

FundNest - A mobile application with open banking solutions to assist financial management

by Craig Redmond

Submission date: 08-May-2022

File name: Craig-Redmond-N00171313-Thesis.pdf

Word count: 24090

Character count: 145702

FundNest - A mobile application with open banking solutions to assist financial management

Author: Craig Redmond N00171313

Supervisor: Marian McDonnell

Second Reader: Anne Wright

Project Coordinator: Mohammed Cherbatji

Code: github.com/Craigr99/major-project-fundnest

Date: May 2022



Thesis submitted in partial fulfilment of the requirements for the BSc (Hons) in Creative Computing at the Institute of Art, Design and Technology (IADT).

Declaration of Authorship

I hereby certify that the material, which I now submit for assessment is entirely my own work and has not been taken from the work of others except to the extent of such work which has been cited and acknowledged within the text of my own work.

Declaration

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Signed: *Craig Redmond*

Date: 08/05/2022

Abstract

This document includes the research into the area of user experience within mobile banking applications. The research section of this document delves into the history of E-banking and techniques that can be used to enhance the user experience within these types of applications such as gamification and personalization. It then explores the topics of user experience design (UXD), user experience within mobile and banking applications, as well as the struggle that can be found with balancing security with a positive user experience. The goal of this project is to design and implement a mobile application to assist users with financial management. To conquer this task, a great deal of research and design went into the features that the app provides. The app allows users to link multiple accounts from one or more financial institutions to the application, analyse their income and expenditure from each account, set up and personalize custom saving goals and track bills and subscriptions that need to be paid. One key purpose of this application is to act as a tool for users to manage their finances, with an easy to use interface and a positive user experience to users which some Irish banking applications currently lack due to a lack of features and design flaws. The final application is built using the MERN stack, with the frontend (mobile app) developed using React Native, the database is hosted on MongoDB and the backend/server is hosted on Heroku. The Nordigen API is used to access account data and transaction categorization. Nordigen is a free open banking platform.

Acknowledgements

I would like to thank my project supervisor Marian McDonnell first of all for providing guidance and support continuously throughout the course of the project. The experience and knowledge that Marian has in this area aided a variety of my skills and had a positive impact on the quality of my final project. I would like to thank my second reader Anne Wright for taking her time to provide her invaluable feedback and observations on the project.

I would like to thank Mohammed Cherbati for his organisation of the Major Project. I would like to thank all of my lecturers who I have had the privilege to learn from over the years in IADT for providing me with an indispensable skillset.

I would lastly like to thank my family, who have provided support and guidance throughout each of the last four years, for keeping me motivated which allowed me to work hard and accomplish my goals that were set out throughout the years.

Table of Contents

| | | |
|----------|---|-----------|
| 1 | INTRODUCTION..... | 3 |
| 2 | RESEARCH..... | 5 |
| | EXAMINING HOW THE USER EXPERIENCE AND GAMIFICATION TECHNIQUES CAN ASSIST USERS WITH FINANCIAL MANAGEMENT WITHIN MOBILE BANKING APPLICATIONS..... | 5 |
| 2.1 | E-BANKING | 5 |
| 2.1.1 | HISTORY OF E-BANKING | 5 |
| 2.1.2 | GAMIFICATION OF E-BANKING | 5 |
| 2.2 | USER EXPERIENCE | 6 |
| 2.2.1 | ELEMENTS OF USER EXPERIENCE..... | 6 |
| 2.2.2 | BALANCE BETWEEN UX AND SECURITY | 6 |
| 2.2.3 | UXD FOR MOBILE | 7 |
| 2.3 | USER EXPERIENCE IN E-BANKING APPLICATIONS..... | 8 |
| 2.4 | FUTURE DIRECTIONS OF ONLINE BANKING | 8 |
| 2.5 | CONCLUSION..... | 9 |
| 3 | REQUIREMENTS ANALYSIS..... | 10 |
| 3.1 | INTRODUCTION | 10 |
| 3.2 | EXISTING APPLICATIONS | 10 |
| 3.2.1 | REVOLUT | 10 |
| 3.2.2 | MINT: | 12 |
| 3.2.3 | MONEYFY..... | 14 |
| 3.3 | SURVEYS..... | 16 |
| 3.4 | USER PERSONAS | 19 |
| 3.5 | USE CASE DIAGRAMS..... | 21 |
| 3.6 | REQUIREMENTS | 23 |
| 3.6.1 | USER REQUIREMENTS | 23 |
| 3.6.2 | TECHNICAL REQUIREMENTS | 24 |
| 3.6.3 | FUNCTIONAL REQUIREMENTS..... | 24 |
| 3.6.4 | NON-FUNCTIONAL REQUIREMENTS..... | 24 |
| 3.7 | FEASIBILITY STUDY..... | 25 |
| 3.8 | PROJECT PLAN | 26 |
| 4 | DESIGN | 28 |
| 4.1 | INTRODUCTION | 28 |
| 4.2 | SYSTEM ARCHITECTURE..... | 28 |
| 4.3 | APPLICATION DESIGN..... | 29 |
| 4.3.1 | TECHNOLOGIES | 29 |
| 4.3.2 | DESIGN PATTERNS..... | 30 |
| 4.3.3 | PROCESS DESIGN | 32 |

| | | |
|-------|----------------------------|----|
| 4.4 | USER INTERFACE DESIGN..... | 36 |
| 4.4.1 | WIREFRAME..... | 36 |
| 4.4.2 | STYLE GUIDE | 41 |
| 4.5 | CONCLUSION..... | 41 |

5 IMPLEMENTATION 43

| | | |
|--------|---|----|
| 5.1 | INTRODUCTION..... | 43 |
| 5.2 | DEVELOPMENT ENVIRONMENT..... | 43 |
| 5.3 | DATABASE | 44 |
| 5.4 | BACKEND..... | 45 |
| 5.4.1 | CONFIGURATION – ENVIRONMENT VARIABLES | 45 |
| 5.4.2 | APPLICATION CONFIGURATION & SCRIPTS..... | 45 |
| 5.4.3 | APP/SERVER INITIALIZATION | 45 |
| 5.4.4 | INDEX FILE – APP ENTRY POINT..... | 46 |
| 5.4.5 | APP STRUCTURE..... | 47 |
| 5.4.6 | LINKING AN ACCOUNT..... | 49 |
| 5.4.7 | RETRIEVING ACCOUNTS..... | 53 |
| 5.5 | FRONTEND..... | 55 |
| 5.5.1 | PROJECT DEPENDENCIES..... | 56 |
| 5.5.2 | PARENT COMPONENT | 57 |
| 5.5.3 | AUTHENTICATION SCREENS..... | 59 |
| 5.5.4 | LINKING BANK ACCOUNTS | 63 |
| 5.5.5 | HOME SCREEN – USER INDEX..... | 73 |
| 5.5.6 | SAVINGS SCREEN – INDEX..... | 75 |
| 5.5.7 | SAVINGS SCREEN – SHOW | 78 |
| 5.5.8 | TRANSACTIONS SCREEN – INDEX..... | 79 |
| 5.5.9 | SWITCHING ACCOUNTS | 83 |
| 5.5.10 | ADD ITEM SCREEN..... | 85 |

