Graphics/FMI-03.png

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science

BACHELOR

SUPERVISOR:
Conf. Micotă Flavia Elena
Drd. Beleș Damian-Teodor

GRADUATE:
Stăniloiu Grigore Liviu

TIMIŞOARA
2023

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science

Moneager: A Mobile Application for Managing Financial Transactions

SUPERVISOR:
Conf. Micotă Flavia Elena
Drd. Beleș Damian-Teodor

GRADUATE:
Stăniloiu Grigore Liviu

Abstract

Moneager is a mobile application that helps users manage their finances in a comprehensive and personalized way. Moneager makes it simple to add transactions, create reports, and keep track of expenses with its user-friendly interface. The application has simple navigation and visuals for a better experience.

Besides its core features, Moneager also incorporates educational resources to promote financial literacy and encourage users to develop healthier financial habits. Moneager provides users with tools to make informed financial choices and reach their financial goals.

When compared to other money management applications, Moneager stands out as a user-friendly and accessible option that caters to a wider audience. Moneager's focus on ease of use and affordability makes it a great choice for anyone wanting to manage their finances. Overall, Moneager offers a comprehensive and convenient solution for managing personal finances, with an emphasis on user experience and financial education.

Abstract

Moneager este o aplicație mobilă care ajută utilizatorii să-și gestioneze finanțele într-un mod cuprinzător și personalizat. Moneager face simplă adăugarea tranzacțiilor, crearea de rapoarte și monitorizarea cheltuielilor cu ajutorul interfeței sale prietenoase. Aplicația are o navigare și un aspect vizual simplu pentru o experiență mai bună.

În plus față de funcțiile sale de bază, Moneager integrează și resurse educaționale pentru a promova educația financiară și a încuraja utilizatorii să dezvolte obiceiuri financiare sănătoase. Moneager pune la dispoziție utilizatorilor instrumente pentru a lua decizii financiare informate și a-și atinge obiectivele financiare.

Comparativ cu alte aplicații de gestionare a banilor, Moneager se evidențiază ca o opțiune prietenoasă și accesibilă, adresându-se unui public mai larg. Accentul pus de Moneager pe ușurința în utilizare și accesibilitatea sa o face o alegere excelentă pentru oricine dorește să-și gestioneze finanțele. În ansamblu, Moneager oferă o soluție cuprinzătoare și convenabilă pentru gestionarea finanțelor personale, cu accent pe experiența utilizatorului și educația financiară.

Contents

List of Figures

Chapter 1

Introduction

Managing finances can be a challenge for many people, especially for those who have never received proper education on financial management. Poor financial management can lead to debt, and financial stress, which can negatively impact one's quality of life.

One of the biggest issues with money management is the lack of financial education. Many people have never been taught how to properly manage their finances, leading to poor spending habits, lack of savings, and accumulating debt. Without the proper knowledge and skills, it can be difficult to make informed financial decisions and plan for the future.

Another problem is that many people live beyond their means, which means they spend more money than they earn. This can lead to accumulating debt and struggling to make ends meet. Living beyond one's means can also prevent individuals from saving money for future expenses or emergencies.

Savings are important because they provide a safety net for unexpected expenses and can help individuals achieve their long-term financial goals. Having savings can also reduce financial stress and provide a sense of security. However, managing finances can be a challenge for various people, especially those who lack proper financial education. Poor financial management can lead to debt and financial stress, negatively impacting one's quality of life.

Managing finances can be a challenge for many people, especially for those who have never received proper education on financial management. Poor financial management can lead to debt, and financial stress, which can negatively impact one's quality of life.

The lack of financial education is a major issue, as many individuals have never been taught how to manage their finances effectively. Accumulating debt is a consequence of poor spending habits and a lack of savings. Spending beyond one's means can cause financial hardship, making it difficult to save and pay bills.

Savings are crucial for providing a safety net for unexpected expenses and achieving long-term financial goals. They reduce financial stress and provide a sense of security. A lack of financial education, overspending, and neglecting to prioritize savings all contribute to people's difficulty in saving money.

While there are money management applications available, they often lack comprehensiveness and personalization. Complex applications lack user-friendliness and the ability to meet varied needs. They may be difficult to use or overwhelm users with unnecessary features. The proposed application focuses on user-friendly design, prioritizes key features, and provides relevant financial education resources User-friendliness and a visually appealing interface are crucial for the success of an application. There is a requirement for a mobile app that offers an effective and user-friendly solution for personal finance management. The application should have a personalized approach, offer relevant financial education, and promote long-term financial stability and well-being.

A mobile platform has been proposed to provide an easy-to-use and intuitive financial management experience. The interface prioritizes usability and simplicity, with clear navigation and visual elements. The application incorporates features to promote financial education, offering tips and resources for saving money, investing wisely, and making informed financial decisions.

By offering a comprehensive approach to financial management, the mobile app stands out. It's possible to add transactions, generate reports, monitor expenses, and set savings goals through this service. The application aims to help users develop better financial habits and achieve their goals.

The lack of financial education is a major issue, as many individuals have never been taught how to manage their finances effectively. Accumulating debt is a consequence of poor spending habits and a lack of savings. Spending beyond one's means can cause financial hardship, making it difficult to save and pay bills.

Savings are crucial for providing a safety net for unexpected expenses and achieving long-term financial goals. They reduce financial stress and provide a sense of security. A lack of financial education, overspending, and neglecting to prioritize savings all contribute to people's difficulty in saving money.

While there are money management applications available, they often lack comprehensiveness and personalization. Complex applications lack user-friendliness and the ability to meet varied needs. User-friendliness and a visually appealing interface are crucial for the success of an application. There is a requirement for a mobile app that offers an effective and user-friendly solution for personal finance management. The application should have a personalized approach, offer relevant financial education, and promote long-term financial stability and well-being.

A mobile platform has been proposed to provide an easy-to-use and intuitive financial management experience. The interface prioritizes usability and simplicity, with clear navigation and visual elements. The application incorporates features to promote financial education, offering tips and resources for saving money, investing wisely, and making informed financial decisions.

By offering a comprehensive approach to financial management, the mobile app stands out. It's possible to add transactions, generate reports, monitor expenses, and set savings goals through this service. The application aims to help users develop better financial habits and achieve their goals.

The proposed application focuses on user-friendly design, prioritizes key features, and provides relevant financial education resources.

Two similar solutions, BlueCoins, and Money Lover are compared. BlueCoins offers customization and detailed tracking but may be overwhelming for casual users. Money Lover emphasizes user experience but hypes paid features, which may not be accessible to all users. Revolut, although not a direct competitor, is mentioned for its popularity and user-friendly interface.

1.1 Proposed Solution

The application is user-friendly and provides an intuitive experience for managing finances. Clear navigation and visual elements are used in the user interface to prioritize usability and simplicity, enhancing user experience. The application aims to provide users with the tools and resources to manage their finances efficiently and effectively.

Besides its user-friendly interface, the application incorporates features to promote financial education and literacy. It provides users with tips and resources for saving money, investing wisely, and making informed financial decisions. Financial literacy is essential for individuals to make smart financial decisions.

The proposed mobile application is unique in its comprehensive approach to managing personal finances, offering users a range of features that are tailored to their specific needs. From adding transactions and generating reports to monitor expenses and setting savings goals, the application offers a full suite of tools and resources to help users achieve their financial goals.

Overall, the proposed mobile application offers a powerful combination of user-friendly design and financial education resources, making it an ideal tool for individuals seeking to improve their financial management skills and achieve their long-term financial goals.

1.2 State of the art

Although there are a plethora of money management applications available in the market, the proposed application stands out from the rest because of its exceptional level of customization and personalization, which many other applications cannot provide. In addition, some applications may be difficult to use or have a steep learning curve, which can deter users from using them effectively.

In order to address this gap, the proposed application has been developed to feature a user-friendly interface that considers the principles of both UI and UX. The application has been designed with the user in mind, and it is geared towards making sure that users can perform tasks with no difficulties. These tasks include navigation, transaction addition, report generation, and expense monitoring.

Moreover, the proposed application goes beyond simply tracking expenses by also providing educational resources and tips for saving money. This is important as many people struggle with managing their finances due to a lack of financial education. By incorporating educational features, the proposed application aims to help users develop better financial habits and achieve their financial goals.

While many comparable solutions exist, a significant number of them suffer from "feature creep", which arises when an excessive number of features are added without proper consideration given to how they will impact usability and user experience. The whole purpose of a money management tool is to make it easier for users to manage their finances, but if the application is confusing and overwhelming, it defeats this purpose altogether. The issue at hand is being tackled with our proposed mobile application that gives top priority to a user-friendly interface and concentrates on the key features that are most vital for effective financial management.

In contrast to other money management solutions, the proposed application takes a distinctive and comprehensive approach that prioritizes user experience and education.

1.2.1 BlueCoins

BlueCoins, a comprehensive money management application, offers a wide range of features that meet the power users' needs. Although this application may be appropriate for advanced users who need a wide range of customization and intricate financial monitoring, it might not be the ideal option for every user, specifically those who prioritize the user-friendliness and simplicity of their financial management apps.

One of the main advantages of BlueCoins is its ability to allow users to create custom categories for expenses and set limits for each category to stay within their budget. It also offers features such as transaction categorization, budget forecasting, and reports. Users can also create recurring transactions for monthly expenses such as rent or bills

Despite its various features, the app's lack of focus on user experience may lead to decreased interest from casual users who are seeking a simple and efficient method for managing their finances. The application may experience a phenomenon known as "feature creep," which refers to the situation where various features are continually added to the program, resulting in increased complexity and making it overwhelming for some users.

Although BlueCoins is equipped with detailed financial tracking and an extensive range of customization options, users who value simplicity and a user-friendly interface may not find it to be the best fit for their financial management needs.

