decorators, developers can easily create `API` endpoints and manage different data interactions.

Moreover, `NestJS` promotes the use of `@Injectable` decorators to define services, the backbone of business logic and data manipulation within the application. Services are singleton instances that can be injected into controllers, allowing for efficient separation of concerns and facilitating code reusability. These decorators, along with the dependency injection system provided by `NestJS`, create a smooth workflow for handling complex data operations and maintaining a clean and testable codebase. `NestJS` also introduces custom decorators, which empower developers to create their own reusable and specific annotations. For instance, `@AuthGuard`, a custom decorator, can be implemented to enforce authentication and authorization on specific routes or controllers. By leveraging custom decorators, developers can tailor the application to their specific business needs and improve code

readability, and maintainability.

Beyond decorators, `NestJS` embraces the concept of models, also known as entities or `DTOs` (Data Transfer Objects). These models define the structure of data being transferred between various components of the application. By using models, developers can ensure strong typing and validation of incoming and outgoing data, minimizing runtime errors and enhancing the overall application's reliability.

In conclusion, `NestJS` stands out as a modern and technically advanced `Node.js` framework, primarily due to its clever use of decorators, which provide a structured and intuitive approach to building server-side applications. With its focus on modularity, dependency injection, and powerful custom decorators, `NestJS` empowers developers to create scalable and maintainable codebases, making it an excellent choice for building complex applications in the ever-evolving world of `JavaScript` and `TypeScript` development.
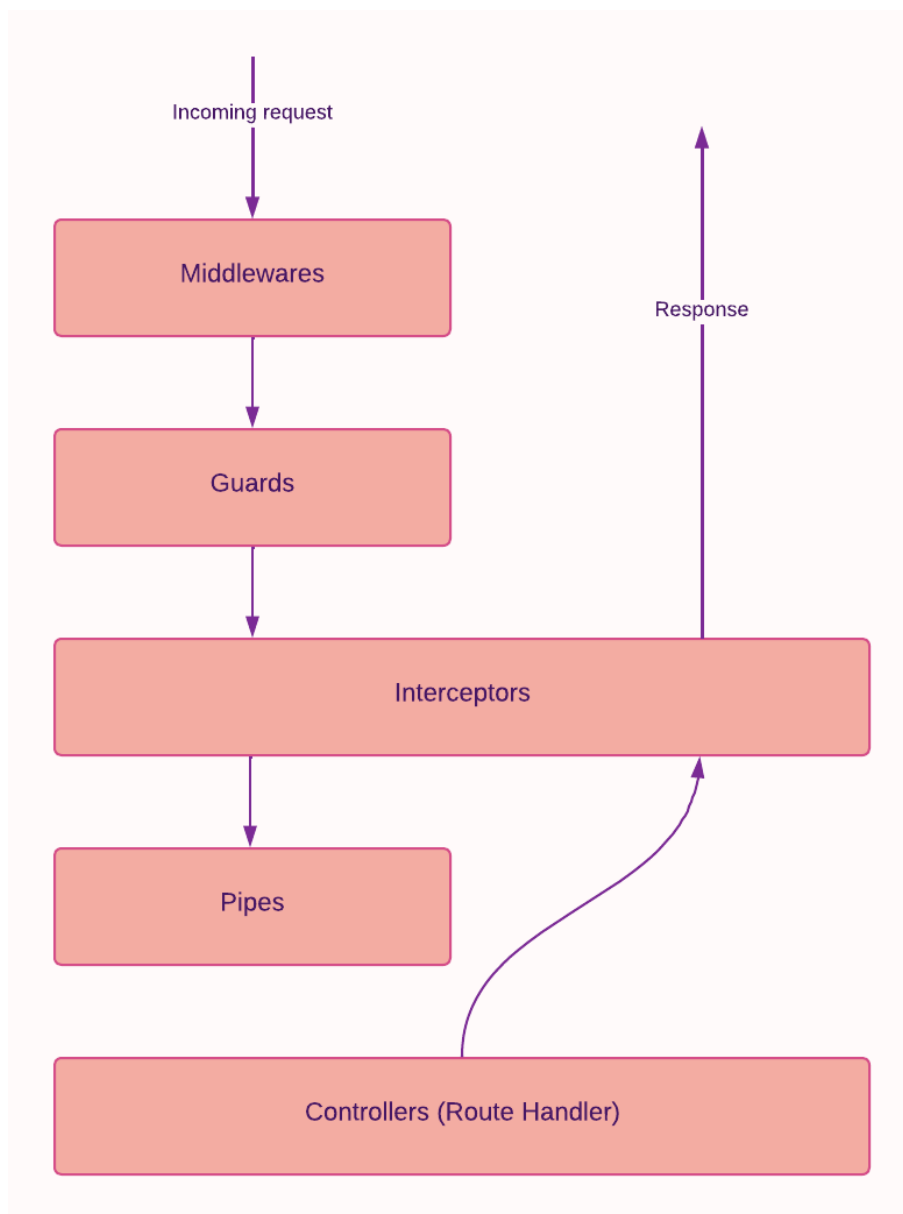


Figure 27: NestJS Routing

In `NestJS`, the handling of requests and responses follows a structured and modular approach, utilizing various key components such as middlewares, guards, interceptors, pipes, and controllers.

Middlewares are utilized to intercept and process incoming requests before they reach the controller, enabling tasks like logging, authentication, and data transformation.

Guards focus on route protection, allowing or denying access based on specific conditions.

Interceptors operate on both incoming requests and outgoing responses, enabling global transformations and validations.

Pipes are responsible for data validation, transformation, and sanitization, ensuring that the data received and sent is accurate and safe.

Lastly, controllers define the endpoints and route handlers, where the actual request processing takes place. They receive input from the incoming request, which has been pre-processed by the aforementioned components, and formulate the appropriate response. By strategically combining these components, `NestJS` offers a highly organized and extensible framework for managing the flow of data through an application, promoting code reusability, maintainability, and overall system robustness.

## 4.4 Database

The heart of this application lies in its robust database architecture, leveraging the power of `MongoDB` as the database engine and `Mongoose` as the Object Document Mapper (`ODM`). This combination provides a flexible and scalable solution to handle data storage and retrieval. `MongoDB`, being a document-oriented database, offers a schema-less data model, where information is stored in documents composed of flexible and dynamic `JSON`-like structures. Unlike traditional relational databases, `MongoDB` make it possible to store varying and evolving data structures without the constraints of fixed schemas, which can be especially advantageous for applications dealing with constantly changing data.

With `Mongoose` as the `ODM`, creates the ability to define schemas for the application data, providing a structured and organized way to model the documents. These schemas help maintain consistency, data integrity, and validation rules for the data stored in the database. By leveraging `TypeScript` support, `Mongoose` empowers it with strong typing and code completion, enhancing code quality and reducing the likelihood of runtime errors.

To ensure optimal performance and data access, the schema and indexes of MongoDB collections designed carefully, based on the queries and use cases. By utilizing the aggregation framework and other powerful features of `MongoDB`, makes it possible to efficiently retrieve, process, and analyze large datasets, enabling complex data operations to be performed with ease.

As the application scales, `MongoDB` provides built-in horizontal scaling through sharding, distributing data across multiple nodes to handle increased traffic and data volume. This sharding capability allows maintaining high availability, fault tolerance, and performance while handling the demands of a growing user base. Data security is of paramount importance, and to safeguard our database, authentication and access control mechanisms are also

implemented. By enforcing role-based access control (`RBAC`), to ensure that only authorized users can access specific data resources, protecting sensitive information and mitigating security risks.

## 4.5   AWS & CDK

The backend application, and a couple of other miscellaneous services and applications are deployed on `AWS` using `CDK`.

AWS CDK[18] (`Cloud Development Kit`) is a powerful infrastructure-as-code framework that allows developers to define and provision `AWS` resources using familiar programming languages, such as `TypeScript`, which this application stack also uses. With `CDK`, developers can model their cloud infrastructure as code, leveraging the full power of their chosen programming language to create, manage, and update `AWS` resources programmatically.

One of the key features of `AWS CDK` is its rich library of constructs, which represent `AWS` resources and services as objects in code. These constructs provide a high-level, typed abstraction over `AWS CloudFormation`, allowing developers to express their infrastructure in a concise and intuitive manner. `CDK` constructs are organized in a hierarchical structure, promoting modularity, reusability, and best practices for resource composition. `CDK` leverages the strength of modern programming languages, enabling developers to use loops, conditionals, and other control structures to define complex cloud environments programmatically. This declarative approach allows developers to express their infrastructure requirements in a familiar coding paradigm, making it easier to reason about, test, and maintain the codebase.

Another notable feature of `AWS CDK` is its support for multi-account and multi-region deployments. Developers can define cross-account and cross-region resource references and policies directly in code, simplifying the process of setting up secure and scalable distributed architectures.

Additionally, `CDK` provides a comprehensive set of `AWS` constructs for commonly used patterns, such as `VPCs`, `load balancers`, `Lambda functions`, and more. This rich set of abstractions reduces the boilerplate code and accelerates the development of cloud-native applications.

In summary, `AWS CDK` is a game-changer for cloud infrastructure development, as it enables developers to utilize their existing programming skills to define, deploy, and manage `AWS` resources efficiently. With its powerful constructs, multi-account support, and extensive library, `CDK` streamlines the process of building scalable and resilient cloud architectures while promoting code reusability and maintainability. As a result, `CDK` empowers developers to adopt infrastructure-as-code best practices and optimize their `AWS` deployments for greater productivity and faster development.

Defining underlying infrastructure using `AWS CDK` (or other tools, such us `Terraform`) is now the industrial standard method. It is a lot more reliable, and easier to maintain than manually creating resources on the `AWS` console. The engineering community is moving towards this direction, and it is important to keep up with the latest trends and best practices. The era of

manual deployment, and VPS servers are over, and it is time to move on.

---

[12]typescriptlang.org
[13]reactnative.dev
[14]expo.io
[15]nestjs.com
[16]nodejs.org/en
[17]aws.amazon.com/cdk

## 5 Implementation

In this section I am going to throughly describe the implementatinon of the application. The section is divided into two parts, the mobile application, and the backend implementation.

### 5.1 AWS Stack

As mentioned in the previous section, the entire application stack is deployed on `AWS`. The stack definition is written in code, using `AWS CDK`. There are two different `CDK` stacks implemented:

- `BackendStack` - contains all the backend resources, including the `DynamoDB` tables, `Lambda` functions, `API Gateway` endpoints, and `MongoDB`, to store application data.

- `StatickStack` - contains the `S3` bucket for hosting static sites, and the `CloudFront` distribution for serving the application.
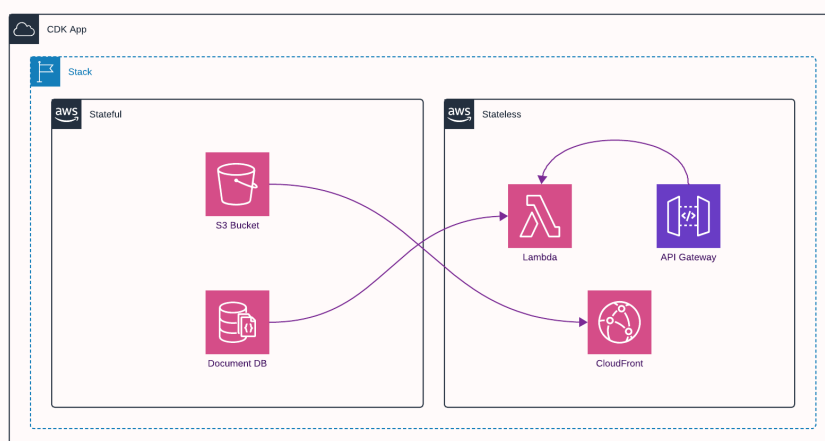


Figure 28: AWS CDK Stack

The semantics above describes the basic way the Pikkukirjasto application's stack.