6 TESTING..... 86

| | | |
|-------|------------------------------------|----|
| 6.1 | INTRODUCTION..... | 86 |
| 6.2 | FUNCTIONAL TESTING..... | 86 |
| 6.2.1 | AUTHENTICATION..... | 87 |
| 6.2.2 | NAVIGATION | 88 |
| 6.2.3 | CALCULATION..... | 88 |
| 6.2.4 | ANALYSIS OF FUNCTIONAL TESTS | 89 |
| 6.3 | USER TESTING | 90 |
| 6.3.1 | TEST PARTICIPANTS | 90 |
| 6.3.2 | TEST ENVIRONMENT | 91 |
| 6.3.3 | TEST METHODS | 91 |
| 6.3.4 | USER TESTING TASKS & RESULTS | 91 |
| 6.4 | CONCLUSION..... | 93 |

7 PROJECT MANAGEMENT 94

| | | |
|-----|---------------|----|
| 7.1 | GITHUB..... | 95 |
| 7.2 | JOURNAL | 97 |

| | | |
|-----|-----------------|----|
| 7.3 | CONCLUSION..... | 97 |
|-----|-----------------|----|

8 CONCLUSION 98

| | | |
|-------|---|-----|
| 8.1 | PROJECT SUMMARY..... | 98 |
| 8.2 | FUTURE WORK | 98 |
| 8.2.1 | SAVINGS FEATURE INTEGRATED INTO REAL BANK ACCOUNTS..... | 99 |
| 8.2.2 | CURRENT FEATURES ENHANCED | 99 |
| 8.2.3 | MODIFICATIONS FROM USER TESTING RESULTS..... | 99 |
| 8.2.4 | FURTHER USER TESTING | 99 |
| 8.2.5 | PROVIDING SUPPORT FOR MULTIPLE COUNTRIES | 99 |
| 8.2.6 | PROVIDING THE APP FOR DIFFERENT MOBILE DEVICES | 100 |
| 8.3 | LIMITATIONS | 100 |
| 8.4 | LEARNING OUTCOMES | 100 |
| 8.5 | FINAL WORDS | 101 |

9 REFERENCES..... 102

10 APPENDICES..... 104

| | | |
|--------|------------------------------------|-----|
| 10.1 | APPENDIX A (SURVEY) | 104 |
| 10.2 | APPENDIX B (PAPER PROTOTYPES)..... | 106 |
| 10.3 | APPENDIX C (SURVEY DOCUMENTS)..... | 109 |
| 10.3.1 | TEST INTRODUCTION | 109 |
| 10.3.2 | CONSENT FORM..... | 110 |
| 10.3.3 | TEST TASKS | 111 |
| 10.3.4 | POST-TEST QUESTIONNAIRE | 112 |
| 10.3.5 | TESTING NOTES | 113 |

Table of Figures

| | |
|--|-------------------------------------|
| FIGURE 1 REVOLUT SCREENS DESIGN | 11 |
| FIGURE 2 MINT SCREENS DESIGN | 13 |
| FIGURE 3 MONEFY SCREENS DESIGN | 15 |
| FIGURE 4 SURVEY AGE CATEGORIES | 16 |
| FIGURE 5 SURVEY GENDERS | 16 |
| FIGURE 6 SURVEY USERS FINANCIAL DIFFICULTIES | ERROR! BOOKMARK NOT DEFINED. |
| FIGURE 7 SURVEY LEARNING TYPES..... | 17 |
| FIGURE 8 SURVEY PREFERRED BANKING APPS | 17 |
| FIGURE 9 SURVEY BUDGETING TOOLS | 18 |
| FIGURE 10 SURVEY CONFIDENCE LEVELS..... | 19 |
| FIGURE 11 USER PERSONA..... | 20 |
| FIGURE 12 USE CASE DIAGRAM 1 | 22 |
| FIGURE 13 USE CASE DIAGRAM 2 | 23 |
| FIGURE 14 FLUTTER..... | 25 |
| FIGURE 15 REACT NATIVE | 26 |
| FIGURE 16 MERN STACK..... | 26 |
| FIGURE 17 PROJECT PLAN SCHEDULE | 27 |
| FIGURE 18 MICROSOFT TASKS PLANNER | 27 |
| FIGURE 19 SYSTEM ARCHITECTURE DESIGN | 29 |
| FIGURE 20 SYSTEM DESIGN PATTERN | 30 |
| FIGURE 21 REDUX ARCHITECTURE - HTTPS://WWW.JAVATPOINT.COM/REACT-REDUX | 32 |
| FIGURE 22 NORDIGEN CUSTOMER JOURNEY | 33 |
| FIGURE 23 CLASS DIAGRAM | 34 |
| FIGURE 24 SEQUENCE DIAGRAM | 35 |
| FIGURE 25 FLOW CHART - LOGGING INTO ACCOUNT..... | 35 |
| FIGURE 26 FLOW CHART - ADDING A NEW SAVING GOAL..... | 36 |
| FIGURE 27 AUTHENTICATION WIREFRAMES..... | 36 |
| FIGURE 28 FIRST ITERATION OF MAIN SCREENS | 37 |
| FIGURE 29 SECOND ITERATION OF SCREENS | 38 |
| FIGURE 30 DIFFERENT DESIGN OF HOME SCREEN | 39 |
| FIGURE 31 ADD & EDIT SAVING GOAL SCREENS | 40 |
| FIGURE 32 STYLE GUIDE | 41 |
| FIGURE 33 DEVELOPMENT ENVIRONMENT WORKFLOW | 44 |
| FIGURE 34 MONGODB CLUSTER | 44 |
| FIGURE 35 ENVIRONMENT VARIABLES | 45 |
| FIGURE 36 PACKAGE.JSON FILE | 45 |
| FIGURE 37 SERVER JS FILE..... | 46 |
| FIGURE 38 INDEX.JS CONFIGURATION | 46 |
| FIGURE 39 INDEX FILE - RUNNING SERVER..... | 47 |
| FIGURE 40 USERS ROUTER | 47 |
| FIGURE 41 USER CONTROLLER - LOGIN FUNCTION | 48 |
| FIGURE 42 USERS DAO – GETUSER()..... | 49 |
| FIGURE 43 TESTING ENDPOINTS IN INSOMNIA | 49 |
| FIGURE 44 ACCOUNTS ROUTER | 49 |
| FIGURE 45 ACCOUNTS CONTROLLER - ADD FUNCTION | 50 |
| FIGURE 46 RETRIEVING ACCOUNTS | 50 |
| FIGURE 47 GETTING ACCOUNT DETAILS..... | 51 |
| FIGURE 48 ACCOUNTS DAO - ADD ACCOUNT | 52 |
| FIGURE 49 RETURNING RESPONSE WITH ACCOUNTS | 52 |
| FIGURE 50 SUCCESSFUL REQUEST | 53 |
| FIGURE 51 ACCOUNTS CONTROLLER - GET ACCOUNTS..... | 53 |
| FIGURE 52 ACCOUNTS DAO – GETACCOUNTSBYEMAIL..... | 54 |
| FIGURE 53 SUCCESSFUL GET ACCOUNTS RESPONSE..... | 55 |