Figure 1.1: Blue Coins Dark main menu 1.

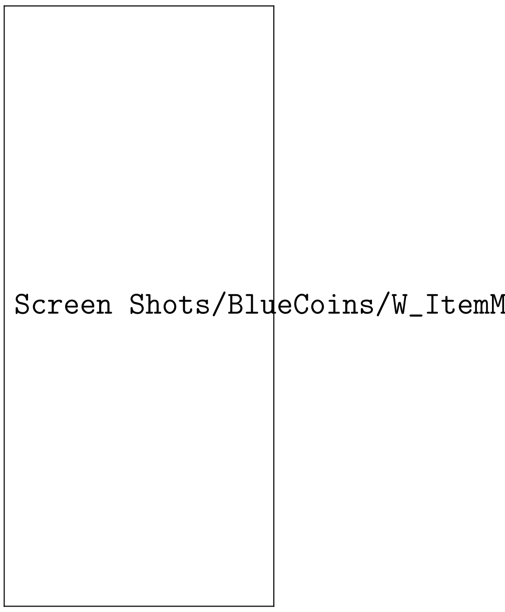

Figure 1.2: Blue coins main menu .

1.2.2 Money Lover

Money Lover is a mobile application that allows users to manage their personal finances by tracking their expenses, creating budgets, and setting financial goals. The app offers features such as expense tracking, bill reminders, and budget planning. It also provides reports and insights to help users better understand their spending habits and financial situation.

Compared to BlueCoins, Money Lover places a greater emphasis on user experience, with a user-friendly interface that is easy to navigate and customize. It also offers a variety of customization options, such as the ability to choose between different themes and fonts. Additionally, Money Lover has a social feature that allows users to share expenses with friends and family, making it a useful tool for group expenses.

Although Money Lover is a robust application for personal finance management, it does have a heavier focus on its paid feature set. This could potentially make it less appealing to the average user who may not yet be ready to invest in additional features.

Moreover, the app may be seen as contradictory to its purpose, as the expensive paid feature set may discourage users who are looking for a cost-effective and accessible solution. In comparison to other solutions, Money Lover may not be as user-friendly and intuitive, especially for those who prioritize simplicity and ease of use in their financial management applications.

While it is a comprehensive tool for managing personal finances, it leans more towards a paid feature set which may be less accessible for the average user. This may be a disadvantage for those who are just starting to manage their finances and are not yet willing or able to pay for additional features.

`Screen Shots/MoneyLover/D_MainMenu.jpg`

Figure 1.3: Money Lover main menu 1.

`Screen Shots/MoneyLover/Account.`

Figure 1.4: Money Lover account screen.

1.2.3 Revolut

Revolut is a financial technology company that provides a mobile application for managing finances, including money transfers, currency exchange, and investment services. The app allows users to hold and exchange money in different currencies at the interbank exchange rate, making it a popular choice for international transactions.

While Revolut may not be a direct competitor to the proposed money management application, it is worth mentioning as it is a popular choice among users for managing their finances. The app's user-friendly interface and innovative features have made it a popular choice among millennials and tech-savvy users.

One advantage of Revolut is its focus on security and fraud prevention. The app offers features such as disposable virtual cards and transaction notifications to help users detect and prevent fraudulent activities. However, one disadvantage of Revolut is its limited support for certain countries and currencies, which may be a drawback for users who require more flexibility in their financial management options.

Overall, while Revolut may not be a direct competitor to the proposed money management application, its success and popularity among users highlight the importance of user-friendly interfaces and innovative features in the financial technology industry.

`Screen Shots/Revolut/D_Revolut.jpg`

Figure 1.5: Revolut Dark main menu 1.

`Screen Shots/Revolut/W_Revolut.j`

Figure 1.6: Revolut main menu .

Chapter 2

Application Design

2.1 Element of design

2.1.1 Introduction in design

Application Design is an evolving field that leverages the principles of Design Theory and the practical approach of Design Thinking. It aims to strike a balance between form and function, aesthetics, and usability. [1] This chapter will delve into the intricate application of these elements in creating interactive, effective, and visually appealing digital solutions, while also addressing the challenges faced during the process. The task of designing applications is not a simple feat. It requires an understanding of the principles of Design Theory, such as balance, emphasis, rhythm, proportion, and unity. It also entails an in-depth comprehension of Design Thinking's core principles, including empathy for the user, the addition of solutions, and iterative refinement of designs.

Through this lens, Application Design transforms from a mere act of creating interfaces into a sophisticated process of solving problems and enhancing user experiences. This process requires both a strong foundation in theoretical principles and a human-centered approach that takes into account the context in which the application will be used.

In the scope of this chapter, we will dissect the core principles of Design Theory and Design Thinking and their respective roles in Application Design. We will illustrate how these principles and methods have been employed in real-world application design scenarios, underscoring their relevance and effectiveness. Furthermore, we also explore contemporary challenges and opportunities in Application Design, preparing readers to navigate and innovate in this ever-evolving field.

The chapter also elaborates on the iterative nature of Design Thinking and its inherent ability to learn from mistakes and progressively improve. We will discuss how Design Thinking and Application Design collectively ensure that digital solutions are not just functional but also user-centric.

In conclusion, this chapter will provide a comprehensive understanding of the symbiotic relationship between Design Theory, Design Thinking, and Application Design. The goal is to foster a robust theoretical and practical understanding that will enable designers to create applications that not only fulfill functional requirements but also resonate with users and deliver outstanding experiences.

Design theory is a broad and multifaceted field that encompasses many different areas of study, including graphic design, web design, user experience design, and more. At its core, design theory is concerned with the principles and elements that underlie all forms of visual communication and problem-solving. Design thinking is an essential component of design theory, and we will examine how this problem-solving approach is used in design to empathize with users, generate ideas, and iterate on solutions.

2.1.2 Spacing

The importance of maintaining constant spacing in design cannot be overstated. Consistent spacing, often referred to as spacing guidelines or whitespace management, plays a crucial role in enhancing the overall visual appeal and readability of a design. It ensures that elements are properly organized, allowing users to navigate and comprehend the content more effectively.

Spacing guidelines provide a sense of balance, harmony, and hierarchy within the design. By establishing consistent margins, padding, and line spacing, designers can create a cohesive and structured layout. This consistency helps users understand the relationships between different elements and makes the design more intuitive to navigate.

Proper spacing also contributes to the overall usability and accessibility of the design. Sufficient spacing between interactive elements, such as buttons or links, allows for easier selection and reduces the chance of accidental clicks. It enhances the user experience by providing a comfortable and uncluttered interface, which improves user satisfaction and engagement.

In addition, consistent spacing contributes to the perception of professionalism and attention to detail. It reflects a well-thought-out design process and gives the impression of a polished product. It shows that the designer has considered the overall user experience and visual aesthetics.

To maintain constant spacing, designers often establish grid systems or use design tools that provide layout guides and alignment features. This ensures that elements are aligned properly and follow the defined spacing guidelines consistently throughout the design.

1. **Margins**: Maintain consistent margins around the edges of the design layout. This ensures that the content is not too close to the edges, providing breathing space and preventing visual clutter.

2. **Padding**: Apply consistent padding within elements to create visual separation and prevent content from feeling cramped. This includes padding around text, images, buttons, and other interactive elements.

3. **Line spacing**: Use line spacing in text blocks to improve readability. Ensure that lines of text are adequately spaced to avoid overcrowding or excessive gaps, making the content more visually appealing and easier to read.

4. **Element spacing**: Maintain consistent spacing between elements such as buttons, icons, images, and paragraphs. This helps create a balanced and organized layout, allowing users to navigate and interact with the design more intuitively.

5. **Grid systems**: Use grid systems to establish a consistent layout structure. Grids help align elements and maintain consistent spacing throughout the design, contributing to a visually pleasing and harmonious composition.

6. **Responsive design**: Consider spacing guidelines for different screen sizes and devices. Optimize spacing to ensure that elements are properly proportioned and readable across various resolutions and orientations.

2.1.3 Grids

Grid systems are a fundamental tool in design that provides a framework for organizing and aligning elements within a layout. They comprise a series of vertical and horizontal lines that create a structured grid of intersecting points or cells.

Grids offer several benefits in design:

1. **Consistency and Structure**: Grids provide a consistent structure for organizing content, creating a sense of order and hierarchy. They help maintain visual consistency across different screens and pages within a project.

2. **Alignment**: Grids facilitate the precise alignment of elements, ensuring that they are visually harmonious. By aligning elements to the grid, designers can achieve a more polished and professional look.

3. **Proportional Layouts**: Grids enable designers to create proportional layouts by dividing the spacebar into well-defined columns and rows. This allows for more efficient use of space and helps maintain a balanced visual composition.

4. **Flexibility and Responsiveness**: Grids can be designed to be flexible, allowing for fluid layouts that adapt to different screen sizes and orientations. By defining responsive breakpoints within the grid system, designers can ensure that the design remains cohesive and functional across various devices.

5. **Efficiency and Workflow**: Grids provide a framework that streamlines the design process. They help designers make consistent decisions about spacing, sizing, and alignment, saving time and effort. Grid-based layouts also make it easier to maintain and update designs as the project evolves.

6. **Visual Hierarchy**: Grids can establish a clear visual hierarchy within a design. By assigning different columns or rows different levels of importance, designers can guide the viewer's attention and emphasize key elements.

Grid systems can be customized to suit the specific needs of a project. They can vary in terms of column widths, gutter sizes, and the number of subdivisions. Designers can choose between fixed grids, where the columns and gutters have predefined sizes, or fluid grids, which adapt to the screen space.

By incorporating grid systems into their design process, designers can create layouts that are visually pleasing, organized, and easy to navigate. Grids serve as a foundational tool that supports the overall structure and coherence of a design, enhancing both the aesthetics and functionality of the final product.