In an `AWS Cloud Development Kit (CDK)` stack, stateful and stateless services play distinct roles in the architecture, offering different characteristics to cater to various application requirements. Stateful services, such as `Amazon RDS (Relational Database Service)` and `Amazon ElastiCache`, maintain persistent data across requests and maintain their internal state. These services are suitable for applications that require data durability, consistency, and complex querying capabilities.

On the other hand, stateless services, like `Amazon EC2 (Elastic Compute Cloud)` instances and `Amazon API Gateway`, do not store session-specific data and treat each request independently. They are ideal for horizontally scalable architectures, enabling seamless deployment, auto-scaling, and high availability. By strategically combining stateful and stateless services within a `CDK` stack, developers can design robust and flexible systems that meet specific application demands, while leveraging the benefits of both state persistence and agility.

### 5.1.1  Backend Stack

The backend stack is the main stack of the application, and it contains all the resources needed
to run the server side application. The application runs on `AWS Lambda`, and it is exposed
through `AWS API Gateway`. The `Lambda` function has a direct access to the `MongoDB` database.
The `CDK` stack is orchestrated by `AWS CodePipeline`, which is triggered by a `GitHub` webhook,
whenever a new commit is pushed to the `main` branch. There are multiple staged introduce in
the stack, to make it possible to have different stages for different environments, like
`staging`, and `production`, which is the corresponds to the release channels of the application.

```
type PikkuKirjastoEnvironment = "dev" | "prod";
```

Listing 5: CDK Stack Environment

```
const appConfig: PikkurKirjastoAppConfig = {
  dev: {
    name: `${PIKKURKIRJASTO}-Dev`,
  },
  prod: {
    name: `${PIKKURKIRJASTO}-Prod`,
  },
};
```

Listing 6: CDK App Config

Using the `PikkuKirjastoEnvironment` type, it is now possible to define the different
environments, and the `CDK` stack will create the resources accordingly.

Both the `dev` and the `production` stacks can be initialized from the main stack configuration,
using the `CDK` context.

```
new PikkuKirjastoStack(pikkurKirjastoApp, appConfig["dev"].name, "dev", {
    stackName: appConfig["dev"].name,
    env: {
        region: "eu-west-1",
    },
});
```

Listing 7: Stack initialization

The following semantics shows the different `AWS` resources being created in the `CDK` stack
definitions for the server application, and also showcases the relation between them.

When a user uses the mobile application, `HTTP` requests are made to the `API Gateway`
endpoint, which is connected to the `Lambda` function. If a request is required to fetch data
from the application's database, the `Lambda` function will query the `MongoDB` database, and
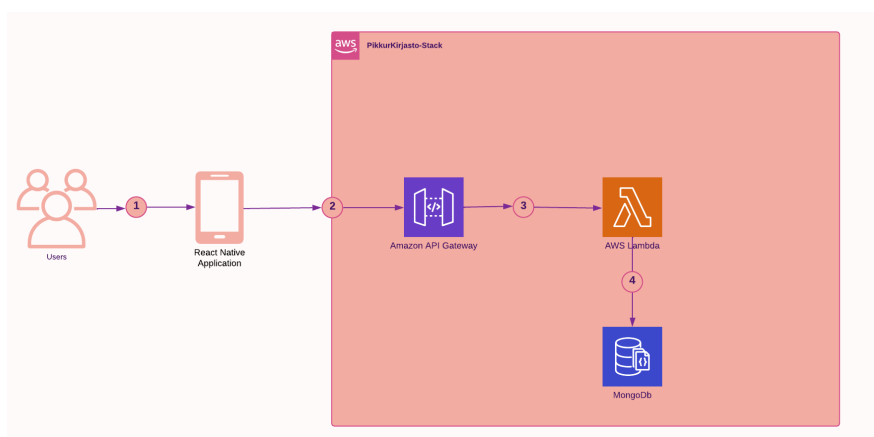return the result to the user.

Figure 29: Backend Stack

The semantics above represents an `AWS CDK` stack for the serverless application `"Pikkukirjasto"` which includes `Lambda` functions and `API Gateway`.

The `PikkuKirjastoStack` class extends the `Stack` class, which is a fundamental construct in `AWS CDK` representing an `CDK CloudFormation` stack. This class will be responsible for creating and managing the `AWS` resources needed for the serverless application.

In the constructor of the `PikkuKirjastoStack` class, there are four parameters: `scope`, `id`, `env`, and `props`. The `scope` defines the scope of the stack, and here it's set to the entire stack. The `id` is a unique identifier for the stack. The `env` parameter is of type `PikkuKirjastoEnvironment`, which represents the environment (either `"dev"` or `"prod"`) for the application. The `props` parameter contains optional properties for the stack, which are passed down from the parent class.

The code then creates a backend `Lambda` function named `BackendLambda` using the `LambdaFunction` class. The function is written in `Nodejs 18x` and has a handler named `main.handler`. The code for the `Lambda` function is sourced from an asset in the `appsserverbuild/dist` directory, excluding the `node_modules` folder. The function has a timeout of 15 seconds.

Next, an `API Gateway` is created using the `LambdaRestApi` class. The `API Gateway` serves as the front-end for the serverless application, forwarding requests to the backend `Lambda` function. The `API Gateway` is configured with `CORS (Cross-Origin Resource Sharing)` options to allow requests from all origins with specific headers. The endpoint configuration is set to `REGIONAL` type. The `API Gateway` is configured to not use a `proxy` and not create a `CloudWatch` role.

The code then defines an integration between the `API Gateway` and the backend `"proxy+"` function using the `LambdaIntegration` class. The content handling is set to `CONVERT_TO_BINARY`.

A resource is added to the `API Gateway`, mapped to the `"proxy+"` path, and configured to use the backend `Lambda` integration for handling requests. The authorization type is set to `NONE`, meaning no authentication is required for accessing the `API Gateway`.

Finally, the stack defines a `CloudFormation` output named `backendLambdaArn`, which provides the `ARN` (`Amazon Resource Name`) of the backend `Lambda` function. The output is exported with the name `props?.stackName ?? id:backendLambdaArn`.

In summary, this `AWS CDK` stack sets up the infrastructure for the `"Pikkukirjasto"` serverless application, including a backend `Lambda` function and an `API Gateway` to handle and route requests. The application is designed to be deployed in either a `"dev"` or `"prod"` environment, as specified by the `env` parameter.

### 5.1.2 Static stack

The `Static` stack is used to host the static sites of the application. The stack contains an `S3` bucket, which is used to store the static files, and a `CloudFront` distribution, which is used to serve these static files. This stack is used to host different miscellaneous applications, such us `codecov` reports, documentations, etc.

The static stack is rather different from the `backend` stack. It does not contain `Lambda` functions, nor `API Gateway` endpoints, because it does not need to serve any `HTTP` requests.

The most important part of the `static` stack is the root and the subdomains where the static sites are going to be deployed.

```
export class StaticStack extends Stack {
  constructor(
    scope: Construct,
    id: string,
    domains: string[],
    props: StackProps
  ) {
    super(scope, id, props);
```

Listing 8: Stack initialization

As the `constructor` shows, the `domain` parameter is an array of strings, which will be used to build up the final domain name.

For example, if the array passed to the `StaticStack` is `["codecov", "app"]`, the final domain name will be `backendcov.app.thesis.richardzilahi.hu`.
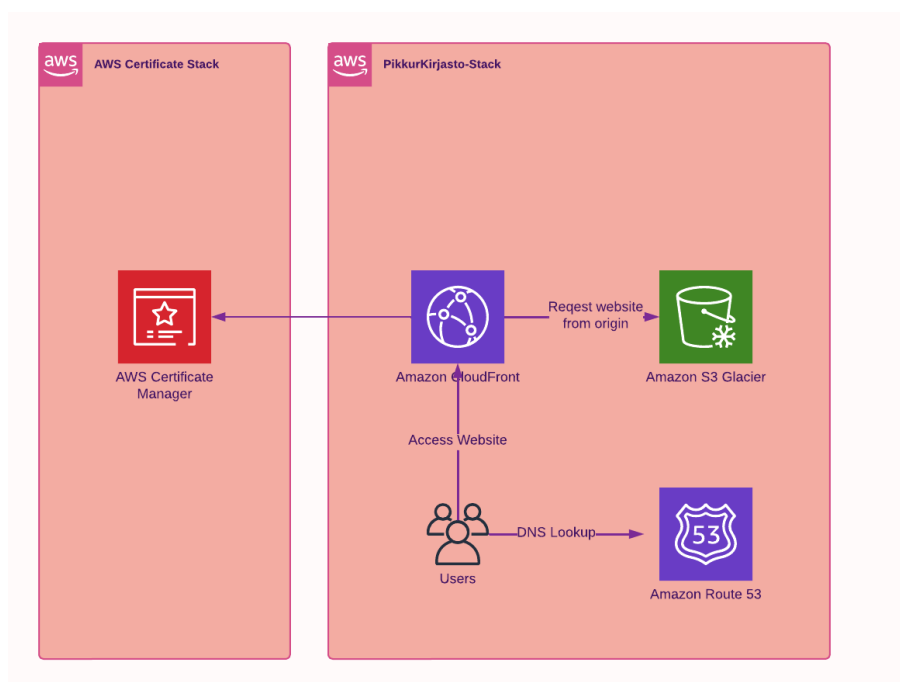
Figure 30: Static Stack

The semantics above represents an `AWS CDK` stack for creating static sites. It leverages various AWS services to set up the necessary infrastructure.

The `StaticStack` class extends the `Stack` class, which is a fundamental construct in AWS CDK representing an AWS CloudFormation stack. This class will be responsible for creating and managing the AWS resources needed for the static site.

In the constructor of the `StaticStack` class, there are four parameters: `scope`, `id`, `domains`, and `props`. The `scope` defines the scope of the stack, and here it's set to the entire stack. The `id` is a unique identifier for the stack. The `domains` parameter is an array of domain names for the static site, and multiple domain names can be provided and joined with dots. The `props` parameter contains optional properties for the stack, which are passed down from the parent class.

The code first creates an S3 bucket to store the static site content. The bucket is configured with `S3.BlockPublicAccess.BLOCK_ALL`, which ensures that the bucket is not publicly accessible. Additionally, `versioned:false` is set to disable versioning for the bucket. Next, the code sets up a `CloudFront` distribution, which serves as the front-end for the static site. `CloudFront` is a content delivery network (`CDN`) that helps improve the performance and speed of delivering content to end-users. The distribution is associated with a `SSL` certificate obtained from `AWS Certificate Manager` (`ACM`) using the provided `ARN`, enabling HTTPS support for the site.

The `CloudFront` distribution is configured with a default behavior, forwarding all requests to the `S3` bucket created earlier. The viewer protocol policy is set to `REDIRECT_TO_HTTPS`, ensuring that all requests are redirected to the `HTTPS` version of the site for improved security.

To handle errors gracefully, the code sets up error responses for the `CloudFront` distribution. If a user encounters a `403` or `404` error (forbidden or not found), the distribution will return

the `index.html` page instead of the default error page.