| | |
|---|-------------------------------------|
| FIGURE 54 NATIVEBASE THEMES - HTTPS://WWW.NPMJS.COM/PACKAGE/NATIVE-BASE | 56 |
| FIGURE 55 FRONTEND DEPENDENCIES | 57 |
| FIGURE 56 PARENT - APP COMPONENT | 57 |
| FIGURE 57 CUSTOM THEME CONTAINER | 58 |
| FIGURE 58 SCREENS STACK | 59 |
| FIGURE 59 DEVELOPMENT OF FIRST SCREENS | 60 |
| FIGURE 60 KEYBOARD AWARE SCROLL VIEW | 61 |
| FIGURE 61 KEYBOARD BLOCKING PASSWORD INPUT | 61 |
| FIGURE 62 REGISTER COMPONENT | 61 |
| FIGURE 63 REGISTER FUNCTION | 62 |
| FIGURE 64 BANKS LIST COMPONENT | 63 |
| FIGURE 65 BANKS LIST JSX | 64 |
| FIGURE 66 LIST OF BANKS | 64 |
| FIGURE 67 SELECTBANK FUNCTION | 65 |
| FIGURE 68 USER AGREEMENT SCREEN | 65 |
| FIGURE 69 CREATEAGREEMENT FUNCTION | 66 |
| FIGURE 70 CONDITIONALLY RENDERING | 67 |
| FIGURE 71 ONREFRESH FUNCTION | 67 |
| FIGURE 72 LIST ACCOUNTS FUNCTIONS | 68 |
| FIGURE 73 GETACCOUNTS FUNCTION | 69 |
| FIGURE 74 CONDITIONALLY RENDER ACCOUNT | 69 |
| FIGURE 75 SELECT ACCOUNT SCREEN | 70 |
| FIGURE 76 SELECTACCOUNT FUNCTION | 71 |
| FIGURE 77 ACCOUNT ADD SUCCESS SCREEN | 72 |
| FIGURE 78 HOME SCREEN | 73 |
| FIGURE 79 USER INDEX COMPONENT | 74 |
| FIGURE 80 LOGOUT FUNCTION | 74 |
| FIGURE 81 SAVINGS INDEX SCREEN | 75 |
| FIGURE 82 FUNCTIONS TO CALCULATE TOTALS | 75 |
| FIGURE 83 CURRENCY FORMATTER FUNCTION | 76 |
| FIGURE 84 PERCENTAGE FUNCTION | 76 |
| FIGURE 85 SAVINGS INDEX - GETSAVINGS() | 77 |
| FIGURE 86 SAVINGS SHOW SCREEN | 78 |
| FIGURE 87 TRANSACTIONS INDEX SCREEN | 79 |
| FIGURE 88 TRANSACTIONS INDEX - USEEFFECT HOOK & GETTRANSACTIONS | 80 |
| FIGURE 89 GETACCOUNTBALANCE FUNCTION | 80 |
| FIGURE 90 TRANSACTIONS LOOP | 81 |
| FIGURE 91 TRANSACTION ICON COMPONENT | 82 |
| FIGURE 92 TRANSACTIONS MARKUP | 82 |
| FIGURE 93 ACCOUNT SELECTION | 83 |
| FIGURE 94 GETACCOUNTS FUNCTION | 84 |
| FIGURE 95 ACCOUNT LOOP IN ACTION SHEET | 84 |
| FIGURE 96 SELECT ACCOUNT FUNCTION | 85 |
| FIGURE 97 ADD ITEM SCREEN | 85 |
| FIGURE 98 MICROSOFT PLANNER | 95 |
| FIGURE 99 GITHUB BRANCHES | 95 |
| FIGURE 100 GITHUB REPOSITORY | 96 |
| FIGURE 101 COMMITS MADE TO THE REPOSITORY OVER 3 MONTHS | 96 |
| FIGURE 102 BANK LIST FOR IRELAND | ERROR! BOOKMARK NOT DEFINED. |

1 Introduction

This project focuses on the elements of User Experience (UX) and User Experience Design (UXD) in E-Banking applications. The development of UX design for mobile apps has evolved enormously in recent years and has made these apps accessible and easy to use for most users. However, some users still struggle with mobile banking apps. The research section of this document initially delves into and then explores a brief history of e-banking, and the gamification of e-banking. A section of the literature review discusses the techniques of gamification that are used in the internet banking industry and how they can have an impact of the user experience. The research then focuses mainly on specific topics of user experience – elements of UX, how the balance between a positive user experience and security elements can be challenging, and some general practices for designing good user experiences for applications on mobile devices.

There are as many as 1.9 billion individuals who use e-banking applications as of 2020 (Statista Research Department, 2021), therefore it is important to understand how to design a good user experience for banking application as it can increase customer loyalty and the demand for mobile banking applications will increase over the next few years as it looks to be the future of finance.

FundNest is a mobile application that aims to assist users with financial management. This application provides users with various features with a goal to provide each user with a positive user experience. FundNest is a platform where users can register an account and link multiple accounts from existing financial institutions to the application with a simple authentication flow. From here, users can display, manage and analyse their banking information in a minimalistic designed user interface, to make the financial figures easy for all types of users to understand.

One of the primary goals of the application is to make it easy for users to manage their finances and savings. For this reason a lot of thought and research went into the design and development of the apps main features. A valuable feature is provided to allow users to set up saving goals, where the user defines the saving goal name and amount. For example if a user would like to save up for a new car they can define a goal for this which includes the goal amount. This feature implements personalization aspects allowing the user to choose an icon and colour for the goal. The concept for this feature was so that the user could transfer funds to and from these saving goals from their main bank accounts balance until they have reached their goal. The user could then be rewarded with feedback from the application which implements a gamification feature, which would make saving more enjoyable.

FundNest has different sections within the application, with each section providing simple design and functionality, to make it easy for each user to use and understand. For this reason, a lot of research and design iterations went into the design phase of the application.

The application was developed using various tools and frameworks. The mobile app (frontend) was developed using the React Native framework, which is a framework provided by React, that allows developers to create native mobile applications – for both Android and

iOS. React Native is written in JavaScript and was created and is maintained by the company Meta, formerly Facebook. The backend of the application, various technologies were adopted such as NodeJS and Express for the server and MongoDB for the database provider. These technologies combined make up the MERN stack (MongoDB, Express, React, Node).

For the project management aspect of the project, agile and SCRUM methodologies were used. This helps provide a simple schedule throughout the project development timeline, which makes it easy to organise time efficiently and develop a structured application. The SCRUM methodology is where the project is split up into different sprints, each one lasting one to two weeks. Within each sprint, work is performed on different items from the product backlog. Sprints are continued until the project deadline has passed. This project was developed in 8 sprints overall, over a period of 4 months, which can be seen in more detail in the project management section of this document. The SCRUM methodology is used by software companies to take an organized approach in project management.

This document is structured as follows – Chapter 2 discusses the research that went into the areas of mobile banking and user experience design in mobile and E-banking applications. This was a literature review on how the user experience and gamification techniques can assist users with financial management. Chapter 3 discusses the requirements analysis of the application, where research into similar applications was performed and the requirements of the application was defined. Chapter 4 describes the design of the application including the system architecture and user interface design. Chapter 5 is the implementation chapter, which discusses how the outlined designs were implemented. Chapter 6 is the testing chapter, which discusses user and functional testing and the analysis on each. Chapter 7 discusses how the project was managed overall using SCRUM methodologies. Chapter 8 discussed the limitations encountered throughout the development of the app and the future development ideas that could be undertaken. Chapter 9 is the conclusion chapter of the project, where the project is summarised and conclusions about the concept of the projects idea is discussed.

The following chapter is a literature review that was completed which includes research of online literature that was available in the areas of mobile banking and User Experience Design.