To integrate grids in our application, we can define a grid system with a specific number of columns and spacing between them. By aligning elements to this grid, we can ensure consistency and proper organization. [2]

`Graphics/Design/Grids/image.png`

Figure 2.1: Example of integrating grids in the application design

In this example, we have implemented an 8-column grid system with a 4px baseline. The grid helps to align and structure various elements within the application's layout. Each column spans an equal width, and the baseline ensures consistent vertical spacing between elements.

By dividing the available space into eight columns, we create a proportional layout that can accommodate different components. The 4px baseline establishes a consistent vertical rhythm, ensuring that elements align harmoniously along the vertical axis.

The grid can guide the placement of elements such as text blocks, images, buttons, and other interactive components. By aligning these elements to the grid, we achieve a visually balanced and organized layout.

To ensure proper integration of the grid, designers can use design tools or frameworks that provide grid functionality. These tools assist in snapping elements to the grid and provide visual cues for aligning components accurately.

Integrating an 8-column, 4px baseline grid brings structure, consistency, and visual harmony to the application's design. It enhances readability, organization, and overall user experience. By following the grid guidelines, designers can create visually appealing and functional interfaces that resonate with users.

2.1.4 Typography

Typography is an essential element of design that encompasses the art and technique of arranging typefaces to create visually appealing and readable text. It plays a crucial role in various forms of communication, such as print media, digital interfaces, advertising, and branding.

At its core, typography is about selecting, organizing, and presenting typefaces in a deliberate and thoughtful manner. It involves making informed choices about font styles, sizes, spacing, alignment, and hierarchy to effectively convey information and evoke specific emotions or responses from the audience.

The primary goal of typography is to ensure readability and legibility. By choosing appropriate typefaces and adjusting factors like font size, line spacing, and letter spacing, typographers strive to create a text that is easy to read and comprehend. They consider the intended audience, context, and medium to optimize the reading experience.

Beyond readability, typography also plays a significant role in visual communication and aesthetics. It can evoke various moods, establish brand identities, and guide the viewer's eye through a design. The arrangement of typefaces, hierarchy, and whitespace contribute to the overall visual impact and message of a piece.

Typography has a rich history dating back to the invention of movable type in the 15th century, which revolutionized printing and led to the proliferation of books. Over the centuries, typographic styles and practices have evolved, reflecting cultural shifts, artistic movements, and technological advancements.

In the digital age, typography has become even more versatile and accessible. Designers can choose from an extensive range of typefaces, customize them with digital tools, and experiment with dynamic and interactive typography in digital interfaces.

Understanding the principles and techniques of typography empowers designers to effectively communicate through text. By employing typography thoughtfully, they can enhance readability, express ideas, evoke emotions, and create visually engaging designs that capture and hold the viewer's attention.

In this chapter, we will explore the fundamental principles, best practices, and practical applications of typography in [specify the context of your paper]. Through this exploration, we aim to show the significance of typography in design and its impact on effective visual communication.

- **Font Selection**: Carefully selecting appropriate fonts that align with the purpose and tone of your paper while considering factors such as readability, professionalism, and brand consistency.

- **Hierarchy**: Establishing a clear visual hierarchy within the typography to guide readers' attention and emphasize key points. This can be achieved through variations in font sizes, weights, and styles, ensuring that important information stands out.

- **Readability**: Ensuring that the text is easily readable by selecting suitable font sizes, line spacing, and letter spacing. Avoiding overcrowding of text and choosing legible typefaces contribute to the overall readability of your paper.

- **Alignment**: Employing proper text alignment, such as left-aligned or justified, to enhance readability and maintain a polished appearance throughout the paper. Consistency in alignment creates a cohesive and visually pleasing reading experience.

- **Whitespace**: Using whitespace effectively to provide visual breathing space between elements and enhance overall clarity. Proper use of whitespace contributes to improved readability and helps organize content in a balanced manner.

- **Consistency**: Maintaining consistency in typography throughout your paper by using consistent font choices, sizes, styles, and alignment. Consistency creates a cohesive visual identity and fosters a professional appearance.

- **Brand Identity**: Using typography to reflect and reinforce your paper's brand identity or academic institution. Selecting fonts that align with the overall style and values of your paper can create a consistent and recognizable visual identity.

- **Visual Impact**: Employing creative typographic treatments, such as emphasis through bold or italicized text, varying font sizes for headings, or using typographic hierarchy to create visual interest and impact in your paper.

- **Effective Communication**: Leveraging typography to enhance the communication of information in your paper. By employing proper font choices, alignment, and hierarchy, you can effectively convey your research findings and ideas to your readers.

Figure 2.2: Typography template
[1]

2.1.5 Color

Color is a fundamental element of design that holds immense significance in conveying messages, evoking emotions, and creating visual interest. It is a powerful tool that can capture attention, establish a visual hierarchy, and elevate the overall user experience. Designers meticulously consider various aspects when working with color to ensure their designs effectively communicate their intended message and captivate their audience.

The color goes beyond its visual appeal. It possesses the ability to evoke emotions, create associations, and convey meaning. Designers delve into the psychology of color, understanding the psychological responses that different colors elicit in individuals. They explore the emotional impact of colors and handpick hues that align with their desired emotional response and convey their intended message effectively.

1. **Psychology of Color**: Designers consider the psychological associations and emotional responses associated with different colors. By understanding the psychology of color, designers can select colors that align with their intended message and evoke the desired emotions in their audience.

2. **Color Harmony**: Achieving color harmony involves selecting colors that complement each other and create a visually pleasing composition. Designers carefully choose color combinations that work well together, ensuring a balanced and harmonious visual experience.

3. **Contrast**: Contrast is an essential consideration in color selection. Designers use contrast to create visual interest, distinguish important elements, and improve readability. By carefully balancing colors with different levels of contrast, designers ensure that key information stands out effectively.

4. **Color Symbolism**: Colors carry symbolic meanings that can vary across cultures and contexts. Designers consider the cultural connotations associated with different colors to ensure their choices align with the intended message and avoid any potential misinterpretation or conflicts.

5. **Color Accessibility**: Designers prioritize color accessibility by considering the needs of all users, including those with visual impairments or color blindness. They ensure sufficient color contrast for readability and provide alternative means of conveying information beyond color alone.

6. **Color Consistency**: Consistency in color usage helps establish a strong visual identity and brand recognition. Designers define a color palette and adhere to it consistently, ensuring that colors remain cohesive across various touch points and materials.

7. **Color and User Experience**: Colors significantly impact the user experience. Designers leverage color to influence users' perception of usability, credibility, and enjoyment. By employing colors strategically, designers create engaging and intuitive experiences for their audience.

8. **Color Trends and Brand Differentiation**: Designers stay informed about color trends to create contemporary and visually appealing designs. They also

consider unique color choices or combinations to differentiate their brand or product from competitors, fostering brand recognition and recall.

Graphics/Design/Color Pallet.png

Figure 2.3: Typography template

2.2 Design choices

The design of the Moneager application was approved Development 3.1 Database Structure 3.2 Endpoints (GraphQL) 19 ched with the user in mind, aiming to create an intuitive and visually pleasing experience. The following design principles were taken into consideration during the development process:

Simplicity: The application was designed to be as simple and user-friendly as possible, with a clean and uncluttered layout. This allows users to easily navigate the app and access the features they need without feeling overwhelmed.

Consistency: The visual elements and design patterns used throughout the application are consistent, creating a cohesive and recognizable user interface. This enhances the user experience by reducing confusion and cognitive load.

Usability: The user interface was designed to be intuitive and easy to use, with clear navigation and visual cues to guide the user. The goal was to create an application that can be used by anyone, regardless of their level of technical expertise.

Visual Appeal: The application was designed with a modern and visually appealing layout, using a consistent color scheme and typography to create a polished and professional look. This was done to ensure that users enjoy using the application and are motivated to continue using it.

In terms of specific design decisions, the application features a minimalistic design with a focus on usability. The layout is centered around the main menu, which provides easy access to all of the application's features. The use of icons and visual elements is consistent throughout the application, enhancing the user experience by making it easy to recognize and navigate different sections of the app.

Another design choice is the use of a modern and sleek layout with a balanced amount of features to ensure simplicity and ease of use for all users. The use of a simple and intuitive navigation system, with clear visual elements and typography, is also prioritized to enhance user experience. Additionally, the incorporation of a high-contrast option and alternative text for images ensures accessibility for all users.

Overall, the design choices made for the Moneager application were focused on creating a user-friendly and visually appealing experience, with a balance between simplicity and functionality.

Design choices are crucial for any mobile application, especially for a financial management application that requires users to engage with the app frequently. One of the primary design choices for this application is the elimination of a drawer menu, which can often clutter the user interface and make navigation confusing. Instead, a user-friendly bottom tab navigation system is implemented to ensure ease of use and simplify the user interface.

The design layout of the application has been carefully crafted to ensure a modern and minimalistic look, while still providing users with the necessary features to manage their finances efficiently. The use of a clean and consistent layout with adequate white space and appropriate visual elements is crucial for a user-friendly interface. This design ensures that users can easily navigate the application and find the information they need quickly and efficiently.

First of all the design is quite a complex step so I tried to iterate multiple designs. In the next Subsection I will describe my design motivation for this design and what are reasons why I had not continued with it.

2.2.1 First Design Iteration

When selecting colors for different transaction types in the application, a set of guidelines and considerations were followed. The primary aim was to ensure that the chosen colors were easily distinguishable. The colors were selected based on their association with the corresponding transaction types commonly recognized by users.