For secure access to the `S3` bucket from the `CloudFront` distribution, a `CloudFront Origin Access Identity` (`OAI`) is created. This identity helps control access and prevents unauthorized direct access to the bucket.

An `IAM` policy is also defined, specifying the permissions allowed for the `CloudFront OAI`. It allows the `OAI` to list objects and get objects from the `S3` bucket.

The code performs a lookup for the `Route53` hosted zone associated with the domain name `thesis.richardzilahi.hu`.

Finally, the code creates a `Route53 A record` in the hosted zone, mapping the provided domain names to the `CloudFront` distribution. This enables users to access the static site using the custom domain.
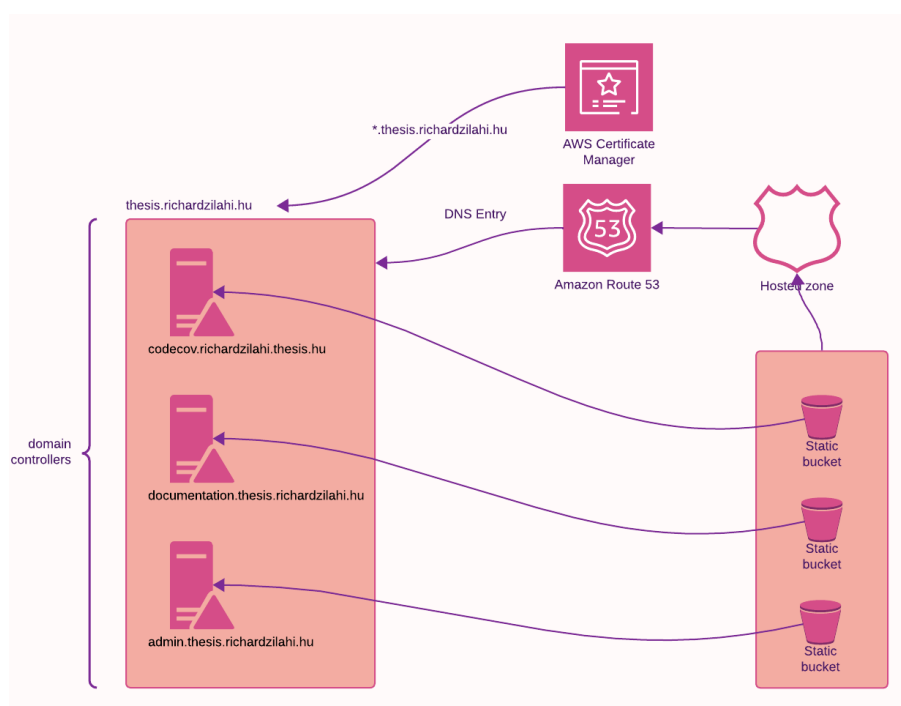


Figure 31: AWS Route 53

The stack defines two `CloudFormation` outputs: `static-url-id` and `static-bucket-id`. The former provides the `CloudFront` domain name that users can use to access the static site, while the latter provides the `ARN` of the `S3` bucket where the static site content is stored.

In summary, this `AWS CDK` stack sets up the infrastructure for hosting a static site, utilizing `S3` for storage, `CloudFront` for content delivery, `Route53` for custom domain mapping, and `IAM` for secure access control.

## 5.2   Mobile Application

The mobile application is written in `React Native`, and it is using `Expo` to build and deploy the application. Both technologies had been introduced and discussed in the previous section. In this section I am going to introduce the implementation details of the mobile application.
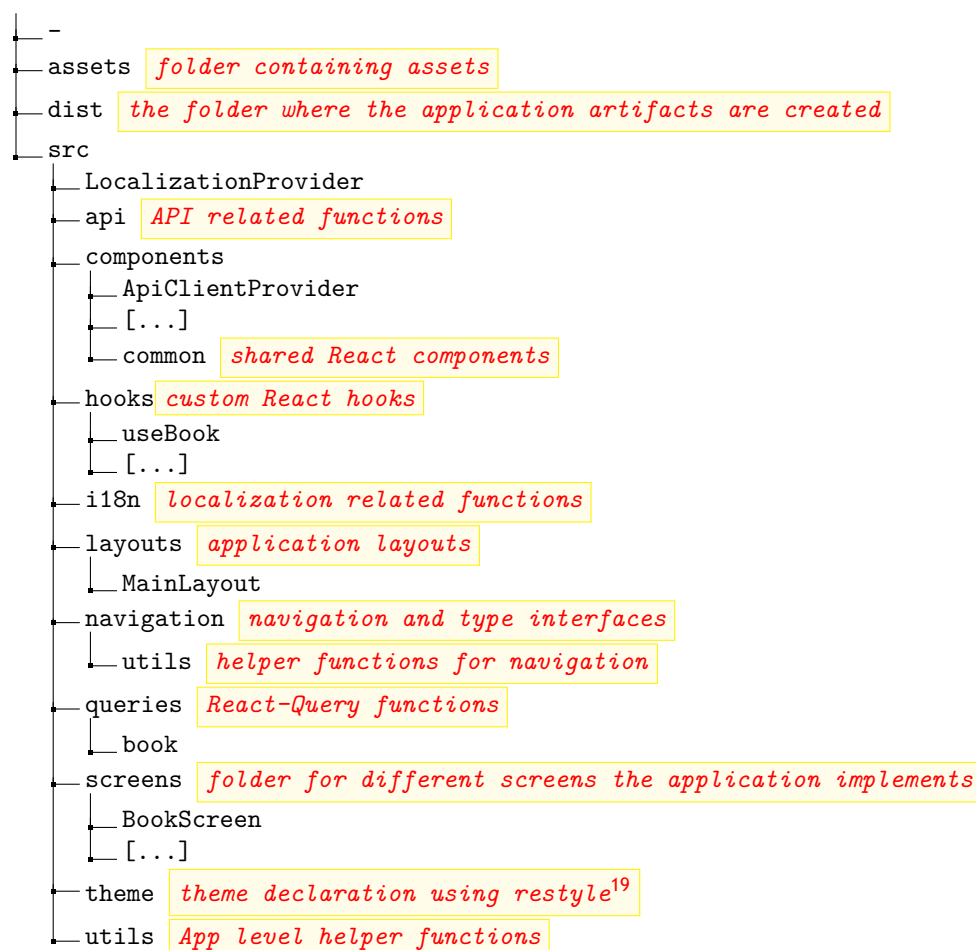
### 5.2.1   Application structure

There are multiple different ways to structure a `React Native` (or `React`) application. This is generally a subjective topic among developers, but there are certainly good and bad practices when it comes to deciding how to structure the source code of an application.

Choosing, and maintaining a good structure for the source code is crucial for the success of the application. A good structure can help developers to understand the codebase better, and it can also help to maintain the application in the long run. The better the structure is, the easier it for it's developers to have an overall understanding of the application, and it has a significant impact on the productivity.

My approach can be best described as both `modular` and `component` based structure.

In this mobile application, the structure of the source code looks as the follows:

```
├── -
├── assets  folder containing assets
├── dist  the folder where the application artifacts are created
└── src
    ├── LocalizationProvider
    ├── api  API related functions
    ├── components
    │   ├── ApiClientProvider
    │   ├── [...]
    │   └── common  shared React components
    ├── hooks  custom React hooks
    │   ├── useBook
    │   └── [...]
    ├── i18n  localization related functions
    ├── layouts  application layouts
    │   └── MainLayout
    ├── navigation  navigation and type interfaces
    │   └── utils  helper functions for navigation
    ├── queries  React-Query functions
    │   └── book
    ├── screens  folder for different screens the application implements
    │   ├── BookScreen
    │   └── [...]
    ├── theme  theme declaration using restyle[19]
    └── utils  App level helper functions
```

### 5.2.2   Design & UI

While `React Native` provides a powerful framework for building cross-platform mobile applications, it does not include any built-in UI components. Developers must either create their own components or use third-party libraries to implement the UI of their apps.

When building mobile applications, the consistency and quality of the UI are crucial for delivering a great user experience. However, designing and implementing UI components from scratch can be a time-consuming and challenging task, especially for complex applications

with multiple screens and navigation flows.

In the Pikkukirjasto application I have decided to implement my own UI system, using the `restyle` library.

The `Restyle` library is maintained by `Shopify`, which provides a powerful, type-enforced system for building UI system in `React Native` with `Typescript`.

`Restyle` needs a theme as a base, which will then be propagated into reusable components, derived from `Restyle` itself.

The `theme Restyle` implements the fundamental building blocks of the application's UI, such as colors, spacing, typography, and layout. It also provides a set of reusable components that can be used to build screens and layouts.

```
const theme = createTheme({
  colors: {},
  spacing: {},
  breakpoints: {},
  textVariants: {},
  cardVariants: {},
  ...
})
```

Listing 9: Theme declaration

Besides the `theme` declaration, `Restyle` also provides a set of primitives, which can be used on top of the theme, to define the basic components of the application.

```
import {createBox} from '@shopify/restyle';
import {Theme} from './theme';

const Box = createBox<Theme>();

export default Box;
```

Listing 10: Restyle primitives

Every component which is declared by `Restyle` is type safe, and it can be extended with additional props, which are also type safe.

Using `Restyle` and the `theme` makes it possible to create `variants` of components.

```
import {createText, createTheme, createVariant} from '@shopify/restyle';

const theme = createTheme({
  textVariants: {
    bookTitle: { fontSize: 20 }
  }
})

const Text = createText<Theme>();

export default Text;
```

Listing 11: Text component

Then the `variants` of the `Text` component can be used as follows:

```
<Text variant="bookTitle">Book Title</Text>
```

Listing 12: Text variant

### 5.2.3 Navigation

`React Native Navigation` provides a powerful solution for creating complex and efficient navigation structures in React Native apps through the use of nested navigators. The nested navigators system allows developers to compose different types of navigators (such as Stack, Tab, Drawer) within one another, enabling more granular control over the navigation flow and user experience.

At the core of `React Native Navigation`, there is a "NavigationContainer" component, serving as the root of the navigation hierarchy. Inside this container, developers can nest various navigators, creating a tree-like structure where each navigator manages its own set of screens. This design allows for easy organization and separation of concerns in large-scale applications.

One of the key advantages of nested navigators is the ability to customize navigation for different sections of the app independently. For instance, a `StackNavigator` can be used to manage a linear flow of screens, while a `TabNavigator` can handle a set of screens with tabs. By combining these within a DrawerNavigator, you can implement a fully featured app with multiple navigation paradigms.

Furthermore, nested navigators allow for better performance and memory management. When a screen is not in the active stack, it gets unmounted, freeing up resources. This makes `React Native Navigation` more efficient for handling complex navigation structures compared to a single, monolithic navigator.

To achieve this seamless nested navigation experience, `React Native Navigation` utilizes native navigation components provided by the platform (e.g., `UINavigationController` for `iOS` and `Fragment` for `Android`). This enables developers to leverage the native capabilities and gestures, resulting in a smoother user experience and faster transitions between screens.

Another significant advantage of `React Native Navigation` is its support for deep linking and universal links, making it easier to handle deep navigation directly from external sources like URLs or push notifications.

`React Native Navigation`'s nested navigators system empowers developers to build sophisticated navigation flows with optimal performance and a native-like experience. The ability to nest different types of navigators enables flexibility in handling various UI paradigms and helps maintain a clean and modular codebase, making it an excellent choice for complex `React Native` applications.