2 Research

Examining how the user experience and gamification techniques can assist users with financial management within mobile banking applications

2.1 E-Banking

2.1.1 History of E-Banking

E-Banking is the term that describes customers using the internet to access services for their bank accounts and the banking transactions that take place online. It can involve the facilities such as account access, fund transfers, and the ability to buy financial services or products online. Financial institutions began implementing e-banking services in the mid 1990's, however many consumers were hesitant to conduct transactions over the internet. It took a few years for these services to gain adoption and by 2000, 80 percent of U.S banks offered e-banking services to customers (Keivani, Jouzbarkand, Khodadadi, & Sourkouhi, 2012). It took well-renowned companies such as Amazon, eBay and America Online to make the idea of paying for items online widespread. In 2001, Bank of America became the first bank to reach 3 million online banking customers, which was more than 20 percent of its customer base. Their customers made a record 3.1 million electronic bill payments by the end of 2001, which totaled more than \$1 billion. In 2009, a Gartner Group report estimated that 47 percent of the U.S population of adults and 30 percent of the United Kingdom banked online (Batchelor, 2017). The rise and development from the past to present has been rapid, and as of 2020, as many as 1.9 billion users worldwide actively used online banking services, with this number forecast to reach 2.5 billion by 2024 (Statista Research Department, 2021).

In 2017, 58% of the Irish population used mobile banking and this figure was expected to reach over 70% by 2024. These numbers have drastically increased since then due to some factors. Like numerous other aspects of the world, Covid-19 has prompted a change in peoples banking habits. For much of the year in 2019 to 2020 people did not have access to bank branches. Everyone had to use online banking for their banking activities. In 2021, 78% of users do their banking online and 60% do their banking on mobile devices (Hennessy, 2021). Due to the increase of users adopting online banking, banks across Ireland have begun downsizing - Bank of Ireland are set to close more than 100 branches across the country and Ulster Bank have decided to shut down indefinitely. This has resulted in banks being required to offer better digital services for their users everyday banking needs.

2.1.2 Gamification of E-Banking

Gamification has gained popularity in certain industries such as in education and healthcare and has started being brought into the area of banking in recent years. Gamification is the term used to describe the game principles and mechanics being applied to something to motivate and encourage their users to perform specified activities or change the behavior of a group. These methods appeal to natural human desires and impulses such as the need for entertainment, fun, reward, and competition (Babrovich, 2017). The adoption of gamification

in e-banking has the potential to change the customers attitude towards money and finance by making financial management fun, and to encourage better behaviors and attitudes towards things such as increasing savings and investments. There are numerous methods that developers can follow to implement gamification in banking, some examples include the BBVA banks game that has a goal to promote the use of online banking, by allowing their users to win points by using their services that they can exchange for gifts and entries to prize draws (Rodrigues, Oliveira, & Costa, 2016).

These elements of gamification can influence the enjoyment and ease-of-use on the application and will result in a positive user experience overall. The social cues and ease of use which online games comprise sometimes result in the users the users playing the same game repeatedly, almost to the point of addiction, and it has been found that these cues could lead users to increase their commitment to online business applications, such as an e-banking app. Some attention should be paid to some factors in the worldwide financial situations - some users may have experienced financial losses due to some financial factors, such as the global financial crisis between 2007 and 2009, where many banks around the world incurred large losses and relied on government support to avoid bankruptcy. These factors may have led to a decrease in their confidence in banking in general. Variables that must be paid attention to are usefulness, socialness, ease of use and enjoyment, as getting these factors correct, will inevitably make the bank user confident using the system. There is a lot of potential in gamification in banking, such as the potential to simplify the perception of complex banking products and services, gather data insights about customers and the possibility to increase financial literacy among customers and their kids (Babrovich, 2017).

2.2 User Experience

2.2.1 Elements of user experience

User Experience (UX) consists of all aspects of interaction between a user and a product. User Experience Design (UXD) is the process that design teams use to create products such as a mobile app that will provide meaningful experiences for the user. (The Interaction Design Foundation, 2021). There are numerous design principles that designers follow to create products with a good user experience. There are multiple factors that contribute to the user experience, for example if a person is reading a book or an article, the experience can be ruined by reading a tiny font that can be uncomfortable on your eyes, or if a persons is using an app, the app forgetting where they last stopped reading can be frustrating to users (Perea & Giner, 2017). The main topic related to UX is User Interface (UI). The UI refers to a system and a user interacting with each other through commands or techniques to perform operations, use contents and input data. The user interfaces can range from desktop computers to mobile devices and games (Joo, 2017). The focus of the research is on user interface and user experience design for mobile applications, as many of the modern e-banking applications are now on mobile devices.

2.2.2 Balance between UX and security

When designing an e-banking application, maintaining a balance between security and a positive user experience can prove a challenge. It is apparent that the various forms of security can hinder a user's experience, such as the authentication methods when logging into a banking app taking time for the user. However according to the work of Svilar &

Zupančič (2016), many users suggested that security and usability are the most important features in banking apps, and the less important features that followed were ease of use, responsiveness, reliability, and accessibility (Svilar & Zupančič, 2016). This implies that the security of online banking apps is more important to the user than the design or interface. Users are more cautious when using banking applications, so they want to be sure that their finances are safe and secure. The users agree that additional passwords are necessary and that banking applications are useful products and are easy to use and learn. Reducing the numbers of steps involved in the authentication process is important in maintaining a good user experience. According to Krol, Philipou, De Cristofaro and Sasse (2015), an ideal process when authenticating a user would require fewer steps and not demand the use of additional devices for one-time passwords.

In the last few years, the use of biometric data has been used to authenticate users on devices. Biometric data is anything that relates to the measurement of people's features and characteristics. This data is used to prove a person's uniqueness and verify if a person is who they are (GoodID team, 2021). For example, fingerprints and facial recognition is now being used in modern smartphones to unlock them. Banking institutions have begun integrating these technologies into their applications. For example, Revolut uses the user's biometric data such as Face ID or fingerprints to login to the app. Many FinTech analysts predict that the PIN number method of authentication may become obsolete within the next few years with the advancement of this technology (Clearbridge Mobile, 2018). Using passwords and PIN numbers as the main security method makes them an easy target for fraud. With the advent of multi-factor authentication, which combines a user's fingerprint with a variety of factors such as voice or facial recognition, it's becoming more prevalent to use this method in the banking industry. With the advancement in these new technologies, the UX and security is enhanced for users.

2.2.3 UXD for Mobile

Mobile UX design is the design of user experiences for handheld and wearable devices. Designers create solutions to meet mobile users' restrictions and requirements. The focus is on accessibility, ease of use, and efficiency to optimize on the go interactive experiences (The Interaction Design Foundation, 2014). The success of an application is related to the degree of the user's acceptance. User Experience Design (UXD) plays an important role the development of mobile applications because it can have a huge influence on a product's success or failure. There are many factors to consider when developing a mobile application. In the study carried out by Yazid and Jantan (Azwa & Azrul, 2017), they presented a UXD strategy for creating a mobile flight booking application under different headings such as ease of use, learnability, user interface, security, user satisfaction and behavioral intent. These headings can be used as a UX strategy for designing any type of mobile application because each of them can contribute to a good user experience in a mobile application.

Ease of use elements could be divided into different sub elements that cover navigation, usability, and data accessibility. Usability is how the application can serve the user to achieve the specified goals in the specified context of use. The actions to navigate through an app should be memorable to the user for when the return to an app, the step count should be minimized to simplify a process. The design of the navigation should be intuitive, predictable, and consistent. When designing a user interface for mobile devices there are some challenges

that are posed. Compared to a computer screen, when designing on smaller mobile screens, a higher percentage of the screen must be devoted to text in order to make it easily legible, which can sometimes leave less room for illustrations or visual enhancements on screen design. It is important to think about these factors when designing for mobile screen sizes. Developers must think about content prioritization – the goal is to satisfy most of the user’s needs, but not include any unnecessary that may negatively impact the mobile UX, especially elements that will increase page load time due to a slow network (Enginess, 2016).