In the color selection for different transaction types in the application, the following associations were made:

Certainly! Here's the information presented in a list format using LaTeX code:

- Income transactions: **Green** - commonly associated with money and growth.
- Expenses: **Red** - chosen due to its association with negative actions or consequences.
- Investing: **Yellow** or **gold** - symbolizing wealth and luxury.
- Savings: **Blue** - selected for its connotations of stability and reliability.

These color choices were made to ensure easy distinguishing ability and to align with the recognized associations of these colors by users.

It is important to note that, during the color selection process, the goal was to create a visually balanced composition that would enhance the application's overall usability and modernity. The chosen colors were carefully implemented to maintain simplicity while incorporating enough features. The resulting design layout, illustrated in the accompanying Figma file, showcases the successful execution of these principles.

Upon reflection, it was realized that some color compositions were uneven and required further changes to ensure a harmonious visual presentation.

Below, we provide some illustrations that showcase the design layout of the application. The illustrations demonstrate how the application has been developed to prioritize simplicity, usability, and modernity, with a balanced amount of features and a sleek design.

Graphics/Design/TM_E.png

Figure 2.4: Expense Transaction Screen

Graphics/Design/TM_Menu.png

Figure 2.5: Transaction Creator Menu

Graphics/Design/TM_I.png

Figure 2.6: Income Transaction Screen

After this short brainstorming session, a design layout in Figma was created. Upon reflection, it was realized that the color composition appeared to be uneven. The color palette used in the design was diverse and did not align with the simplistic and easy-to-use interface envisioned.

2.2.2 Second Design Iteration

Purple was chosen as the primary color for the transaction menu, taking into consideration factors such as uniqueness and elegance. It was believed that *purple* could add a touch of *luxury, creativity, and sophistication* to the app, which is not commonly seen in financial apps.

To ensure cohesiveness, different shades of *purple* were utilized to represent various transaction types. A lighter shade of *purple* was assigned to income transactions, while a darker shade was designated for expenses. Additionally, the author considered using other colors, such as *green* or *blue*, to represent additional transaction types like savings or investing.

The author made sure that the chosen *purple* color was easily distinguishable from other colors used in the app and did not clash with other elements on the screen. To achieve this, a color wheel or palette generator was employed to find complementary colors to use alongside *purple*.

As part of the design process, a simple Profile Page was created for the application. The author found that the new design was a better fit for their desired simplistic approach and aligned well with their overall vision.

In this second design iteration, the application's color scheme was carefully chosen to create a visually appealing and cohesive user interface. The following images showcase various perennially designed templates that were incorporated into the application, enhancing its overall aesthetic and user experience. These design templates provide consistent styling and layout for key components such as buttons, forms, and cards, creating a unified and professional look throughout the application.

`Graphics/FinalDesigns/FirstMenu.png`

Figure 2.7: First Menu

`Graphics/FinalDesigns/LoginSc`

Figure 2.8: Login Screen

Chapter 3

Application Development

In this chapter, we will focus on the frontend development of our money management application. We will discuss the technologies and tools used, the implementation of the user interface, and the enhancements made to improve the user experience.

3.1 Technologies and Tools

3.1.1 Technologies

In this paper, the following technologies have been employed for development:

- **React Native**: React Native is a TypeScript framework used for building mobile applications. It allows developers to create native-like apps for iOS and Android platforms using a single codebase. React Native leverages the power of TypeScript and React to deliver high-performance and visually appealing mobile applications [3].

- **Expo**: Expo is a set of tools and services built around React Native that simplifies the development process. It provides an integrated development environment (IDE) and various libraries to streamline the frontend development workflow. Expo offers features such as hot reloading, over-the-air updates, and built-in APIs that enhance productivity and app functionality [4].

- **TypeScript**: TypeScript is a statically typed superset of JavaScript that brings additional features and benefits to the development process. It introduces static typing, enabling developers to catch potential errors and bugs early in the development phase. TypeScript enhances code readability, provides better tooling support, and improves the overall maintainability of the codebase [5].

- **GraphQL**: GraphQL is a query language for APIs and a runtime for executing queries with existing data. It allows clients to specify the data they need, and the server responds with only the requested data, reducing over-fetching and under-fetching of data. GraphQL simplifies data fetching and manipulation, improves performance, and provides a flexible and efficient approach to API development [6].

By utilizing React Native, Expo, TypeScript, and GraphQL, this paper's development ensures efficient cross-platform app development, simplified workflow, enhanced productivity, improved code quality, and effective data handling. These technologies work together to deliver a robust and user-friendly mobile application experience.

3.1.2 Tools

In the frontend development process, the following tools have been utilized:

- **Figma**: Figma is a cloud-based design and prototyping tool that enables collaborative interface design. It provides a user-friendly interface for creating and sharing design mockups, wireframes, and interactive prototypes. Figma allows designers and developers to collaborate in real time, making it easier to iterate on the design and ensure a cohesive user experience [7].

- **WebStorm**: WebStorm is an integrated development environment (IDE) specifically designed for web development. It offers a wide range of features, including intelligent code completion, code analysis, and debugging capabilities. WebStorm provides a smooth development experience for writing and managing TypeScript, JavaScript, HTML, and CSS code, making it an ideal choice for frontend development. [8]

- **Expo Go**: Expo Go is a mobile app that enables developers to test and preview their React Native applications in real-time on physical devices. It allows developers to quickly iterate and view changes made to the app without the need for complicated setup or app building processes. Expo Go provides a convenient way to test app functionality and UI responsiveness during the development phase [9].

- **React Developer Tools**: React Developer Tools is a browser extension that enhances the development experience for React applications. It provides a set of debugging and inspection tools specifically tailored for React components. Developers can inspect component hierarchies, view state and props, and track component updates in real-time. React Developer Tools greatly aids in understanding and troubleshooting React components during frontend development [10].

- **Apollo Studio**: Apollo Studio is a powerful tool for managing, monitoring, and analyzing GraphQL APIs. It provides a comprehensive platform that enables developers to design, document, and collaborate on GraphQL schemas, track API usage, and gain insights into performance and errors. With Apollo Studio, developers can easily optimize their GraphQL APIs, debug issues, and ensure efficient data delivery to frontend applications. It offers features such as schema management, query performance tracking, error tracking, and integration with popular development tools. By incorporating Apollo Studio into the frontend development process, developers can enhance the overall GraphQL API development experience and deliver high-quality and performant applications [11].

These tools enhance the frontend development process, enabling developers to streamline their workflow, ensure code quality, and deliver high-quality mobile applications. Each tool plays a unique role in supporting different aspects of frontend development, from design and coding to testing and monitoring. By leveraging these tools, developers can effectively collaborate, debug applications, perform real-time testing, optimize GraphQL APIs, and maintain code quality throughout the development lifecycle.

3.2 User Experience

In the context of app development, user flow refers to the series of steps a user takes while navigating through an application or interacting with its features. It encompasses the paths, actions, and interactions that users follow to accomplish their goals within the app.

User flow plays a crucial role in creating a seamless and intuitive user experience. By understanding and designing effective user flows, developers can guide users through the app's functionalities, ensuring they can easily access the desired information or perform specific actions.

In this section, the user flow of the application will be explored and analyzed, focusing on the key screens and interactions that users encounter during their journey. By examining the user flow, potential bottlenecks can be identified, usability can be improved, and the overall user experience can be optimized.

3.2.1 User Flow: Transaction

The registration process allows users to create an account and gain access to the application's features. It involves the following steps:

1. User Arrival: The user arrives at the registration screen and is presented with a registration form.

2. Input Fields: The registration form includes fields for name, email, password, and any additional required information.

3. Validation: The user's input is validated to ensure accuracy and completeness. Error messages are displayed for any missing or invalid information.

4. Create Account: Once all required information is provided, the user can proceed to create their account by clicking the "Create Account" button.

5. Confirmation: A confirmation message is displayed to notify the user that their account has been successfully created.

`Graphics/UserFlows/RegisterUserFlow.png`

Figure 3.1: User Flow: Registration Process

In Figure 3.1, the image illustrates the sequential steps involved in the registration process, guiding the user from arrival to account creation.

This user flow prompt provides an overview of the registration process and complements the visual representation in the accompanying image.

3.2.2 User Flow: Login

The login process allows registered users to access their accounts and utilize the application's features. It involves the following steps:

1. User Arrival: The user arrives at the login screen and is presented with a login form.

2. Input Fields: The login form includes fields for email and password.

3. Validation: The user's input is validated to ensure accuracy and completeness. Error messages are displayed for any missing or incorrect information.

4. Login: Once the user enters the correct email and password, they can proceed to login by clicking the "Login" button.

5. Authentication: The user's credentials are authenticated against the stored account information.

6. Redirect: Upon successful login, the user is redirected to the main page/dashboard of the application.

`Graphics/UserFlows/LoginUserFlow.png`

Figure 3.2: User Flow: Login Process

In Figure 3.2, the image illustrates the sequential steps involved in the login process, guiding the user from arrival to successful authentication.

This user flow prompt provides an overview of the login process and complements the visual representation in the accompanying image.

3.2.3 User flow: Transaction

A transaction is a key interaction in the application where users can perform financial operations such as recording expenses or income. Let's explore the user flow for a typical transaction:

Step 1: The user navigates to the transaction section of the app.

Step 2: The user is presented with a form to enter transaction details, including the transaction type (expense or income), amount, category, and additional notes.

Step 3: After filling in the required information, the user clicks on the "Submit" button to proceed.

Step 4: The app validates the transaction data and performs any necessary calculations or checks.

Step 5: If the transaction is successful, the app displays a confirmation message or animation to indicate that the transaction has been recorded.

`Graphics/UserFlows/TransactionUserFlow.png`

Figure 3.3: User Flow for a Transaction

Figure 3.16 illustrates the user flow for a transaction in a visual format. Users follow these steps to complete a transaction and ensure accurate record-keeping of their financial activities.