In addition to its navigational power, `React Native Navigation` also offers strong type safety and support for generics, enhancing the developer experience and reducing potential runtime errors. With `TypeScript` integration, developers can define strict types for their navigation props, ensuring that only valid routes and parameters can be accessed within the app. This helps catch errors during development and provides better autocompletion and documentation for navigation-related code.

`React Native Navigation` also introduces the concept of composite screen types, which allows developers to define screens that accept different sets of navigation props based on their position within the navigator hierarchy. For example, a screen nested within a `TabNavigator` may receive different navigation props compared to a screen within a `StackNavigator`. This dynamic typing ensures that screens receive the appropriate props, making the codebase more robust and easier to maintain.

Moreover, `React Native Navigation` supports generics, enabling developers to create reusable navigation components and hooks that work with different types of navigators and route configurations. This encourages code consistency and abstraction, reducing duplication and improving overall code quality.

By embracing type safety, generics, and composite screen types, `React Native Navigation` promotes better coding practices and streamlines the development process. Developers can have confidence in their navigation-related code, reduce the chances of bugs, and enjoy the benefits of improved productivity and maintainability in their `React Native` projects.

In the `Pikkukirjasto` application, the navigation uses this `React Native Navigation` library, and it implements three different navigators.

- `BottomTabNavigator` - the tab navigator, which is presented at the bottom of the screen, and it contains the screens related to the application's main functionality.

- `ProfileNavigator` - a stack navigator which implements the screens related to the user, and the user profile.

- `RootNavigator` - the main navigator which contains both the `BottomTabNavigator` and the `ProfileNavigator`.

Stack navigator is a type of navigator that provides a way for your app to transition between screens where each new screen is placed on top of a stack.
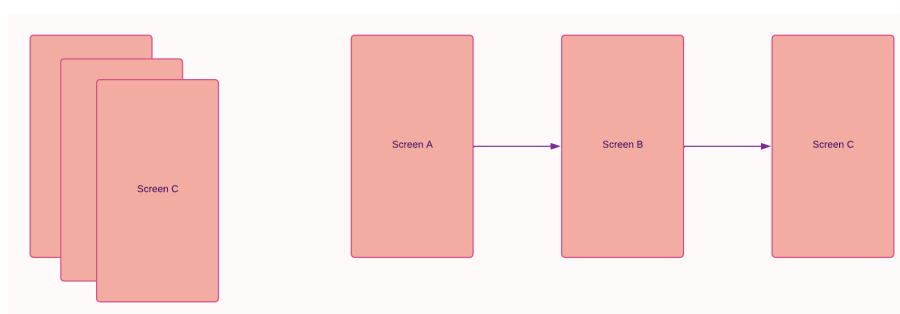


Figure 32: React Navigation - Navigate Forward

Navigating to a screen will push it onto the stack, the topmost screen on the stack is visible to the user.
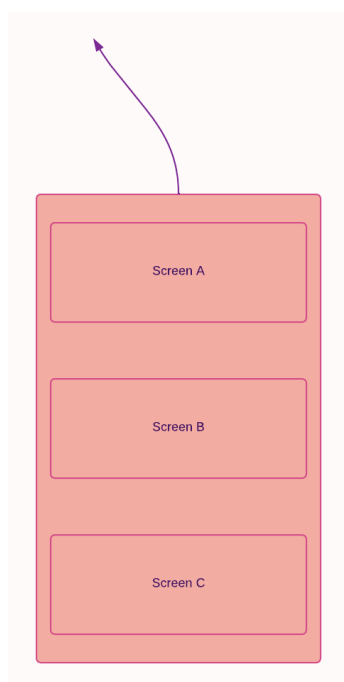
Figure 33: React Navigation - Navigate Back

Navigating back will pop the topmost screen off the stack and navigate to the screen below it.

In the `Pikkukirjasto` application the `RootNavigator` implements 3 different screens, where every screen implements another navigator. This is a common pattern in `React Native` applications, and it's called `nested navigators`.

```
const Stack = createStackNavigator();
<Stack.Screen
    name="Root"
    component={BottomTabNavigator} // this is a tab navigator
/>
<Stack.Screen
    name="Profile"
    component={ProfileNavigator} // this is another stack navigator
/>
```

Listing 13: React Native Root Stacknavigator

This structure of the navigators make sure that the navigation is consistent throughout the application, only certain screens are available in certain navigators.

To exploit the biggest advantage of `React Native Navigation`, the type safety, it's important to properly define the types for each navigator, and the screens they implement.

To make a navigator type safe, the only thing required is to pass an `Type` when creating a navigator, with the screens.

```
const BottomTab = createBottomTabNavigator<BottomTabParamList>();
```

Listing 14: Type safe navigator

where `BottomTabParamList` is a type, which defines the screens the navigator implements:

```
type BottomTabParamList = {
```

```
  Search: undefined;
  Library: undefined;
  Profile: undefined;
  Trending: undefined;
};
```
Listing 15: BottomTabParamList type

As the `Type` defines above, the `BottomTabNavigator` implements four screens:

- `Search`

- `Library`

- `Profile`

- `Trending`

Since the `BottomTabNavigator` is technically a screen on the `RootNavigator`, it also needs to be defined in the `RootStackParamList`:

```
type RootStackParamList = {
  Root: undefined;
  Book: { book: Book };
  BorrowBook: { book: Book };
};
```
Listing 16: RootStackParamList type

To make the entire navigation type safe, a `CombinedStackParamList` type is needed, which merges the two types into a single type:

```
type CombinedStackParamList = RootStackParamList & BottomTabParamList;
```
Listing 17: CombinedStackParamList type

In `React Native`, every screen receives a `navigation` prop, which contains the `navigation` functions, and the `route` prop, which contains the `params` passed to the screen. It's crucia to make these type safe also, so the navigation prop knows exactly which screens exists on that specific navigator.

To achieve this, the following `Type` needed to be defined:

```
type RootTabScreenProps<Screen extends keyof CombinedStackParamList> =
  CompositeScreenProps<
    BottomTabScreenProps<CombinedStackParamList, Screen>,
    NativeStackScreenProps<RootStackParamList>
  >;
```
Listing 18: CombinedStackParamList type

The snipped above utilizes a few `Typescript` features.

- `keyof` - returns the keys of an object as a union type.

- `CompositeScreenProps` - merges the props of two different screens.

- BottomTabScreenProps - the props of the BottomTabNavigator.

- NativeStackScreenProps - the props of the RootStackParamList.

Then, in the implementation of a screen, the function's arguments can be typed with this Type:

```
function Book({ route, navigation }: RootTabScreenProps<"Book">): ReactElement {
    // ...
}
```
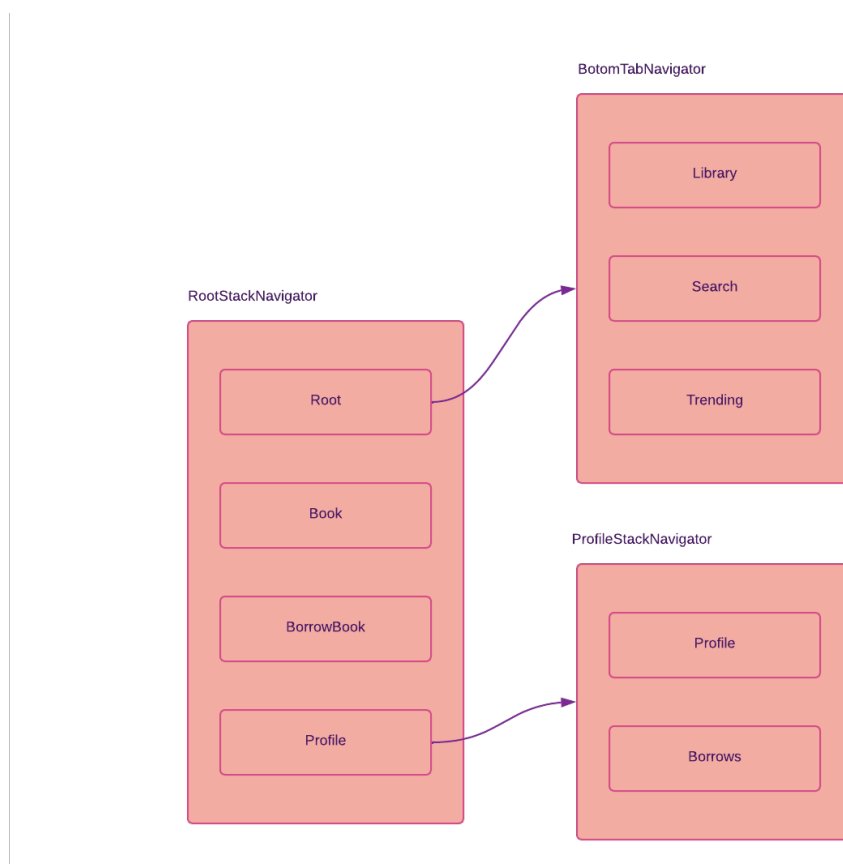
Listing 19: Typing arguments on a Screen



Figure 34: Pikkukirjasto navigation

The books of the library is cached in react-query, therefore to avoid unnecessary network requests, the Book object for the target screen, can be passed as a param in the navigation state.

While the screen is being transitioned to the target screen, the Book object is passed as a param in the navigation state, and the target screen can access this param from the route prop.
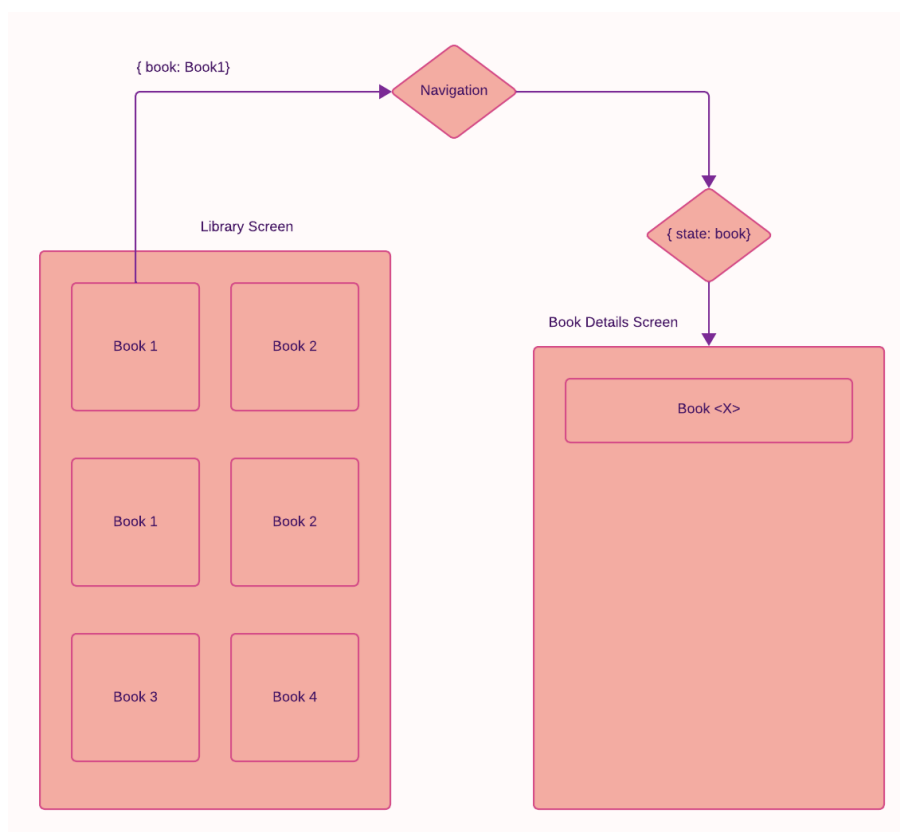
Figure 35: Navigation State

### 5.2.4  Data Fetching Abstraction

The `Pikkukirjasto` application uses `React-Query` to interact with the `Server`, and retrieve data from the database.