2.3 User Experience in e-banking applications

Internet banking has changed the way consumers carry out their banking activities. Most of these activities are now done through mobile devices. This was not always the case however, as when mobile banking apps started being used back in 2009, most of these applications were not native mobile applications, but were instead web apps that allowed the user to access the internet banking features on a mobile device (Meadows, 2019). More customers are beginning to use online banking services as technology develops, but there are still some barriers to the usage of e-banking including the perception of risks and the understanding of the benefits of e-banking. It has been found that a good way to make customers engaged with the applications is through user experience elements such as personalized services as the banking preferences of customers keep changing (Wang, Cho, & Denton, 2017).

Personalization increases digital engagement of users. People are inclined to use an app more if they can customize it the way they want it. This is more about the functionality and features that the user gets out of the app and the experience rather than changing colors or background photos. An example would be what screen should the user see when they login, what order the accounts are in etc. These contribute to a good user experience in e-banking (Meadows, 2019). It is important to have an application that the user perceives as easy to use, as if the user finds the service difficult to use, they will find a different way to make transactions. The more difficult an app is to use, the more unlikely it is that a user will want to use it (Rahi, Ghandi, & Alnaser, 2017). The users that are more inexperienced with e-banking in general have indicated that personalization allows them to find more utility in their experience in e-banking applications according to the results of Wang, Cho, and Denton’s (2017) research.

According to a study by Amin et al. it is clear that perceived usefulness of an application is a big predictor factor for trust in e-banking. This means that the usability of the technology is important in gaining the users satisfaction and loyalty (Amin, Rezaei, & Abolghasemi, 2014).

2.4 Future Directions of Online Banking

Due to the constantly evolving technologies, there will be more changes in the future to internet banking industry than ever before. The user experience in banking will be significantly affected by the development of Artificial Intelligence. This will be noticeable through the delivery of mass personalization and assisting customers to overcome lower levels of financial literacy (KPMG, 2019). KPMG are suggesting that by 2030, algorithms and data models will be built around optimizing financial outcomes for customers and will reinforce positive behaviors through “nudging” people to do certain things. An example of this would be preventing customers from making poor financial decisions. It is also predicted

that by 2030 everything will be connected. Carrying a plastic card to tap at dedicated points of sale or making mobile payments might be replaced by a secure voice command or facial expression. Another future direction in this area will be the simplification of banking. Customers now expect straightforward, easy to use digital experiences from their service providers. Users can buy from Amazon with a click, get instant access to music on Spotify etc. and they will expect similar experiences from their financial providers. To remain relevant to the competition, banks will move in the direction of offering simple, easy to use, customer-centric digital banking apps. We are likely to see the user experience become simplified, which will be designed around real-life customer journeys to get rid of the friction from the user experience and enable customers to move across digital and physical banking channels seamlessly (Finextra Editorial Team, 2019).

2.5 Conclusion

Many of the research studies carried out in the area of e-banking and user experience indicate that there is a lot of potential for future directions of internet banking. The modernization and principles in current user experience design patterns will be important to users in their mobile banking apps. It is important that the users can have a positive and intuitive experience on the apps and that they can perform some sort of personalization. Nonetheless, it is important that these banking apps will have good security features so that the user can feel safe and secure while using the app, so it may always be a challenge to have a positive fluid user experience in applications involving finance, but research suggests that having more security elements will always be more important than the user experience, so it is a compromise that is worth it.

3 Requirements Analysis

3.1 Introduction

Requirement gathering is performed to determine what the project needs to achieve and what needs to be created to make that happen. This is achieved by the process of generating a list of different requirements such as functional, technical, non-functional and user requirements.

3.2 Existing Applications

Numerous mobile banking and budgeting apps were analyzed to see what features they offer and the pros and cons of each app. The following apps were analyzed:

1. Revolut
2. Mint
3. Monefy

Each app's features were analyzed to see if they provided a good user experience.

3.2.1 Revolut

Features:

- Spending categorized by the type of transaction (shopping, restaurants etc.)
- Set monthly budgets and spending goals, for overall and different categories
- Link multiple accounts

Pros:

- User interface
- Expanding range of features
- Premium subscriptions with handy perks
- Can top up and hold several currencies
- Free

Cons:

- Poor customer support
- Customers' accounts can be frozen temporarily due to security reasons
- Certain services can only be accessed with premium subscription memberships

Screenshots:

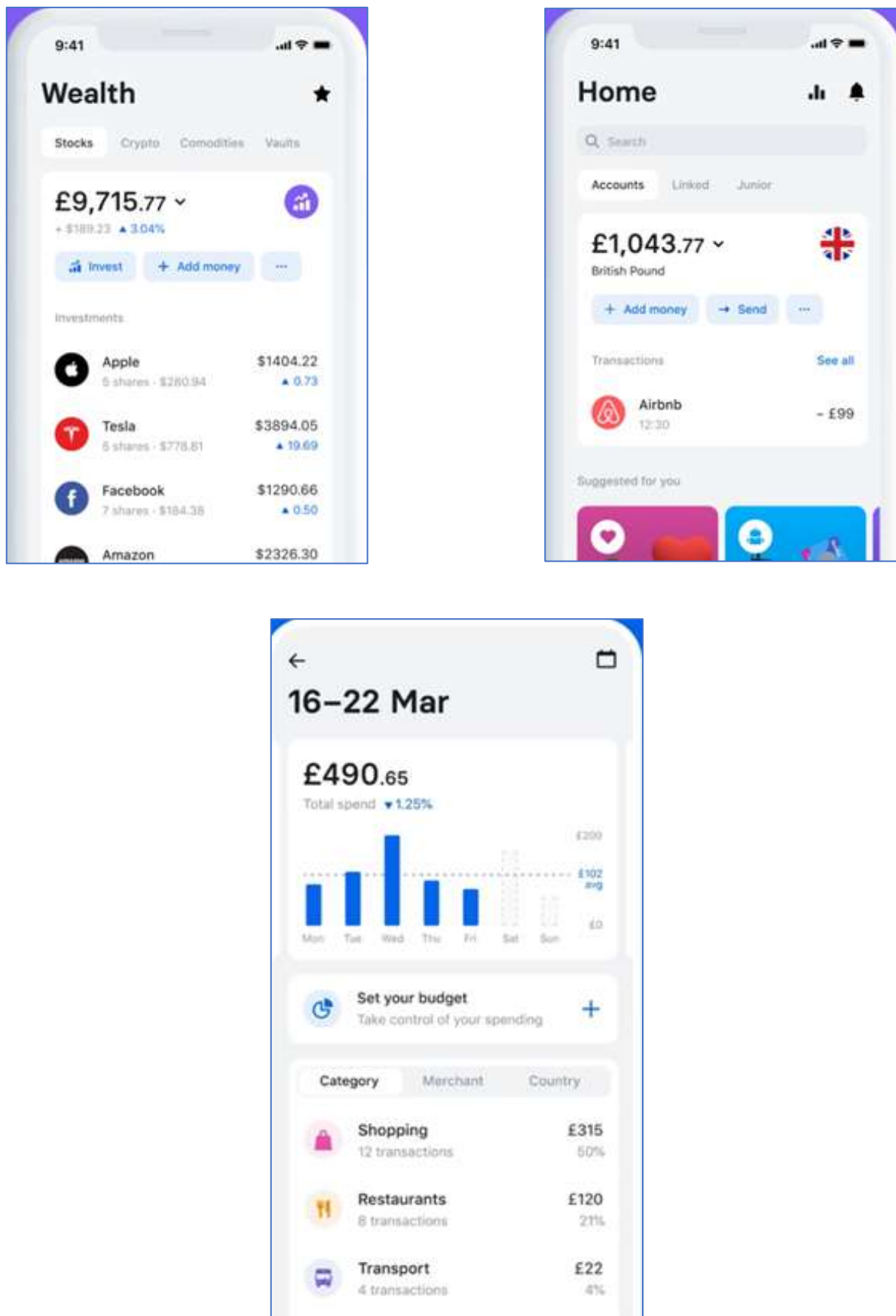


Figure 1 Revolut Screens Design

3.2.2 Mint:

Features:

- Connect multiple accounts – from cash to credit card, loans and investments
- Simple UI so you can see your complete financial picture clearly
- Track cash flow – adding bills and subscriptions, users get notified when subscription costs increase and when bills are due
- Save smarter with custom budgets – categorized transactions

Pros:

- Free
- Ease of use
- Link multiple accounts
- Credit score – users are notified of any changes in their report
- Trusted by millions of users
- Security – 4-digit code, touch ID. Users can delete account information remotely

Cons:

- In app advertisements
- Doesn't support multiple currencies
- Can't assign multiple savings goals to one account
- Only available in US and Canada

Screenshots:

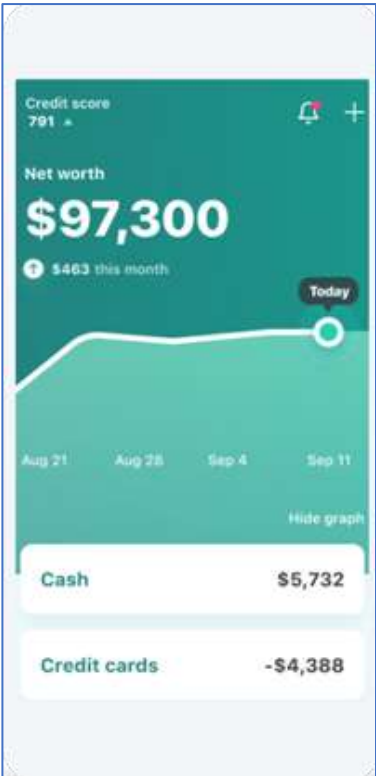
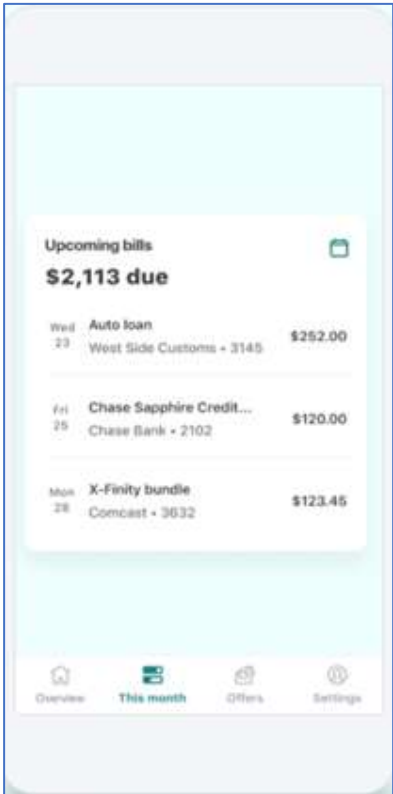
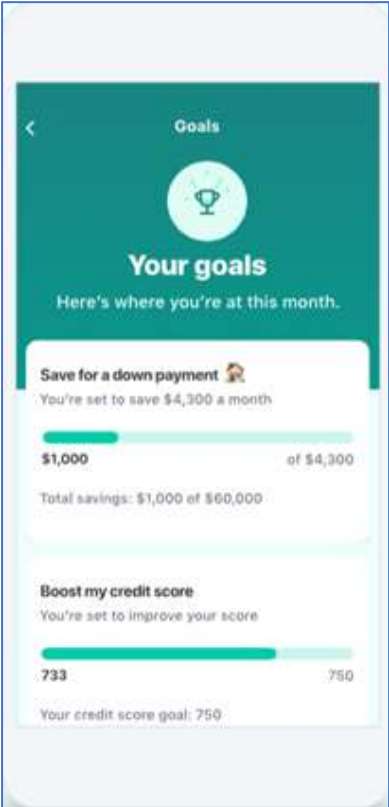


Figure 2 Mint Screens Design

3.2.3 Monefy

Features:

- Breaks down expenses in a simple, intuitive, and understandable way
- Lightning-fast tracking – add or remove expenses within a few seconds by tapping “+” or “-” button, enter amount and select category. Or tap the category icon on the chart and hit add
- Fully customizable – comes with many default categories, however a user can add custom categories

Pros:

- Free
- Nice user interface: Displayed in a chart
- Easy to use – quickly add and remove expenses with clear call-to-action buttons
- Detailed overview of categorized expenses

Cons:

- Each expense needs to be entered manually
- Possibly limited functionality

Screenshots:



Figure 3 Moneyfy Screens Design



3.3 Surveys

A survey was completed by 68 users that related to online banking in order to get a better idea of the users' requirements for mobile banking applications. This survey could then be used to create user personas and user journeys, and to see what functionality was most important for this type of application.



Figure 4 Survey age categories

From the survey results, the respondents belonged to varying age groups, most of them being younger users from the age 16-24 and 25-44 (Figure 4).



Figure 5 Survey genders

Most users participating in this survey were female as shown in Figure 5 above. As shown in these pie charts in **Error! Reference source not found.**, it is noticeable that users sometimes find it difficult to manage their finances and that they find banking applications difficult to use at times. There could be some correlation here between the two. As shown in Figure 6 below, most users tend to be visual learners and therefore having a well-designed application with a pleasing user experience should assist users in learning how to use the application quickly and finding it easier to complete their tasks.

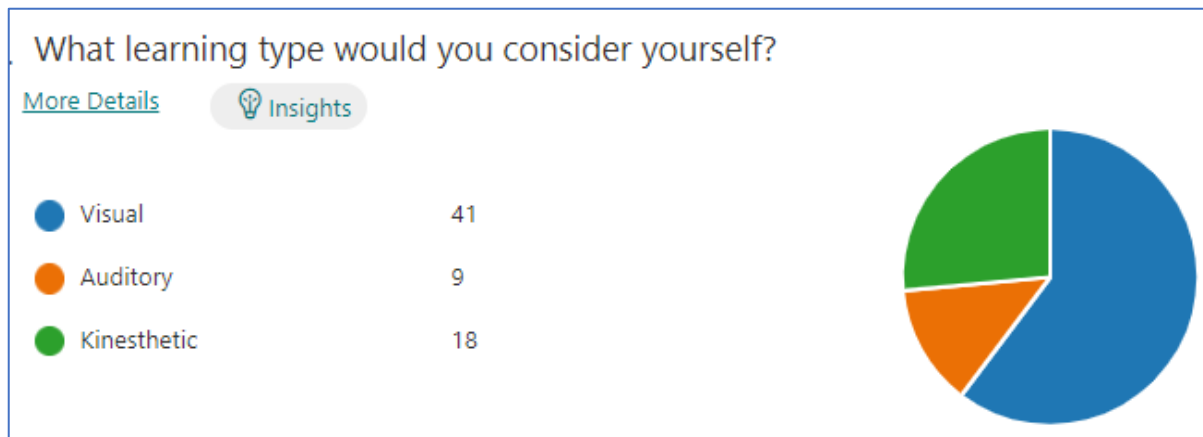


Figure 6 Survey learning types

When users were asked what their preferred banking app to use is, most people answered 'Revolut' and 'AIB' in Figure 7. This is most likely because both apps have a well-designed user interface and make it easy for users to use and grasp the information on the screen.