The transaction process is designed to be intuitive and user-friendly, allowing users to quickly and efficiently record their expenses and income within the app.

3.3 Application Screens

- Login Screen: Users can access the app by entering their login credentials on this screen (Figure 3.4).
- Login Error - Password Missing: This screen indicates an error during the login process due to a missing password (Figure 3.5).

Figure 3.4: Login Screen

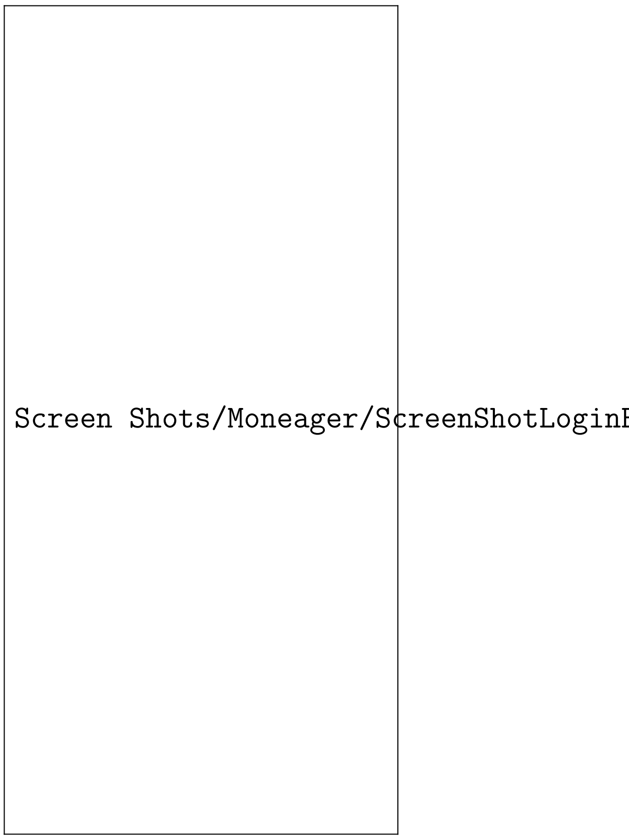

Figure 3.5: Login Error - Password Missing

- Register Screen: This screen presents a user registration form where new users can provide their information to create an account (Figure 3.6).
- User Registration Error: This screen alerts users about an error that occurred during the registration process due to invalid user registration data (Figure 3.7).

Screen Shots/Moneager/RegisterScreen.png

Figure 3.6: Register Screen

Screen Shots/Moneager/ScreenShotRegist

Figure 3.7: User Registration Error

3.3.1 Transaction Screens

- Transaction History Screen: This screen provides a comprehensive list of the user's past transactions (Figure 3.8).

- Transaction Creator Screen: This screen enables users to create new transactions (Figure 3.9).

- Invalid Transaction Example: This screen displays an error message when the user attempts to enter an invalid transaction (Figure 3.10).

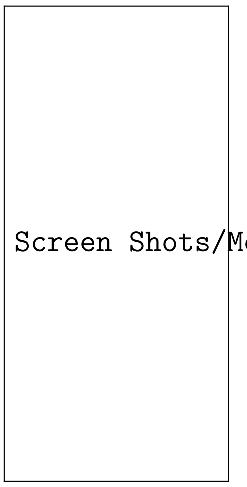

Figure 3.8: Transaction History Screen

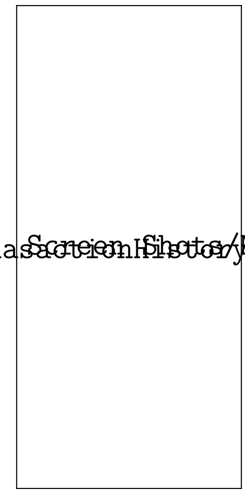

Figure 3.9: Transaction Creator Screen

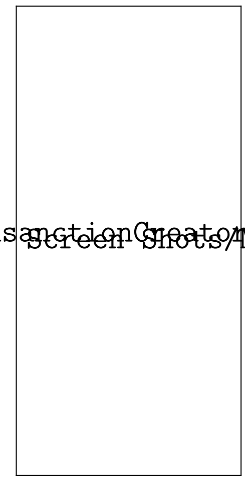

Figure 3.10: Invalid Transaction Example

3.3.2 Additional Screens

- Transaction Screen Screen: This screen serves as the app's transaction creator screen (Figure 3.11).

- Settings Screen: This screen allows the user to modify the settings of the application (Figure 3.12).

- Statistics Screen: Users can see a statistical overview of their financial activities on this screen (Figure 3.13).

- Wallet Balance Screen: This screen displays the current balance of the user's wallet (Figure 3.14).

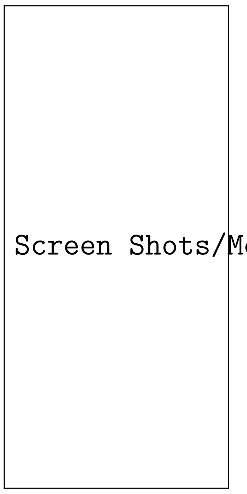

Figure 3.11: Transaction Screen

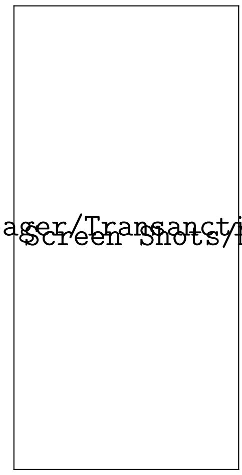

Figure 3.12: Settings Screen

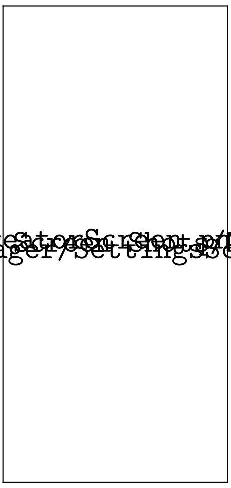

Figure 3.13: Statistics Screen

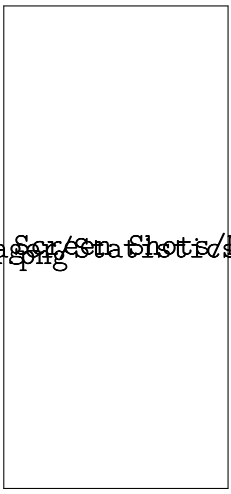

Figure 3.14: Wallet Balance Screen

3.4 Project Structure

The provided directory structure represents the organization of files and folders in the front-end development of the application. It encompasses essential aspects such as assets, components, constants, navigation, and services. This structured approach facilitates modularity, code re-usability, and maintainability, enabling developers to effectively manage resources and functionalities throughout the development process.

It is important to note that only a subset of files and folders are shown in the provided structure for the sake of feasibility. The actual structure may include additional files and folders depending on the specific requirements of the project. The shown files provide an overview of the key elements present in the front-end development, while other supporting files and folders are assumed to be present to ensure the smooth functioning of the application.

- `assets`: The `assets` folder is used to store various asset files such as images, icons, and splash screens.
  - `images`: This subfolder within `assets` is specifically used to store different images used in the application.
- `components`: The `components` folder contains different components used in the application.
  - `screens`: This subfolder within `components` is dedicated to screen components.
  - `shared`: This subfolder within `components` contains shared components.
  - `ui`: This subfolder within `components` is specifically used for UI components.

- `constants`: The `constants` folder is used to store files related to constants or configurations.
  - `colors.tsx`: This file within the `constants` folder defines color constants used in the application.
  - `fonts`: This subfolder within `constants` is dedicated to font-related files.
  - `graphql`: This subfolder within `constants` is specifically used for GraphQL-related configurations.
- `navigation`: The `navigation` folder contains files related to navigation within the application.
  - `accountCreationStackNavigator.tsx`: This file defines the stack navigator specifically for the account creation process.
  - `skect_bottomTabNavigator.tsx`: This file defines the bottom tab navigator used for the application.
- `services`: The `services` folder is used to store files related to various services or utilities.
  - `AuthHandler.tsx`: This file within the `services` folder handles authentication-related functionality.
  - `CustomResourceLoading.tsx`: This file within the `services` folder handles custom resource loading.
  - `ErrorHandler`: This subfolder within `services` contains files related to error handling.

3.4.1 Applications core logic

The main UML diagram, as shown in Figure 3.16, provides an overview of the core logic and structure of the application. It represents the relationships and interactions between various components and modules, highlighting the flow of data and control within the system.

This comprehensive diagram serves as a visual representation of the application's architecture and helps developers and stakeholders gain a better understanding of how different components and modules interact with each other. It provides insights into the major functionalities, data flow, and dependencies within the system.

The short outline diagram, shown in Figure 3.16, provides a condensed overview of the application's structure and major components. It offers a high-level representation of the key modules and their relationships, serving as a quick reference for understanding the overall architecture of the system.