`React Query`[20] is a powerful and flexible data-fetching library for React applications. It is designed to simplify the management of remote data by providing a declarative and hook-based API, seamlessly handling caching, background data refetching, and pagination. `React Query` utilizes the "queryKey" as a unique identifier for each data query, allowing for efficient cache management and data invalidation.

The core feature of `React Query` is its caching mechanism. When a query is executed, the library automatically stores the fetched data in an internal cache. Subsequent requests with the same `queryKey` will utilize the cached data, reducing unnecessary network requests and improving application performance. Developers can also configure cache expiration policies to ensure data stays up to date.

Queries in `React Query` can have dependencies, and the library automatically takes care of updating dependent queries when their underlying data changes. This feature is especially useful when dealing with derived data or complex relationships between different data queries.

For mutations, `React Query` provides a "useMutation" hook that handles asynchronous data mutations and their optimistic updates. Developers can perform data mutations while the library optimistically updates the UI to provide a smoother user experience. In case of failure,

the optimistic update is rolled back.

`React Query` plays nicely with `TypeScript`, offering full type safety and intelligent autocompletion for queries and mutations. Developers can define specific types for query response data and mutation variables, ensuring type correctness throughout the application.

The library also supports pagination, making it easy to fetch and manage large sets of data. By using the built-in pagination features, developers can fetch data in batches, efficiently handling large datasets without compromising performance.

`React Query` embraces a flexible and extensible plugin system that enables developers to add custom logic and modify default behavior. This allows for tailored data-fetching solutions based on specific project requirements.

Overall, `React Query`'s combination of caching, query keys, mutations, `TypeScript` support, and pagination make it an invaluable tool for handling remote data in React applications. It simplifies data management, improves performance, and reduces the complexity associated with handling asynchronous data-fetching tasks.

In the application, `React Query` is mostly abstracted into separate custom hooks, to be able to reuse the same queries in different parts of the application. These custom hooks are, for example:

- `useBook`

- `useBookData`

- `useUser`

- `useDeviceId`

- `useBorrowedBooks`

The implementation of a reusable custom `React Hook` which implements a `React Query` instance:

```
function useBook() {
  const query = useQuery<Book[]>(
    queryKey: [bookQueryKeys.getAllBooks],
    queryFn: async () => {
      const { data } = await Api.get(apiEndpoints.getAllBooks);
      return data;
    },
    {
      refetchOnWindowFocus: false,
    }
  );
  return query
}
```

Listing 20: CombinedStackParamList type

In the example above, there are multiple abstractions.

The `queryKey` is a unique identifier for the query, and it's used to identify the query in the cache. The `queryKey` is an array of strings, and it identifies the data in the cache.

```
import type { QueryKey } from '@tanstack/react-query';
const bookQueryKeys = {
  all: ['books'] as QueryKey,
  bookById: (id: string): QueryKey => [
    ...bookQueryKeys.all,
    id,
  ],
  bookByAuthor: (author: string): QueryKey => [
    ...bookQueryKeys.all,
    author
  ],
  bookByIsbn: (isbn: string): QueryKey => [
    ...bookQueryKeys.all,
    isbn,
  ],
};
```

Listing 21: CombinedStackParamList type

The above example shows an implementation of a `QueryKeyFactory`. This factory is used to create `QueryKeys` for the `React Query` instance. The cache invalidation in `React Query` also works by the `QueryKey`, so having such implementation in place makes it easy to interact with the `React Query` cache.

These keys play a vital role in ensuring accurate internal data caching by the library. When a query's dependencies change, these keys enable automatic refetching, maintaining data integrity. Moreover, they provide the capability to manually engage with the `Query Cache` when specific situations arise. For instance, after performing a mutation, or when deliberate invalidation of certain queries becomes necessary. Before delving into how I optimize the organization of `Query Keys` for enhanced efficiency in achieving these objectives.

The flowchart below represents how `React Query` works under the hood. The `staleTime` and `cacheTime` play a crucial role in deciding the state of the query.

- `staleTime`: the duration until a query transitions from fresh to stale.

- `cacheTime`: the duration until inactive queries will be removed from the cache.
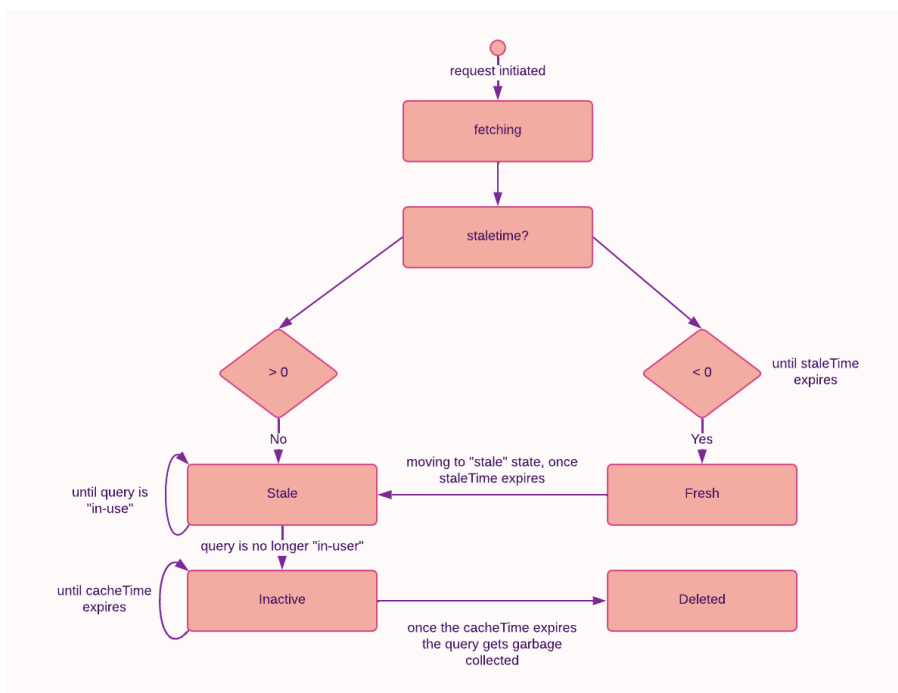
Figure 36: React Query

### 5.2.5 User & Auhtentication

To use the library, customer's do not need to create any type of subscriptions, nor provide any personal information. Therefore, the application is not collecting any personal data, user's do not need to create any account and go through any login, or authentication flow when using the `Pikkukirjasto` application.

However, it's important to have a way to identify users, and to be able to store user specific data, like the books they borrowed. Apart from that, it's also important to protect the backend, and the database from unauthorized access.

In an ordinary application, where there is an actual user, and the users are stored in a database, a basic authentication flow would work for example with `JWT` tokens, where the token is issued by the server after a successful login. The idea is similar here as well, but instead of having an account, the identifier of a user is the mobile device they are using.
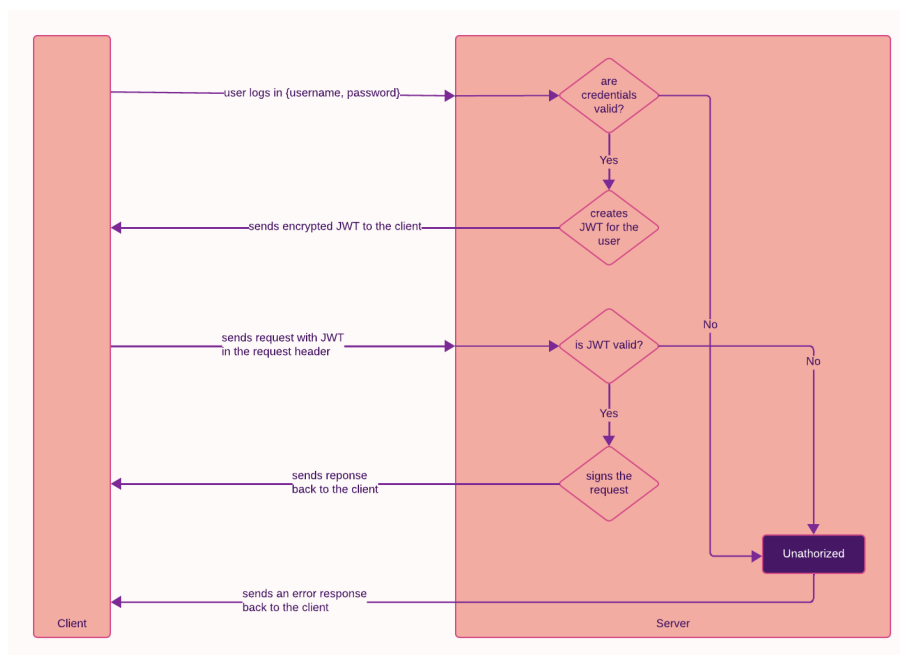
Figure 37: JWT Token Based Authentication

In the diagram above, a most generic authentication flow is shown. The user sends a request to the server, with the credentials, and the server responds with a JWT token. Then the JWT token will be sent in the header of every request the client initiated towards the server, and the server will attempt to validate the token, if it's valid, responds with the requested data.

In the Pikkukirjasto application the authentication flow is similar, but instead of having an account, the user's device will be identified by a unique identifier, which is generated by the application, and stored in the AsyncStorage.

There is an extra layer of authentication in addition to the Token exchange. The authentication with the API Gateway on the server happens through the biometric authentication of the user's device. This is a very secure way of authentication, because the user's device is the only device which can access the API Gateway.

Therefore, the update semantics of the authentication model is the following:
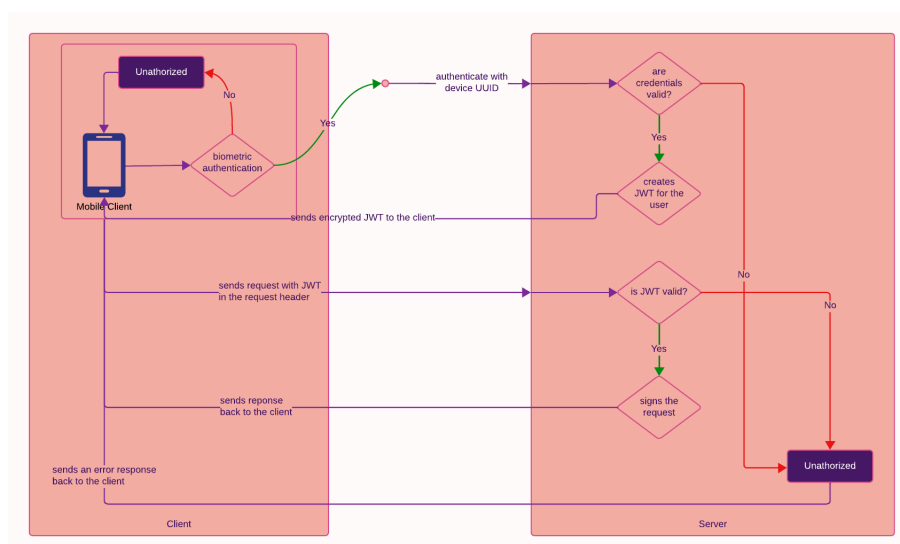
Figure 38: PikkuKirjasto Authentication Flow

Whenever there's a request towards the `API` the app will ask the user to authenticate with their biometric data, and if the authentication is succesful, the request will be sent to the server.