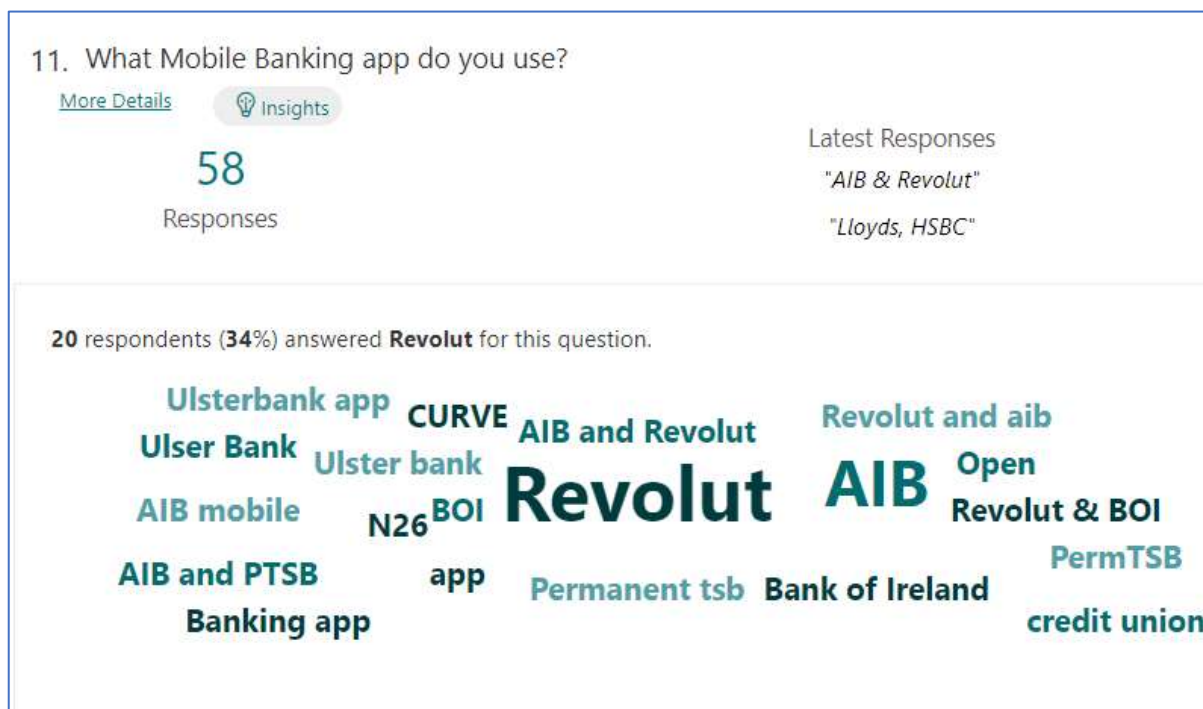


Figure 7 Survey preferred banking apps

While trying to specify the requirements for the application, the users were asked if they use any form of digital budgeting tool for financial planning. Most users answered 'No' to this question and the users that do use a budgeting tool answered which tool they use. These answers included various tools such as Google Sheets, Revolut account, their wallet etc. (Figure 8) This could be because their banking apps do not provide good/pleasing budgeting tools.

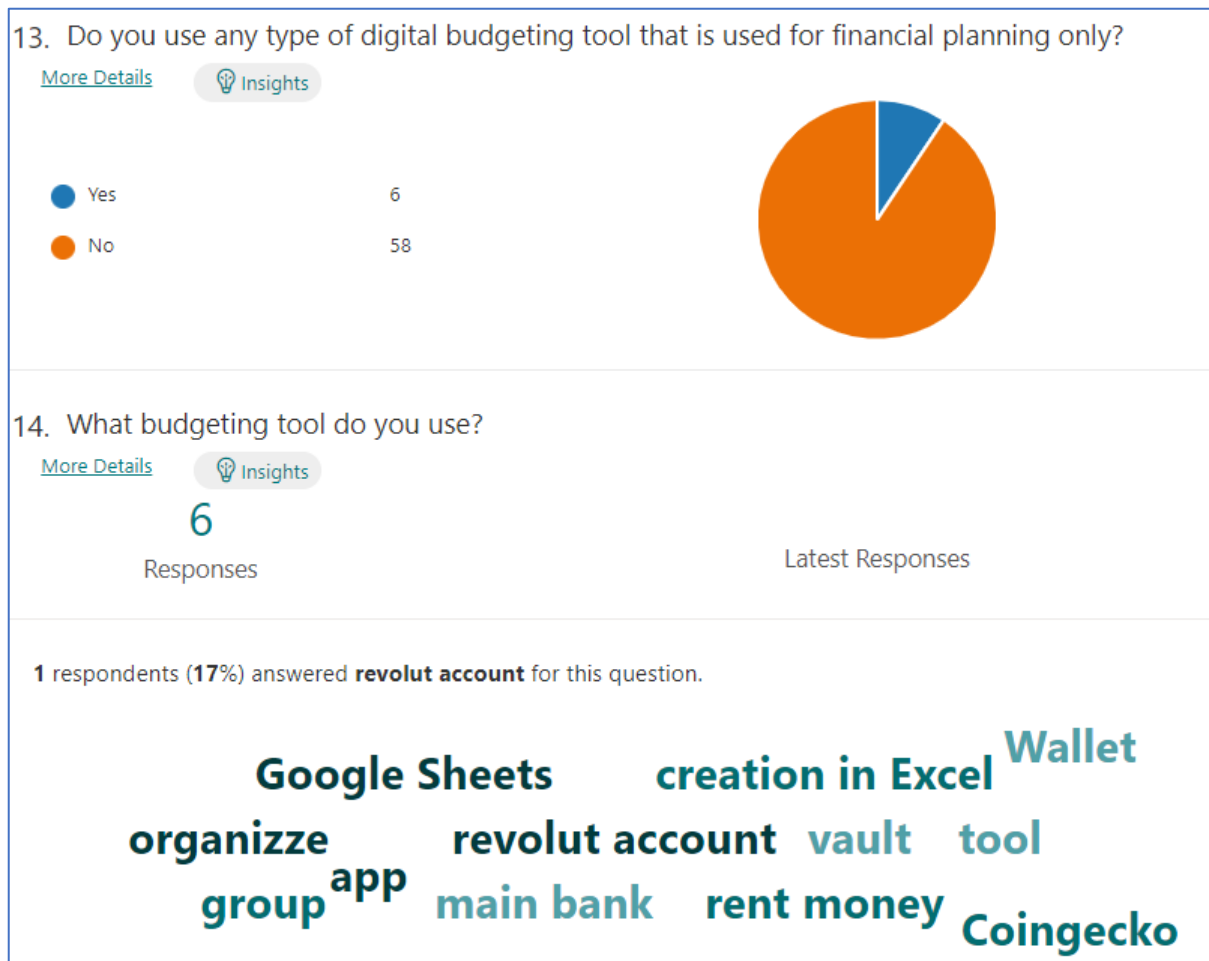


Figure 8 Survey budgeting tools

The budgeting tools within these applications might not be easy to use, as shown in Figure 9 below, users were asked about their confidence levels when performing different tasks in banking apps. Most users answered not confident to the task 'Making a Budget'.