In this subsection, the main file App.tsx is utilized and extended to create the core logic of the application. The App.tsx file serves as the entry point for the application and is responsible for initializing the required components and configuring the overall structure of the app. The code and its functionality can be described as follows:

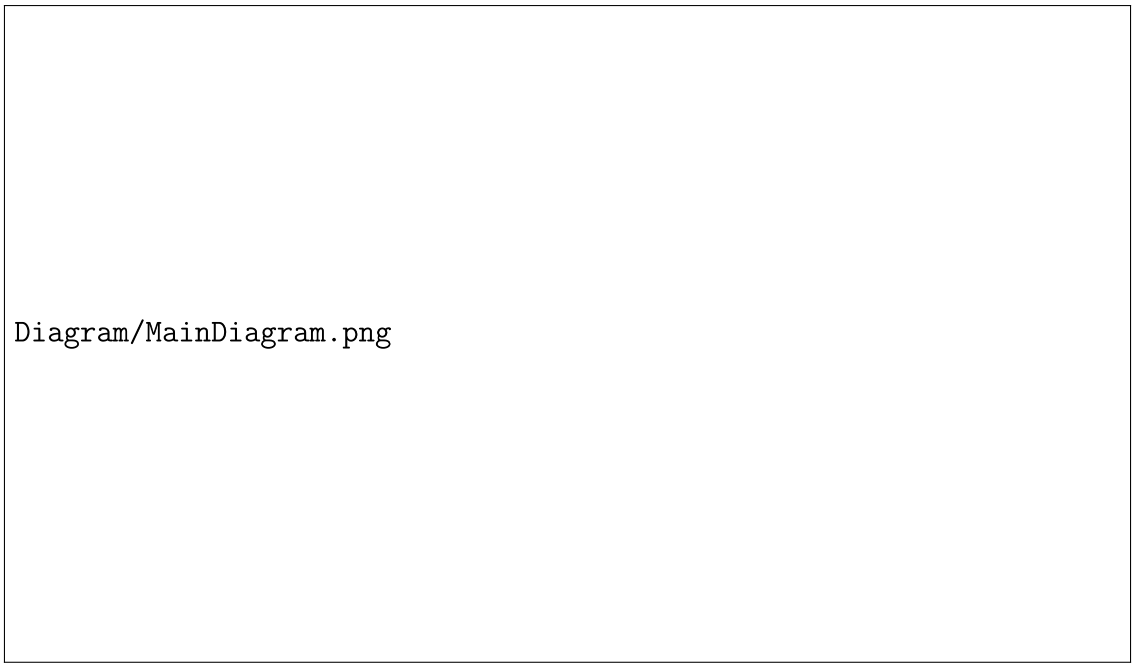

Figure 3.15: Main UML diagram

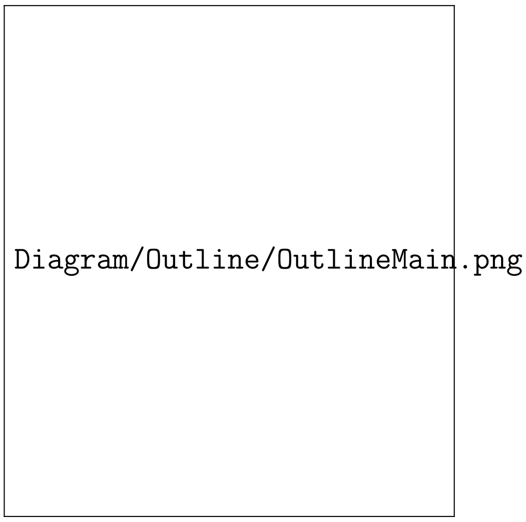

Figure 3.16: Short Outline diagram

```
// Backend Link
const authLink = setContext((_, { headers }) => {
    // Get the JWT token
    const token = jwt();
    return {
        headers: {
            ...headers,
            // Add the token to the Authorization header
            authorization: token ? ‘Bearer ${token}‘ : "",
        }
    }
});
```

The *authLink* is a middleware function that adds the JWT token to the headers of outgoing requests. It uses the *setContext* function from Apollo Link to modify the request headers and includes the token in the Authorization header.

```
// GraphQL Endpoint
const httpLink = createHttpLink({
    uri: 'http://your\_graphql\_server:1337/graphql',
});
```

The *httpLink* defines the GraphQL endpoint URL where the client sends its requests. It uses the *createHttpLink* function from Apollo Link to create a link that connects to the specified GraphQL server.

```
const Stack = createNativeStackNavigator();

// Remember the client Cache
const client = new ApolloClient({
    link: authLink.concat(httpLink),
    cache: new InMemoryCache()
});
```

The *Stack* is a component from the React Navigation library that manages the navigation stack. The *createNativeStackNavigator* function is used to create a stack navigator instance.

The *client* is an instance of the *ApolloClient* that provides the connection to the GraphQL server. It is configured with the *authLink* and *httpLink* to handle authentication and network requests. The cache is created using the *InMemoryCache* class, which manages the client-side cache for storing query results.

```
// Prevent unnecessary redraws
export const AppStack = memo(() => {
    return (
        <Stack.Navigator>
            <Stack.Screen name="LoginScreen" component={LoginScreen
    } options={{ headerShown: false }}/>
            <Stack.Screen name="RegisterScreen" component={
    RegisterScreen} options={{ headerShown: false }}/>
        </Stack.Navigator>
    );
});
```

The *AppStack* is a React component defined using the *memo* function to optimize rendering performance by preventing unnecessary redraws. It represents the stack navigator for the app screens. It includes screen components for login and registration, where the *headerShown* option is set to *false* to hide the header.

```
// Navigation checker
export const Navigation = () => {
    const isLoggedIn = useReactiveVar(jwt);
    return (
        <NavigationContainer>
            {!isLoggedIn ? <AppStack/> : <AccountCreationNavigator
  />}
        </NavigationContainer>
    );
}
```

The *Navigation* component serves as the entry point for the app's navigation. It uses the *NavigationContainer* component from React Navigation to wrap the app's navigation hierarchy. It conditionally renders the *AppStack* component or the *AccountCreationNavigator* component based on whether the user is logged in or not.

```
export default function App() {
    const isLoading = useCustomResourceLoading();

    return isLoading ? null : (
        <ApolloProvider client={client}>
            <SafeAreaProvider style={GlobalStyles.droidSafeArea}>
                <Navigation/>
            </SafeAreaProvider>
        </ApolloProvider>
    );
}
```

The *App* component is the root component of the app. It wraps the app with the *ApolloProvider* component from Apollo Client, providing the client instance as the GraphQL client for the app.

It also wraps the app with the *SafeAreaProvider* component from *react-native-safe-area-context* to handle safe area insets on different devices. The *Navigation* component is rendered inside the *SafeAreaProvider*.

The *isLoading* variable is used to conditionally render *null* during the initial loading of custom resources.

Overall, this code sets up the necessary components and configurations for Apollo Client, React Navigation, and the app's navigation structure. It handles authentication, network requests, and provides the necessary context for making GraphQL queries and mutations.

The *useCustomResourceLoading* function is a custom React Hook that is designed to manage the loading of custom resources within a React application. This hook helps ensure that the necessary resources, such as fonts, are loaded before the application renders its content, providing a smoother user experience.

The code snippet demonstrates the implementation of the *useCustomResourceLoading* function:

```
export const useCustomResourceLoading = () => {
    let [fontsLoaded] = useFonts({
        Inter_400Regular,
        Inter_500Medium,
        NanumBrushScript_400Regular,
        Lora_400Regular,
        Lora_600SemiBold,
    });
}
```

In this code block, the *useFonts* hook is utilized from the *expo-font* package. It is responsible for loading the specified custom fonts required by the application. The *fontsLoaded* variable represents the loading state of the fonts.

```
const [appIsReady, setAppIsReady] = useState(false);

useEffect(() => {
    async function prepare() {
        try {
            await SplashScreen.preventAutoHideAsync();
        } catch (err) {
            console.warn(err);
        } finally {
            setAppIsReady(true);
        }
    }

    prepare();
}, []);
    return (!fontsLoaded || !appIsReady);
};
```

The *appIsReady* state variable and its corresponding setter function, *setAppIsReady*, are created using the *useState* hook. Initially, the *appIsReady* state is set to false.

Within the *useEffect* hook, an asynchronous function named prepare is defined. This function attempts to prevent the splash screen from automatically hiding by calling the *SplashScreen.preventAutoHideAsync()* method. If any errors occur during this process, they are logged to the console. Finally, regardless of the outcome, the *setAppIsReady* function is invoked to set the *appIsReady* state to true.

Finally, the *useCustomResourceLoading* hook returns a boolean value that determines whether the custom resources are still being loaded. If either the fonts are not yet loaded or the app is not ready, the hook will return true.

By using the *useCustomResourceLoading* hook within a React application, developers can ensure that the necessary custom resources are loaded before rendering the app's content. This helps provide a seamless and visually consistent user experience.

Chapter 4

Backend Development

In this chapter, we will explore the backend development of our money management application. We will discuss the various technologies and methodologies employed to create a robust and efficient backend system. Specifically, we will focus on the following key components:

4.1 Introduction to Backend

In the backend development process, several key elements have been utilized:

- **Strapi**: Strapi is an open-source headless CMS (Content Management System) that provides a powerful and flexible backend framework for building API-driven applications. It offers a user-friendly interface for content creation and management, allowing developers to define custom content types and relationships. Strapi handles authentication, authorization, and data validation, making it easier to develop robust and secure APIs. With its extensible plugin system and GraphQL support, Strapi enables developers to create custom APIs tailored to their application's specific needs. [12]

- **GraphQL**: GraphQL is a query language for APIs that enables clients to request specific data from the server. It provides a more efficient and flexible way of fetching data compared to traditional REST APIs. GraphQL allows clients to specify the exact data requirements, reducing over-fetching and under-fetching of data. It also enables multiple data sources to be combined into a single response, simplifying the data fetching process. By integrating GraphQL into the backend, developers can create efficient and precise APIs that optimize data transfer between the backend and frontend. [6]

- **Apollo Server**: Apollo Server is an open-source GraphQL server implementation that simplifies the process of building GraphQL APIs. It provides a set of tools and utilities for defining GraphQL schemas, resolvers, and data sources. Apollo Server supports various features, such as data caching, real-time subscriptions, and error handling. By using Apollo Server, developers can easily create scalable and performant GraphQL APIs that integrate seamlessly with their backend systems. [13]

- **SQLite**: SQLite is a lightweight and self-contained relational database management system. It is widely used in mobile and web applications due to its simplicity, portability, and low resource footprint. SQLite offers SQL support and provides ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring data integrity and reliability. By utilizing SQLite as the backend database, developers can efficiently store and retrieve data for their applications. [14]

These backend elements, including Strapi, GraphQL, Apollo Server, and SQLite, work together to provide a robust and scalable backend infrastructure. Strapi acts as the CMS and API generator, enabling content management and API development. GraphQL serves as the query language and runtime for APIs, allowing efficient data fetching and integration of multiple data sources. Apollo Server simplifies the creation of GraphQL APIs, providing tools and utilities for server-side development. SQLite acts as the backend database, ensuring reliable data storage and retrieval. By integrating these elements, developers can build powerful and flexible backend systems that communicate seamlessly with the frontend, enabling the development of dynamic and data-driven applications.