## 5.3 Backend

### 5.3.1 Localization

Considering the fact, that the initial release of the application is only focusing to a single group of users, and considering the outcome of the UX research, especially the personas, it became clear that the application's `User Interface` will require localization, so the users are given the option to choose their preferred language, and to switch between English and Finnish.

In `React`, (and therefore `React Native`) the language change happens during runtime, which also means that the localized elements (such as text, CTA Buttons, helper text, and so on) holds translation for every language that the application offers.

In this application I've utilized `i18n`, which is the industrial standard localization provider for `React` applications.

In order to use `i18n` in a `React` application, the `root` entry point of the app needs to be wrapped with `I18nextProvider`, which is part of the `i18n` library. To keep the entry point as clean and possible, I tend to handle the local props of different third-party wrappers in self-managed `Higher Order Functions`. The same applies for the `I18nextProvider` as well, which is implemented inside a custom wrapper component named `LocalizationProvider`

```
const LocalizationProvider = ({
    children,
}: ILocalizationProvider): ReactElement => (
    <I18nextProvider i18n={i18n}>{children}</I18nextProvider>
);
```

Listing 22: LocalizationProvider

In the code snippet above also shows, how the `config` for the `i18n` handled.

```
import i18n from "../i18n";

use(initReactI18next).init({
    resources,
    lng: "en",
    compatibilityJSON: "v3",
    debug: true,
});
```

Listing 23: `i18n configuration`

The `use` function is provided by the `i18n` library, and it takes a config `object` as a parameter, which sets the languages, and the actual translations, on an `object key` called `resources`.

```
export type Language = "en" | "fi";
const resources: Racord<Language, Object> = {
  en: {
    login: "Login",
  },
  fi: {
    login: "Kirjaudu"
  },
}
```

Listing 24: `i18n` translations

The language config object above demonstrates, how the `i18n` works. For every language the application offers, needs to have the same `key: string: string` value combination available for every text the application should localize.

Then, on the component level, it's fairly simple to use the localized texts:

```
const { t } = useTranslation();

const SomeComponent(): ReactElement => (
  <View>
      <Text>
        {t("hello.world")}
      </Text>
  </View>
)
```

Listing 25: Localization Example

The `t()` function takes a string as an argument, which represents the `path` the function will make to a specified text in the `object` being passed to the `resourced` config of the `i18n`.

---

[18]github.com/Shopify/restyle
[19]github.com/tanstack/react-query

## 6  Database

The database is a `MongoDB` database, and it's deployment via `AWS`, using the `AWS DocumentDB` service.

The cluster used by `DocumentDB` is also created in the `Pikkukirjasto CDK` stack definition, just like the other parts of the infrastructure.

The schemas of the database are defined in the implementation of the backend, and the database is seeded with the initial data in the `CDK` stack definition.

### 6.1  Database Schemas

There are three different schemas used in this application.

- `Book`

- `User`

- `Borrow`

The schemas are defined using `Mongoose`, which is a `MongoDB` object modeling tool designed to work in an asynchronous environment. It also utilizes the `Typescript` type system, so the schemas can be defined using `Typescript` types, together with `NestJS` decorators.

### 6.2  Book Schema

The following schema definition illustrates the structure and attributes of a `Book` entity within a `MongoDB` database, defined using the `Mongoose` library in a `TypeScript` environment. The `@Schema` decorator specifies that this schema is associated with a collection named `books` in the database.

The Book class represents the blueprint for the documents in the `Books` collection. Each document will have several properties defined using the `@Prop` decorator.

```
@Schema({ collection: 'books' })
export class Book {
  @Prop({ required: true })
  isbn: string;

  @Prop({ required: true, index: true })
  title: string;

  @Prop({ required: true })
  author: string;

  @Prop({ required: true })
  cover: string;

  @Prop({
    required: false,
    default: [],
    type: [MongooseSchema.Types.ObjectId],
    ref: 'Borrow',
```

```
    unique: true,
  })
  borrows: Types.ObjectId[];
}
```
Listing 26: Book Schema definition

- The `ISBN` property represents the `ISBN` code of the book and is marked as required.

- The `title` property holds the title of the book and is both required and indexed for efficient querying.

- The `author` property specifies the author's name of the book and is marked as required.

- The `cover` property contains a `URL` or path to the cover image of the book and is required.

- The `borrows` property represents an array of references to `Borrow` documents. It is not required and has a default empty array. It is of type `Types.ObjectId[]` (imported from `Mongoose`) and is linked to the `Borrow` schema using the ref field. This establishes a relationship between books and their associated borrow records. Additionally, the unique attribute ensures that each book's borrows array contains unique references.

Overall, this schema definition outlines the essential attributes of a `Book` document, along with its relationships to the `Borrow` documents, within the context of a `MongoDB` database using `Mongoose`.

## 6.3  User Schema

The following schema definition delineates the structure and characteristics of a `User` entity in a `MongoDB` database, constructed using the `Mongoose` library in a `TypeScript` setting. The `UserDocument` type is a hydrated version of the document of type `User`, enhancing its functionality.

The `User` class defines the blueprint for documents within the `users` collection. The `@Schema` decorator signifies that this schema corresponds to the `users` collection in the database. The timestamps: true option indicates that `MongoDB` should automatically generate `createdAt` and `updatedAt` timestamps for each document.

The `User` class features a single property:

The `userId` property signifies a unique identifier for each user and is marked as required and unique. This ensures that each user has a distinct identifier and prevents duplication. The `UserSchema` constant, generated using `SchemaFactory.createForClass(User)`, encapsulates the schema and provides a means to interact with it. This arrangement delineates the structure of `User` documents within the `MongoDB` database and employs `Mongoose`'s functionalities for improved data management.

```
@Schema({ collection: 'users', timestamps: true })
export class User {
    @Prop({ required: true, unique: true })
    userId: string;
}
```
Listing 27: Book Schema definition

6.4   Borrow Schema

The following schema definition delineates the structure and characteristics of a `Borrow` entity in a `MongoDB` database, constructed using the `Mongoose` library in a `TypeScript` setting. The `BorrowDocument` type is a hydrated version of the document of type `Borrow`, enhancing its functionality.

The `Borrow` class defines the blueprint for documents within the `borrows` collection. The `@Schema` decorator signifies that this schema corresponds to the `borrows` collection in the database. The timestamps: true option indicates that `MongoDB` should automatically generate `createdAt` and `updatedAt` timestamps for each document.

```
@Schema({ collection: 'borrows', timestamps: true })
export class Borrow {
  @Prop({
    required: true,
    unique: false,
    type: MongooseSchema.Types.ObjectId,
    ref: 'User',
  })
  user: MongooseSchema.Types.ObjectId;

  @Prop({ type: MongooseSchema.Types.ObjectId, ref: 'Book', required: true })
  book: Types.ObjectId;

  @Prop({ required: true, unique: false })
  isbn: string;

  @Prop({ required: true, unique: false, default: false })
  isBorrowed: boolean;
}
```

Listing 28: Book Schema definition

The `Borrow` schema connects the borrowed `Book` with it's own `ObjectId`, and the `User` who borrowed it. By making this, it's possible to query the database for all the books borrowed by a specific user, and all the users who borrowed a specific book. Having the `MongooseSchema.Types.ObjectId` as type of specific property, makes it possible to populate the data of that property, when querying the database.
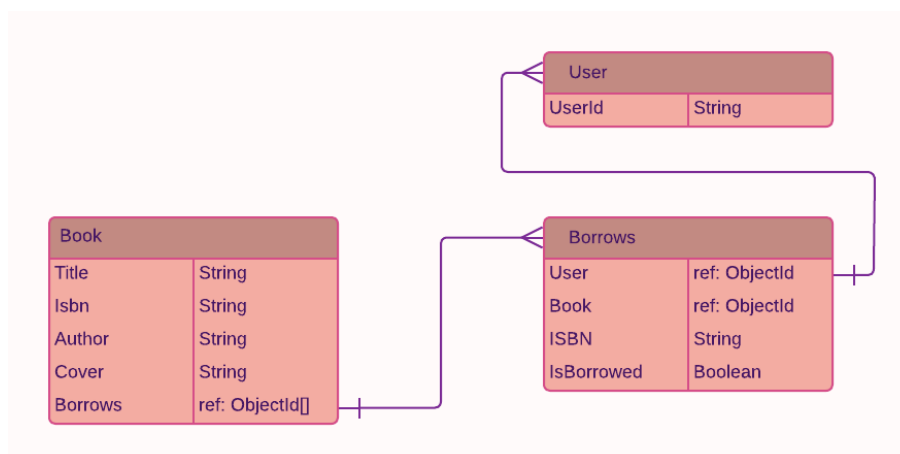


Figure 39: Pikkukirjasto DB Schema

7   Testing

Both of the application are heavily tested. The test results, and the coverage reports are available in a hosted standalone application.

7.1   Test-Driven Development

`Test-Driven Development`[21] (`TDD`) is a software development methodology that emphasizes writing tests before writing the actual code. It follows a cyclic process consisting of three main steps: writing a failing test, writing the minimum code required to make the test pass, and then refactoring the code while ensuring that the test still passes.

`TDD` starts with writing a test case that describes the desired behavior of the code. This test initially fails since there is no implementation yet. Then, developers proceed to write the simplest code that fulfills the requirements of the test. This approach ensures that the code is written to satisfy specific functional expectations.

`TDD` promotes a strong feedback loop as tests are run frequently during development. This not only helps to catch bugs early but also aids in designing modular and loosely coupled code, leading to better maintainability and extensibility. Additionally, `TDD` encourages developers to focus on writing code that is tightly aligned with the requirements, which can lead to more reliable and predictable software.

By writing tests before the code, `TDD` also acts as a form of documentation, providing clear examples of how the code is intended to be used. This can be especially helpful for other developers who might work on the codebase in the future.

While `TDD` can lead to improved code quality and faster bug detection, it can also demand a mindset shift for some developers. Writing tests before writing code requires careful planning and consideration of the software's architecture. However, this initial investment often pays off in terms of reduced debugging time and increased confidence in the correctness of the codebase.

Since both the `Backend` and the `Mobile Application` are written in `TypeScript`, I have used a `JavaScript` testing framework called `Jest`.

While both application in written in `TypeScript`, due to their different nature, their testing approach is also rather different.

7.2   Backend testing

The backend is a `Node.js` application, therefore it is tested with `Jest`. There are two different types of tests in the backend. Unit tests, and integration tests. The unit tests are testing the individual functions, and the integration tests are testing the endpoints of the application, along with the database abstraction.

In the context of a `REST API` developed using `NestJS`, effective unit testing involves the comprehensive validation of individual components and functions within the application. Utilizing `NestJS`'s built-in testing utilities and libraries like `Jest`, unit tests should cover various aspects of the `API`, such as endpoints, services, controllers, and data manipulation.

For instance, when testing a controller method, the unit test would ideally mock dependencies and simulate `HTTP` requests, ensuring that the controller's response and behavior align with expectations. Similarly, unit tests for services would assess business logic, database interactions, and error handling, all while isolating the service from external factors. By meticulously crafting unit tests in this manner, developers can swiftly identify regressions and bugs within the `API`, leading to a robust and reliable `RESTful` service that adheres to the principles of maintainability and testability.

The test files are placed close to the actual implementation, and it follows the `.spec.ts` naming convention.