4.2 UML Class Diagram

The UML class diagram represents the data structure of the ”Moneager” application. It consists of several classes that define the entities and their relationships.

- **Users**: This class represents the users of the application. It has attributes such as `id` (unique identifier), `username`, `email`, `password`, and `preferences`.
- **Wallets**: The `Wallets` class represents the wallets associated with each user. It has attributes like `id` (unique identifier), `userId` (foreign key referencing the user), and `name` (the name of the wallet).
- **Transactions**: The `Transactions` class represents the individual transactions made within each wallet. It has attributes such as `id` (unique identifier), `walletId` (foreign key referencing the wallet), `date` (the date of the transaction), `description` (a brief description of the transaction), `category` (the category of the transaction), and `amount` (the amount of the transaction).
- **Categories**: The `Categories` class represents the different categories that can be assigned to transactions. It has attributes like `id` (unique identifier) and `name` (the name of the category).

The relationships between the classes are defined as follows:

- Users have a one-to-many relationship with Wallets, indicating that a user can have multiple wallets.
- Wallets have a one-to-many relationship with Transactions, indicating that a wallet can have multiple transactions.
- Transactions have a one-to-one relationship with Categories, indicating that each transaction is associated with a single category.

`Diagram/DatabaseUmlDiagram_Reports.png`

Figure 4.1: UML Class Diagram for the Moneager Application

4.3 Strapi Backend Schema Development

In this section, we will explore the backend development of our money management application. We will discuss the various components and provide an overview of the backend architecture.

4.3.1 API Endpoints

We have designed several API endpoints to facilitate data retrieval and manipulation. Here are some examples:

Figure 4.2 demonstrates the process of making a JWT authentication request using Postman. The request is sent to the specified API endpoint, and upon successful authentication, a JSON Web Token (JWT) is returned.

In Figure 4.3, we showcase an example of a login request using Postman. The user provides their credentials (e.g., email and password), and the backend verifies the information and returns the appropriate response.

4.3.2 Database Schema

Our backend system employs a relational database to store user transactions and related information. The following diagrams illustrate the database schema:

Figure 4.4 depicts the relationship between transactions and wallets in the database. Each transaction is associated with a specific wallet, allowing for proper organization and tracking.

Strapi/ExampleJWT Get Postman.png

Figure 4.2: Example of making a JWT authentication request using Postman.

Strapi/Example of Login using PostMan.png

Figure 4.3: Example of a login request using Postman.

`Strapi/Rel/TransactionToWallet.png`

Figure 4.4: Relationship between transactions and wallets in the database.

`Strapi/Rel/UserToWallet.png`

Figure 4.5: Relationship between users and wallets in the database.

The relationship between users and wallets is illustrated in Figure 4.5. Each user can have multiple wallets, providing flexibility and customization options.

4.3.3 Data Overview

To provide a better understanding of the backend data, we have included some overview diagrams:

`Strapi/TranOverview.png`

Figure 4.6: Overview of user transactions in the application.

Figure 4.6 presents an overview of user transactions within the application. It displays the transaction details and allows for easy tracking and analysis.

Lastly, Figure 4.7 provides an overview of user information stored in the backend. It includes details such as user profiles, authentication data, and associated wallets.

Figure 4.7: Overview of user information in the application.

4.3.4 Conclusion

In conclusion, the backend development of our money management application encompasses various API endpoints, a well-defined database schema, and a comprehensive overview of user transactions and data. These components work together to ensure a robust and efficient backend system. By leveraging the power of APIs and a relational database, we enable seamless communication between the frontend and backend, providing users with a seamless and intuitive experience.

4.4 GraphQL

GraphQL is a query language for APIs that offers efficient data fetching and flexibility in data querying. We will introduce the concept of GraphQL, discuss its advantages over traditional REST APIs, and demonstrate how to implement GraphQL in our backend.

In this paper, GraphQL was utilized as the query language and runtime for our money management application's backend development. GraphQL played a crucial role in efficiently fetching and manipulating data between the frontend and backend.

By leveraging GraphQL, we were able to establish a clear and structured communication protocol between the client and server. The GraphQL schema defined the available data types, queries, and mutations, serving as a contract between the frontend and backend teams. This schema provided a unified and intuitive way for clients to request specific data and operations.

One of the key benefits of GraphQL was its ability to prevent over-fetching and under-fetching of data. Clients could precisely specify the data they needed in their queries, reducing unnecessary data transfer and improving performance. This granular control over data fetching helped optimize the application's performance and minimize network bandwidth usage.

These are some of the transation used in the aplication:

- **CREATE_WALLET_MUTATION**:
  - Mutation for creating a new wallet.
  - Input: `WalletInput` object.
  - Returns: ID and attributes (amount, name, type) of the created wallet.

```
        export const CREATE_WALLET_MUTATION = gql`
            mutation CreateWallet($data: WalletInput!) {
                createWallet(data: $data) {
                    data {
                        id
                        attributes {
                            amount
                            name
                            type
                        }
                    }
                }
            }`;

```

- **GET_TRANSACTION_ENUMS_QUERY**:
  - Query for retrieving transaction category and type enums.
  - Returns: Enum values for "ENUM_TRANSACTION_CATEGORY" and "ENUM_TRANSACTION_TYPE".

```
        export const GET_TRANSACTION_ENUMS_QUERY = gql`
            query GetTransactionEnums {
                categoryEnums: __type(name: "
    ENUM_TRANSACTION_CATEGORY") {
                    enumValues {
                        name
                    }
                }
                typeEnums: __type(name: "
    ENUM_TRANSACTION_TYPE") {
                    enumValues {
                        name
                    }
                }
            }`;

```

- **CREATE_TRANSACTION_MUTATION**:
  - Mutation for creating a new transaction.
  - Input: `TransactionInput` object.
  - Returns: ID and attributes (amount, category, name, wallet ID) of the created transaction.

```
    export const CREATE_TRANSACTION_MUTATION = gql`
        mutation CreateTransaction($data: TransactionInput!) {
            createTransaction(data: $data) {
                data {
                    attributes {
                        amount
                        category
                        name
                        wallet {
                            data {
                                id
                            }
                        }
                    }
                    id
                }
            }
        }`;

```

- **UPDATE_WALLET_AMOUNT_MUTATION**:
  - Mutation for updating the amount of a specific wallet.
  - Input: Wallet ID and new amount.
  - Returns: ID and attributes (amount, name, type) of the updated wallet.

```
    export const UPDATE_WALLET_AMOUNT_MUTATION = gql`
        mutation UpdateWallet($id: ID!, $amount: Float!) {
            updateWallet(id: $id, data: { amount: $amount }) {
                data {
                    id
                    attributes {
                        amount
                        name
                        type
                    }
                }
            }
        }`;

```

- **GET_WALLET_BALANCE_QUERY**:
  - Query for retrieving the balance of a specific wallet.
  - Input: Wallet ID.
  - Returns: Amount of the wallet.

```
        export const GET_WALLET_BALANCE_QUERY = gql`
            query GetWalletBalanceQUERY($walletId: ID) {
                wallet(id: $walletId) {
                    data {
                        attributes {
                            amount
                        }
                    }
                }
            }`;

```

- **GET_USER_WALLETS_QUERY**:
  - Query for retrieving wallets associated with a user.
  - Input: Optional filters (e.g., wallet type).
  - Returns: IDs, types, and names of user wallets.

```
    export const GET_USER_WALLETS_QUERY = gql`
        query GetUserWallets($filters: WalletFiltersInput) {
            wallets(filters: $filters) {
                data {
                    id
                    attributes {
                        type
                        name
                    }
                }
            }
        }`;
```

- **LOGIN_MUTATION**:
  - Mutation for user login.
  - Input: `UsersPermissionsLoginInput` object.
  - Returns: JWT, user ID, email, and username upon successful login.

```
export const LOGIN_MUTATION = gql`
    mutation Login($input: UsersPermissionsLoginInput!) {
        login(input: $input) {
            jwt
            user {
                id
                email
                username
            }
        }
    }`;
```

- **SIGNUP_MUTATION**:
  - Mutation for user registration.
  - Input: `UsersPermissionsRegisterInput` object.
  - Returns: JWT and user ID upon successful registration.

```
export const SIGNUP_MUTATION = gql`
    mutation Register($input:
   UsersPermissionsRegisterInput!) {
        register(input: $input) {
            jwt
            user {
                id
            }
        }
    }`;
```

GET_TRANSACTION_ENUMS_QUERY:

- Query for retrieving transaction category and type enums.
- Returns: Enum values for "ENUM_TRANSACTION_CATEGORY" and "ENUM_TRANSACTION_TYPE".

```
    export const GET_TRANSACTION_ENUMS_QUERY = gql`
        query GetTransactionEnums {
            categoryEnums: __type(name: "
   ENUM_TRANSACTION_CATEGORY") {
                enumValues {
                    name
                }
            }
            typeEnums: __type(name: "ENUM_TRANSACTION_TYPE") {
                enumValues {
                    name
                }
            }
        }`;

```

CREATE_TRANSACTION_MUTATION:

- Mutation for creating a new transaction.
- Input: TransactionInput object.
- Returns: ID and attributes (amount, category, name, wallet ID) of the created transaction.

```
    export const CREATE_TRANSACTION_MUTATION = gql`
        mutation CreateTransaction($data: \text{{
   TransactionInput}}!) {
            createTransaction(data: $data) {
                data {
                    attributes {
                        amount
                        category
                        name
                        wallet {
                            data {
                                id
                            }
                        }
                    }
                    id
                }
            }
        }`;

```

UPDATE_WALLET_AMOUNT_MUTATION:

- Mutation for updating the amount of a specific wallet.
- Input: Wallet ID and new amount.
- Returns: ID and attributes (amount, name, type) of the updated wallet.

```
    export const UPDATE_WALLET_AMOUNT_MUTATION = gql`
        mutation UpdateWallet($id: ID!, $amount: Float!) {
            updateWallet(id: $id, data: { amount: $amount }) {
                data {
                    id
                    attributes {
                        amount
                        name
                        type
                    }
                }
            }
        }`;

```

GET_WALLET_BALANCE_QUERY:

- Query for retrieving the balance of a specific wallet.
- Input: Wallet ID.
- Returns: Amount of the wallet.

```
    export const GET_WALLET_BALANCE_QUERY = gql`
        query GetWalletBalanceQUERY($walletId: ID) {
            wallet(id: $walletId) {
                data {
                    attributes {
                        amount
                    }
                }
            }
        }`;

```

GET_USER_WALLETS_QUERY:

- Query for retrieving wallets associated with a user.
- Input: Optional filters (e.g., wallet type).
- Returns: IDs, types, and names of user wallets.

```
export const GET_USER_WALLETS_QUERY = gql`
    query GetUserWallets($filters: WalletFiltersInput) {
        wallets(filters: $filters) {
            data {
                id
                attributes {
                    type
                    name
                }
            }
        }
    }`;
```

GET_TRANSACTIONS_QUERY:

- Query for retrieving transactions based on filters.
- Input: Transaction filters.
- Returns: IDs, attributes (createdAt, amount, category, name, wallet), of the transactions.

```
     export const GET_TRANSACTIONS_QUERY = gql`
         query GetTransactions($filters: TransactionFiltersInput) {
             transactions(filters: $filters) {
                 data {
                     id
                     attributes {
                         createdAt
                         amount
                         category
                         name
                         wallet {
                             data {
                                 attributes {
                                     type
                                 }
                             }
```

```
                            }
                        }
                    }
                }
            }`;

```

GET_WALLET_BALANCES:

- Query for retrieving the balances of multiple wallets.
- Input: Wallet IDs.
- Returns: Amounts of the wallets.

```
        export const GET_WALLET_BALANCES = gql`
            query GetWalletBalanceQUERY($walletId: ID) {
                wallet(id: $walletId) {
                    data {
                        attributes {
                            amount
                        }
                    }
                }
            }`;

```

GET_ALL_WALLETS_QUERY:

- Query for retrieving all wallets.
- Returns: IDs, amounts, and names of all wallets.

```
        export const GET_ALL_WALLETS_QUERY = gql`
            query GetAllWallets {
                wallets {
                    data {
                        id
                        attributes {
                            amount
                            name
                        }
                    }
                }
            }`;

```

GET_ALL_WALLETS_QUERY_NEW:

- Query for retrieving all wallets with additional details (transactions, type).
- Input: Optional filters (e.g., wallet type).

- Returns: IDs, attributes (amount, name, transactions, type) of all wallets.

```
export const GET_ALL_WALLETS_QUERY_NEW = gql`
    query Query_NEW($filters: WalletFiltersInput) {
        wallets(filters: $filters) {
            data {
                attributes {
                    amount
                    name
                    transactions {
                        data {
                            attributes {
                                name
                                updatedAt
                                amount
                                category
                            }
                        }
                    }
                    type
                }
            }
        }
    }`;

```

GET_ALL_WALLETS_QUERY_sta:

- Query for retrieving all wallets.
- Returns: IDs amounts of all wallets.

```
export const GET_ALL_WALLETS_QUERY_sta = gql`
    query GetAllWallets {
        wallets {
            data {
                id
                attributes {
                    amount
                }
            }
        }
    }`;

```

Chapter 5

Conclusion

5.1 Future direction

In the future, our money management application aims to incorporate AI capabilities to analyze users' buying patterns and provide personalized recommendations. By leveraging machine learning algorithms, we can gather insights from users' transaction data, identify their spending habits, and understand their preferences.

The AI system will continuously analyze patterns and trends in users' purchasing behavior, taking into account factors such as transaction frequency, purchase categories, spending amounts, and time of purchases. By applying advanced data analysis techniques, the AI will identify correlations and patterns that may not be immediately apparent to users.

Based on these insights, the AI system will generate personalized recommendations for users, suggesting relevant products or services that align with their interests and preferences. These recommendations may include discounts, special offers, or suggestions for similar products or services that users may find appealing.

To ensure privacy and data security, the AI system will adhere to strict data protection protocols. User data will be anonymized and processed in a secure and confidential manner, with the utmost respect for user privacy.

By integrating AI-driven recommendation systems into our money management application, we aim to enhance users' financial decision-making and provide them with valuable insights and suggestions to optimize their spending habits and achieve their financial goals.

5.2 Conclusion

In conclusion, the paper presents *Moneager*, a mobile application designed to provide users with a comprehensive and personalized solution for managing their financial transactions. *Moneager* offers a range of features and functionalities that enable users to track their expenses, create reports, and make informed financial decisions.

Throughout the development process, various technologies and tools were utilized to ensure the effectiveness and efficiency of the application. *React Native*, a TypeScript framework, served as the foundation for building native-like mobile applications for both iOS and Android platforms. The *Expo* development environment streamlined the frontend development workflow, providing features such as hot reloading and over-the-air updates.

The integration of *GraphQL* facilitated efficient data fetching and manipulation, reducing over-fetching and under-fetching of data. *TypeScript* enhanced the development process by introducing static typing, leading to improved code quality and better maintainability. Additionally, tools like *Figma* and *WebStorm* played a vital role in designing and developing the user interface, ensuring a seamless and intuitive user experience.

The implementation of *Moneager* demonstrated the significance of collaborative interface design and prototyping. *Figma*, a cloud-based design tool, enabled designers and developers to work together in real time, resulting in a cohesive and visually appealing user interface. The integration of *Apollo Studio* provided effective management, monitoring, and analysis of the GraphQL API, contributing to optimized performance and improved data delivery.

Overall, *Moneager* stands out as a user-friendly and comprehensive solution for managing personal finances. By incorporating educational resources and promoting financial literacy, *Moneager* aims to empower users to make informed financial decisions and achieve their financial goals. With its intuitive interface, collaborative design process, and robust backend architecture, *Moneager* offers a compelling user experience for individuals seeking an efficient and accessible tool for managing their finances.

In the future, further enhancements can be made to *Moneager*, such as expanding its feature set, integrating additional financial services, and incorporating advanced security measures. With continuous improvements and user feedback, *Moneager* has the potential to become a leading mobile application for financial management, helping users take control of their financial well-being and achieve long-term financial stability.

In summary, *Moneager* demonstrates the power of technology and innovative design in simplifying the complex task of managing personal finances. By leveraging the capabilities of *React Native*, *GraphQL*, and various supporting tools, *Moneager* provides users with a comprehensive and personalized solution for tracking expenses, making informed financial decisions, and ultimately achieving their financial goals.

Bibliography

[1] Brian Fling. *Mobile design and development: Practical concepts and techniques for creating mobile sites and Web apps.* O'Reilly Media, Inc., 2009.

[2] Daniel Schifano. Complete web & mobile designer resources. `https://danielschifano.notion.site/Complete-Web-Mobile-Designer-Resources-d1d9b6868a7746ffb3b6f02703ac7724`, 2023. Throughout the course, we are going to be using various tools and resources for designers to build our project.

[3] Facebook. *React Native.* O'Reilly Media, Inc., 2023.

[4] Expo Team. Expo. `https://expo.io/`, 2023. Expo is a set of tools and services built around React Native that simplifies the development process.

[5] Microsoft. Typescript. `https://www.typescriptlang.org/`, 2023. TypeScript is a statically typed superset of JavaScript that brings additional features and benefits to the development process.

[6] Facebook. Graphql. `https://graphql.org/`, 2023. GraphQL is a query language for APIs and a runtime for executing queries with existing data.

[7] Figma Team. Figma. `https://www.figma.com/`, 2023. Figma is a cloud-based design and prototyping tool for collaborative interface design.

[8] JetBrains. Webstorm. `https://www.jetbrains.com/webstorm/`, 2023. WebStorm is an IDE specifically designed for web development with advanced features for TypeScript, JavaScript, HTML, and CSS.

[9] Expo Team. Expo go. `https://expo.io/client`, 2023. Expo Go is a mobile app for testing and previewing React Native applications on physical devices.

[10] Facebook. React developer tools. `https://github.com/facebook/react-devtools`, 2023. React Developer Tools is a browser extension for enhancing the development experience with React applications.

[11] Apollo GraphQL. Apollo studio. `https://www.apollographql.com/studio/`, 2023. Apollo Studio is a tool for managing and analyzing GraphQL APIs.

[12] Strapi Team. Strapi. `https://strapi.io/`, 2023. Strapi is an open-source headless CMS for building API-driven applications.

[13] Apollo GraphQL. Apollo server. `https://www.apollographql.com/docs/apollo-server/`, 2023. Apollo Server is an open-source GraphQL server implementation for building scalable APIs.

[14] SQLite Consortium. Sqlite. `https://www.sqlite.org/`, 2023. SQLite is a lightweight and self-contained relational database management system.