In unit testing, the goal is to isolate each part of the program and show that the individual parts are correct. By isolating the parts of the program, unexpected interactions are avoided, so unit tests make it easier to test the program's correctness. When writing unit tests in `Jest`, the `describe` and `it` methods are used to define the test suite and test cases, respectively. The `expect` method is then used to define the expected output of the test case. The `expect` method takes a value called the `actual` and compares it against the `expected` value. If the `actual` value matches the `expected` value, then the test case is passed. Otherwise, it is failed.

Mocking in `Jest` is a powerful technique used to simulate the behavior of external dependencies, functions, or modules during unit testing. By creating mock implementations, developers can control the responses and interactions of these dependencies, allowing for isolated testing of specific code components. This is particularly valuable when testing interactions with databases, `APIs`, or other complex systems that should be decoupled from the unit under test. `Jest` provides built-in mocking utilities that enable the creation of mock functions and objects, making it easier to simulate real-world scenarios without actually invoking external resources. Overall, mocking in `Jest` enhances the precision and efficiency of unit tests by providing a controlled environment to assess the behavior of individual code units in isolation.

```
getAllBooks: jest.fn().mockResolvedValue([...demoBook]),
```
Listing 29: Example mocking in Jest

The example above mocks a simple function in the `book.controller.spec.ts`, which is the `getAllBooks` function. The `jest.fn()` creates a mock function, and the `mockResolvedValue` sets the return value of the mock function. This way, the function can be tested without the need of a database connection.

When a specific implementation needs chained mocking, that can be achieved also, utilizing the `mockImplementation` method.

```
jest.spyOn(model, 'findOne').mockReturnValue(
    createMock<Query<BookDocument, BookDocument>>({
        populate: jest.fn().mockImplementationOnce(() => ({
        exec: jest.fn().mockResolvedValueOnce(thisBook),
        })),
    }),
);
```
Listing 30: Chained mocking example in Jest

The tests can be executed using the `CLI` provided by `Jest`.

```
pnpm run text
```



Figure 40: Backend Jest output

The test cases are executed in a `CI` environment as well, and the test results are available in the `CI` pipeline.



Figure 41: Backend Jest output Github CLI

### 7.2.1  Test coverage

Test coverage is a metric that measures the percentage of code that is covered by automated tests. It is an indicator of the quality of the tests, and it can be used to assess the risk of undetected bugs in the codebase. A higher test coverage indicates that there is a lower risk of bugs in the codebase, while a lower test coverage indicates that there is a higher risk of bugs in the codebase. The test coverage of the backend is measured by `Jest`, and the results are generated in the `CI` pipeline, during either staging or production release.
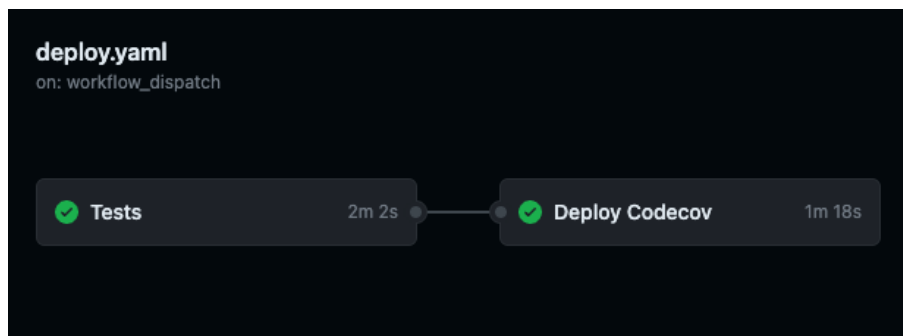


Figure 42: Generating Test Coverage Report

```
-------------------------- --------- --------- --------- --------- --------------------
File                       | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------- --------- --------- --------- --------- --------------------
All files                  | 28.93   | 0        | 40.54   | 28.48   |
 src                       | 25.45   | 0        | 30      | 21.27   |
  app.controller.ts        | 100     | 100      | 100     | 100     |
  app.module.ts            | 0       | 100      | 0       | 0       | 1-35
  app.service.ts           | 100     | 100      | 100     | 100     |
  main.express.ts          | 0       | 0        | 0       | 0       | 1-36
  main.ts                  | 0       | 0        | 0       | 0       | 1-29
 src/controllers/book      | 100     | 100      | 100     | 100     |
  book.controller.ts       | 100     | 100      | 100     | 100     |
 src/controllers/borrow    | 0       | 0        | 0       | 0       |
  borrow.controller.ts     | 0       | 0        | 0       | 0       | 1-53
 src/controllers/user      | 0       | 100      | 0       | 0       |
  user.controller.ts       | 0       | 100      | 0       | 0       | 1-23
 src/modules/book          | 0       | 100      | 100     | 0       |
  book.module.ts           | 0       | 100      | 100     | 0       | 1-23
 src/modules/borrow        | 0       | 100      | 100     | 0       |
  borrow.module.ts         | 0       | 100      | 100     | 0       | 1-25
 src/modules/user          | 0       | 100      | 100     | 0       |
  user.module.ts           | 0       | 100      | 100     | 0       | 1-24
 src/schemas/book          | 100     | 100      | 100     | 100     |
  book.schema.ts           | 100     | 100      | 100     | 100     |
 src/schemas/borrow        | 0       | 100      | 100     | 0       |
  borrow.schema.ts         | 0       | 100      | 100     | 0       | 1-26
 src/schemas/user          | 0       | 100      | 100     | 0       |
  user.schema.ts           | 0       | 100      | 100     | 0       | 1-12
 src/services/book         | 94.11   | 100      | 85.71   | 93.33   |
  book.service.ts          | 94.11   | 100      | 85.71   | 93.33   | 55
 src/services/borrow       | 0       | 100      | 0       | 0       |
  borrow.service.ts        | 0       | 100      | 0       | 0       | 1-35
 src/services/user         | 0       | 100      | 0       | 0       |
  user.service.ts          | 0       | 100      | 0       | 0       | 1-27
-------------------------- --------- --------- --------- --------- --------------------
```

Figure 43: Backend Codecov report

The report generated by `Jest` is public avaliable at `backendcov.thesis.richardzilahi.hu`.



Figure 44: Public Codecov Report for Backend

## 7.3   App testing

The testing of the `React Native` mobile application also uses `Jest` as its test runner, however the testing approaches and methods are differs from the server side.

When writing `unit tests` for a `React` application, usually there are three main aspects that every test case should consider:

- ensure the component renders without any errors

- ensure their props are handled correctly

When it comes to writing good unit tests, there are a several best practices to follow:

- Always test components in isolation: React components should be tested in isolation from the rest of the application, to avoid unexpected side-effects from other parts of the application affecting the test results.

- Test the render output: You should test the output of a component to make sure that it is rendering correctly.

- Test edge cases: Make sure to test the component with a variety of different props and state values, including edge cases like empty or invalid data, to make sure that it works correctly in all scenarios.

import React from "react"; import render, fireEvent from "@testing-library/react"; import DummyComponent from "./DummyComponent";

```
describe("DummyComponent", () => {
  it("renders correctly", () => {
    const { container } = render(<DummyComponent />);
    expect(container).toMatchSnapshot();
  });

  it("updates its state when a button is clicked", () => {
    const { getByText } = render(<DummyComponent />);
    const button = getByText("Click me");
    fireEvent.click(button);
    expect(button).toHaveTextContent("Clicked 1 times");
  });
});
```

Listing 31: Jest React Test Example

In this example, the render method from `@testing-library/react` is used to render the `DummyComponent` component into a test environment. The `getByText` method is then used to find a specific element within the component, and the `fireEvent.click` method is used to simulate a user interaction with the button. Finally, the expect function is used to assert that the component's text content has been updated correctly.

When it comes to unit-testing custom react-hooks there are a few aspects to keep in mind to ensure that the tests are effecntive enough:

- Testing the hook behavior: It's important to test the behavior of the hook to ensure that it works as expected. This may include testing the initial state, updating the state, and any side effects that the hook might produce.

- Testing different use cases: The hook may be used in different ways throughout the frontend application, so it's important to test it in a variety of use cases to ensure that it works correctly in all situations.

- Testing edge cases: It's important to test edge cases and error conditions to ensure that the hook behaves as expected in these scenarios. This may include testing with invalid input, empty values, or null values.

- Mocking dependencies: If the hook relies on external dependencies, such as APIs or other hooks, it may be necessary to mock these dependencies in the tests to ensure that the hook behaves consistently and predictably.

- Ensuring proper cleanup: If the hook performs any cleanup operations, such as unsubscribing from event listeners, it's important to ensure that these operations are performed correctly and that any resources are properly released.

### 7.3.1  Test coverage

The test coverage of the mobile application is measured by `Jest`, and the results are generated in the `CI` pipeline, during either staging or production release.

Similar to the `Backend` testing, the test coverage is almost 109% for the `React Native` application as well.



```
================== Coverage summary ==================
Statements   : 99.15% ( 469/473 )
Branches     : 92.3% ( 72/78 )
Functions    : 97.34% ( 110/113 )
Lines        : 99.14% ( 463/467 )

Snapshot Summary
 › 1 snapshot updated from 1 test suite.
Test Suites: 35 passed, 35 total
Tests:       74 passed, 74 total
Snapshots:   1 updated, 4 passed, 5 total
Time:        15.791 s
Ran all test suites.


 > NX    Successfully ran target test for project mobile-client (17s)
```

Figure 45: Public Codecov Report for Mobile Application

Complete, 100% coverage unfortunately was not possible to reach here, because the Application implements a feature on the native camera, to scan the bard codes of the books, and the native camera can not be mocked any meaningful way which would help `Jest` to discover the coverage of that specific code.

The code coverage report of the `React Native` application is available publicly behind the following URL: `appcov.thesis.richardzilahi.hu`

---

[20]en.wikipedia.org/wiki/Test-driven_development

## 8   Documentation

Documentation is a fundamental aspect of software development, playing a critical role in ensuring the clarity, maintainability, and longevity of a codebase. In this chapter I will be discussing the importance of documentation, its best practices, and its integral role in software projects.

Documentation is the written record of every software project's design, functionality, and usage. It serves as a critical source of information for developers, maintainers, and stakeholders. Comprehensive documentation is essential for various reasons, including onboarding new team members, troubleshooting issues, enhancing code maintainability, and ensuring project durability. It offers a structured way to communicate vital details about the software, reducing dependencies on individual knowledge and promoting collaboration.

### 8.1   Types of Documentation

Effective documentation encompasses various types, including code documentation, architectural documentation, user documentation, and `API` documentation. Code documentation, often in the form of comments, provides insights into the codebase's structure and function, helping developers understand how different parts of the code work. Architectural documentation outlines the system's overall design, the relationships between its components, and its dependencies. User documentation is vital for guiding end-users through an application, ensuring they can utilize it efficiently. `API` documentation defines how external components or services interact with the software, fostering integration and interoperability.

### 8.2   Documentation Best Practices

To create valuable documentation, it should be clear, concise, up to date, and easily accessible. It's important to use descriptive names for classes, functions, and variables. Employing meaningful comments to explain the purpose of code segments and any associated constraints or edge cases. Documentation should be consistent and follow a structured format to enhance readability. It's essential to keep documentation up to date as code evolves, ensuring that it remains relevant. Include examples, usage scenarios, and practical information to facilitate understanding. Keeping the documentation in version control systems (like `Git`) can also be used to maintain documentation alongside the codebase, ensuring alignment with code changes. Therefore, documentations should be considered an integral part of the codebase. Embedding it within the code repository makes it easily accessible to developers, fostering a seamless transition when different team members collaborate or when new contributors join the project. Developers can maintain documentation files alongside code files, ensuring they stay in sync with code changes. This practice reinforces the concept that documentation is not a separate task but an ongoing and parallel process within software development.

### 8.3   Utilized Documentation Tools

In this project, I've utilized multiple tools to create and maintain documentation across the codebase, keeping in mind the nature of the `React Native` application, and the server side `Node.js` application.

### 8.3.1 Server Side

As mentioned in the previous sections, the server side application uses `NestJS` framework, which implements a `RestAPI` interface. Documenting `APIs` are the most crucial part of every server side application, as it contains the methods and endpoints that are used to communicate with the application.

The `NestJS` framework provides a built-in tool to generate `API` documentation, based on the `API` endpoints, and the `DTOs` used in the application. This specific tool is called [22]`Swagger`, and it is a tool that can be used to generate `API` documentation, and it also provides a user interface to interact with the `API` endpoints.
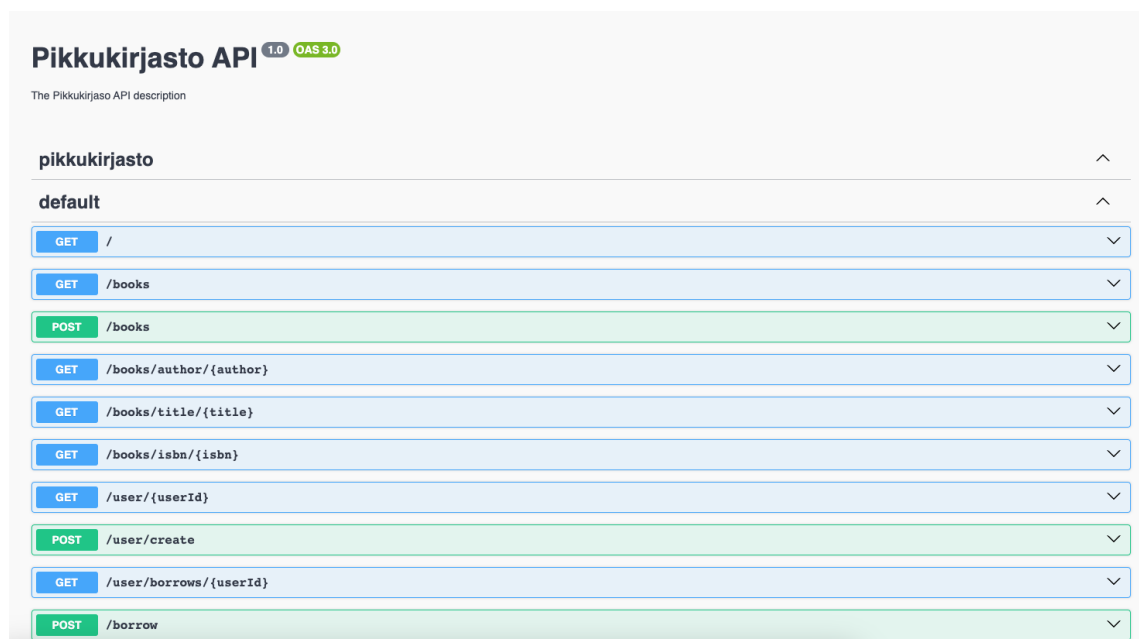


Figure 46: Swagger API Documentation

Similar to the `Codecoverage` reports for both the backend and app applications, the `Swagger` documentation is also deployed to a dedicated `StaticStack` on `AWS`, and it's publicly accessible, at `apidoc.richardzilahi.thesis.hu`

### 8.3.2 Mobile Application

To document the mobile application, I've utilized the `JSDoc` tool, which is a markup language used to annotate `JavaScript` source code files. While `JavaScript` is primarily used to document `JavaScript` application, it can also be used to document `TypeScript` files, using `TypeScript` specific annotations.

When using properly configured `JSDoc` setup for a `TypeScript` project, the `JSDoc` tool can generate documentation for the entire codebase.

The example blow is a generic `JSDoc` comment, in a `JavaScript` file:

```
/**
 * ActionRequest
 * @memberof Action
 * @alias ActionRequest
 */
export type ActionRequest = {
  /**
   * parameters passed in an URL
   */
  params: {
    /**
     * Id of current resource
     */
    resourceId: string;
    /**
     * Id of current record
     */
    recordId?: string;
    /**
     * Name of an action
     */
    action: string;

    [key: string]: any;
  };
}
```

Listing 32: Javascript Documentation Command


Using a specific configuration for `JSDoc` will turn the example above into the following documentation:

```
/**
 * ActionRequest'
 * @memberof Action'
 * @alias ActionRequest'
 * @typedef {object} ActionRequest'
 * @property {object} params    parameters passed in an URL'
 * @property {string} params.resourceId   Id of current resource'
 * @property {string} [params.recordId]   Id of current record'
 * @property {string} params.action   Name of an action'
 * @property {any} params.{...}'
 */
```

Listing 33: TypeScript Documentation Comment


The `JSDoc` documentation is also deployed to a dedicated `StaticStack` on `AWS`, and it's publicly available at `jsdocs.thesis.richardzilahi.hu`.

Figure 47: JSDoc on Mobile Application

Similarly to all the other documentation, the `JSDoc` documentation is also deployed automatically, using the `Deployment GitHub Action`.
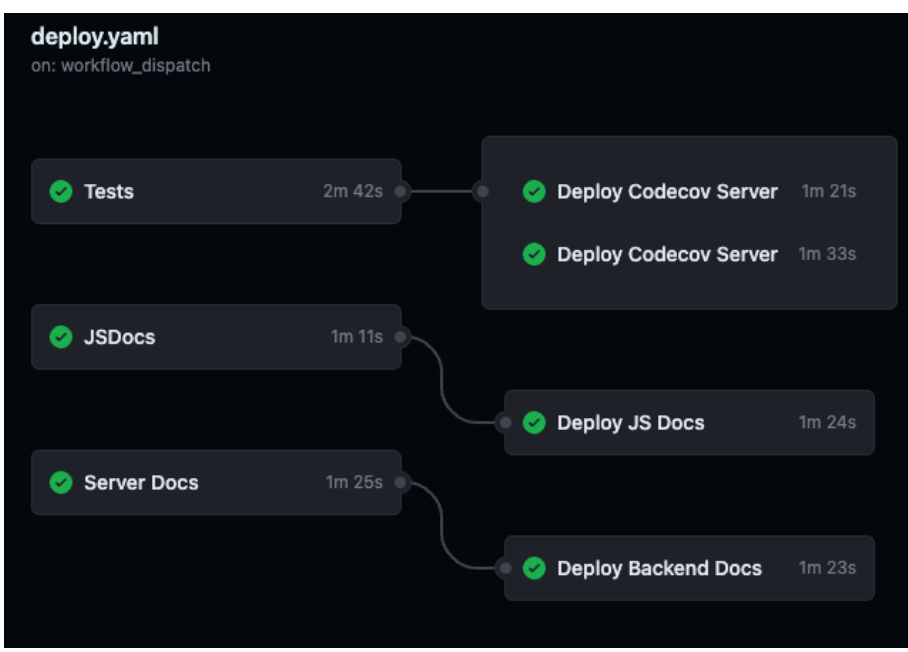


Figure 48: JSDoc Deployment

---

[21]swagger.io

9   Next Steps

The development in this application won't stop by the context of this thesis. It will be actively continued and expanded in the future. There are various different angles and already existing ideas where this application could evolve in the future.

In this chapter of my thesis I am going to introduce these ideas, and potentions where this application could go, and what it still could achieve.

### 9.0.1   Generalizing the idea

Since the very beginning of the development process, what's more, already during the architecture planning, I have kept in mind, that potentially this application won't be stopping on its initial target, which is to supply a digitalized set of services for the `Halkeinkiven Pikkukirjasto` library only. In Finland, luckily there are many other small, community driven libraries, and I am sure most of them lacking such services and solution that I have attempted to solve in this thesis work. Therefore, one future idea would be to open source the architecture, and by making the database more robust and scalable, in the future it will be fairly simply and straightforward to pull in other libraries, and this library's books to the application, and to provide the same features, solutions for other libraries across the country as well.

## 10 Conclusions

In the realm of user experience, the journey began with a deep understanding of the unique needs and expectations of the target users. Through comprehensive user research and persona development, I crafted a holistic picture of the users, which included Aino, Seppo, Sanna, and the contemporary reader, Lauri. This in-depth understanding laid the foundation for the creation of a digital library app that truly resonated with their goals, frustrations, and preferences.

The "empathize" phase was instrumental in ensuring that the app design was not only aesthetically pleasing but also highly functional. I listened to the desires and pain points of the personas, and this knowledge guided the design decisions. The implementation of a light color scheme, where orange plays a central role, was a testament to the care we took in addressing the psychology of color and the significance of visual elements. This, coupled with an intuitive user interface, makes the app easily accessible for users of all ages and backgrounds.

The continuous commitment to iterative refinement and usability testing assured that the app aligns seamlessly with user goals and expectations. This user-centered approach guarantees that users have an engaging and frustration-free experience, thus fulfilling our primary objective in the UX domain.

On the engineering side, the app was constructed using cutting-edge technologies and best practices. The integration of AWS and `CDK` allowed for scalable and cost-efficient cloud infrastructure, while the use of React Native ensured cross-platform compatibility, enabling access for a wide range of users, including those on iOS and Android devices.

The backend, built using `NestJS` and `TypeScript`, provided a robust and well-structured foundation for the server side. The use of MongoDB as the database of choice facilitated efficient data management and storage.

Additionally, I embraced the principles of semantic release, ensuring that the app's versioning and release process flow adhered to industry standards, offering reliability and transparency to users.

In conclusion, the engineering work were guided by a commitment to providing a seamless and performant cross-platform mobile application. The harmonious fusion of the UX and engineering aspects resulted in an app that caters to users' goals and expectations while delivering a technically sound and dependable platform. This convergence of user experience and engineering excellence represents the essence of our thesis and underscores our dedication to creating a user-friendly and technologically robust solution for the diverse set of users.

References

Brown, Tim (2023). *Design Thinking*. Accessed 2023-6-15
https://www.interaction-design.org/literature/article/what-is-design-thinking-and-why-is-it-so-popular.

Gibbons, Sarah (2023). *Design Thinking 101*. Accessed 2023-7-1
https://www.nngroup.com/articles/design-thinking.

Kalik, Anton (2023). *Implementing Semantic Release for Public Non-Scoped Packages*.
Accessed 2023-9-12
https://betterprogramming.pub/implementing-semantic-release-for-public-non-scoped-packages-4593ff81f39f.

Khorrami, Najma (2023). *How to Understand Customer Needs*. Accessed 2023-7-1
https://www.entrepreneur.com/growing-a-business/how-to-understand-customer-needs/429343.

Onuta, Anca (2023). *How to Get Started with Test Driven Development — A Step by Step Guide*. Accessed 2023-6-16
https://medium.com/geekculture/how-to-get-started-with-test-driven-development-a-step-by-step-guide-bfdd831fae71.

Stevens, Emily (2023). *How To Conduct UX Research For Maximum Value*. Accessed 2023-6-29
https://usabilitygeek.com/how-to-conduct-ux-research.

Veal, Raven (2023). *How to Define a User Persona*. Accessed 2023-7-2
https://www.nngroup.com/articles/design-thinking.

Figures

Tables