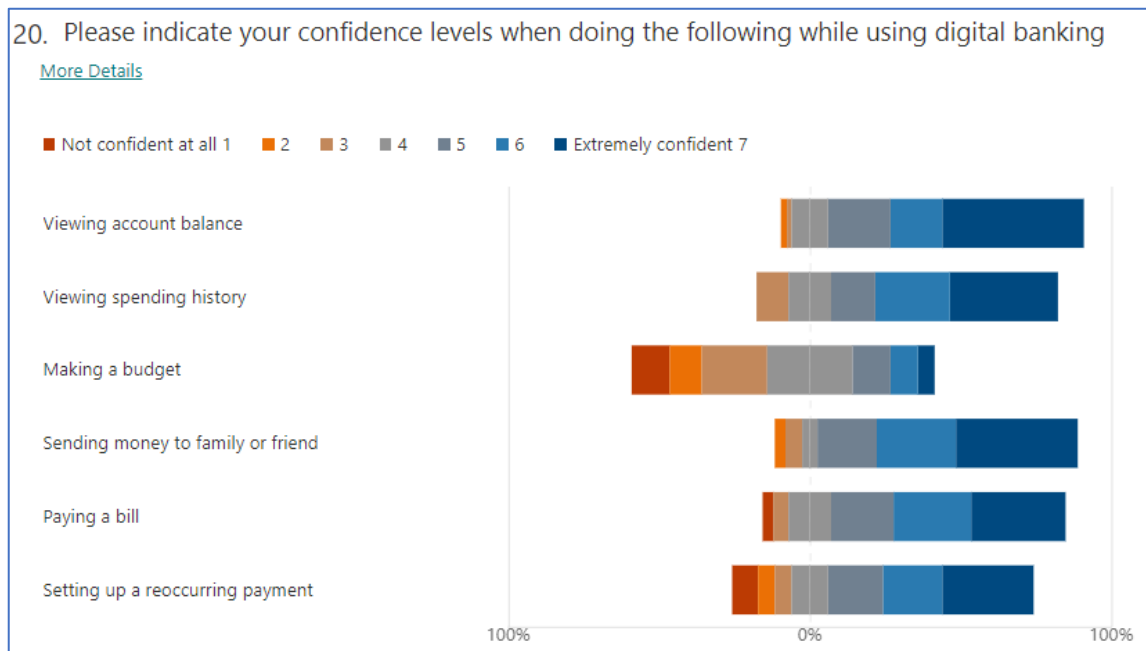


Figure 9 Survey confidence levels

3.4 User Personas

Another task carried out during the requirement's gathering stage was the development of user personas. Upon completion of the surveys, it was possible to analyse the data and come up with user personas to reflect the data. The goal of this activity was to define the potential users that could use the app and what type of characteristics they have, as well as determining the needs and goals of each user. Understanding the users' goals and different levels of technical abilities helped in making certain design decisions.

The first persona created was a young user named Jamie, who struggles with numbers from time to time and is an impulse buyer, therefore finds it difficult to save money (Figure 10). This user uses banking applications however suggested it would help to have an app where he could create saving goals to assist in saving and budgeting to save for nice things. This research activity was completed before any designs were produced for the application and the input from the users helped clarifying potential problems that need to be considered while designing and developing the application.



Figure 10 User Persona 1

The second user persona that was developed was a user named Amy who is 39 years old, who has multiple bank accounts set up for her family including an account which is used to deposit savings into. She finds it hard to manage these accounts sometimes as it involves switching between apps and it can make the process of saving and financial management unpleasant (Figure 11). This user suggests it would make her life easier and more efficient if it was possible to have a single app to analyse her family's finances and manage her savings account.

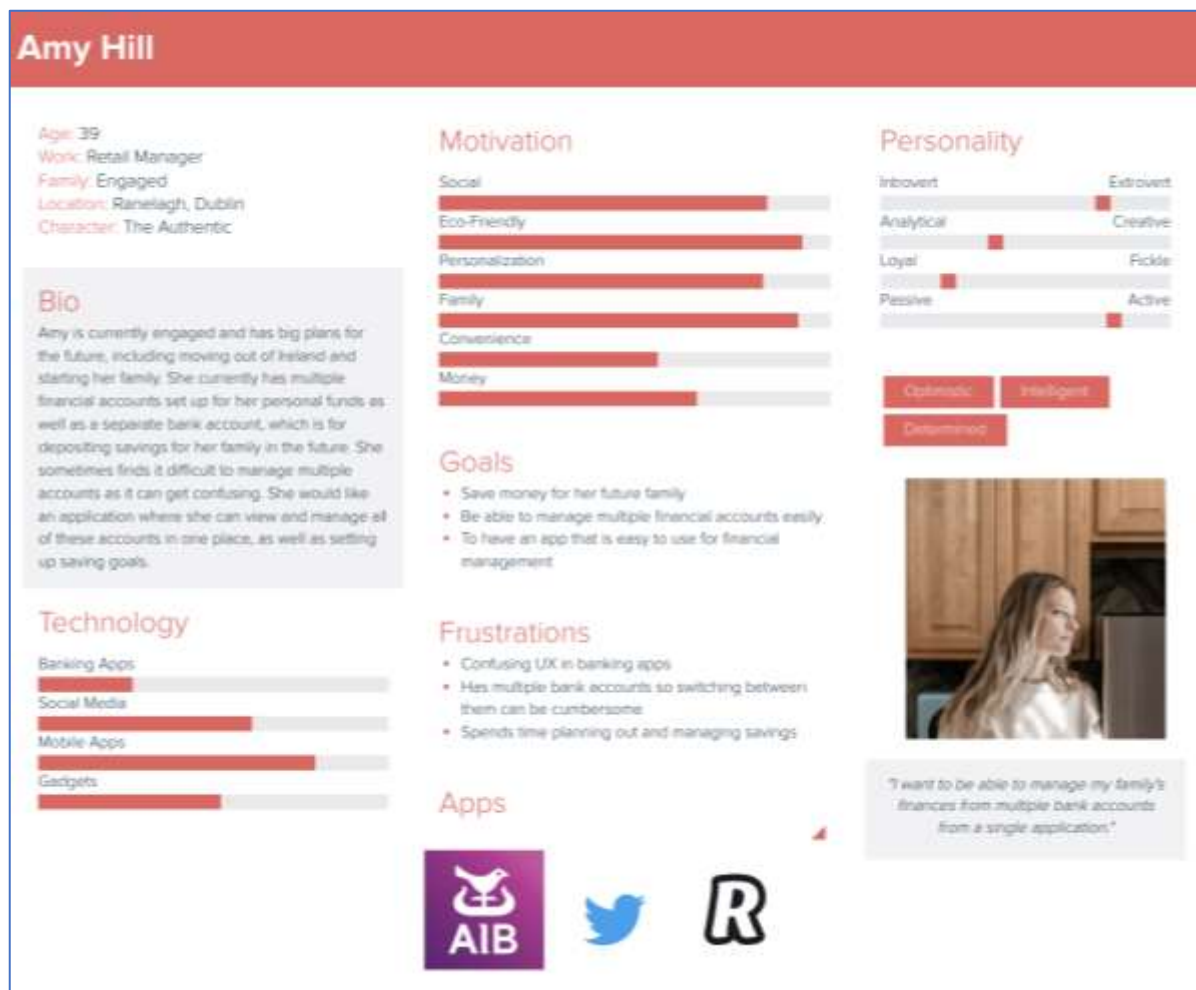


Figure 11 User Persona 2

3.5 Use Case Diagrams

Use case diagrams were created to model the behaviour of the system to help understand what the requirements of the system should be. Use case diagrams identify the possible interactions between a user and the system, and they can describe the high-level functions and scope of a system (IBM, 2021).

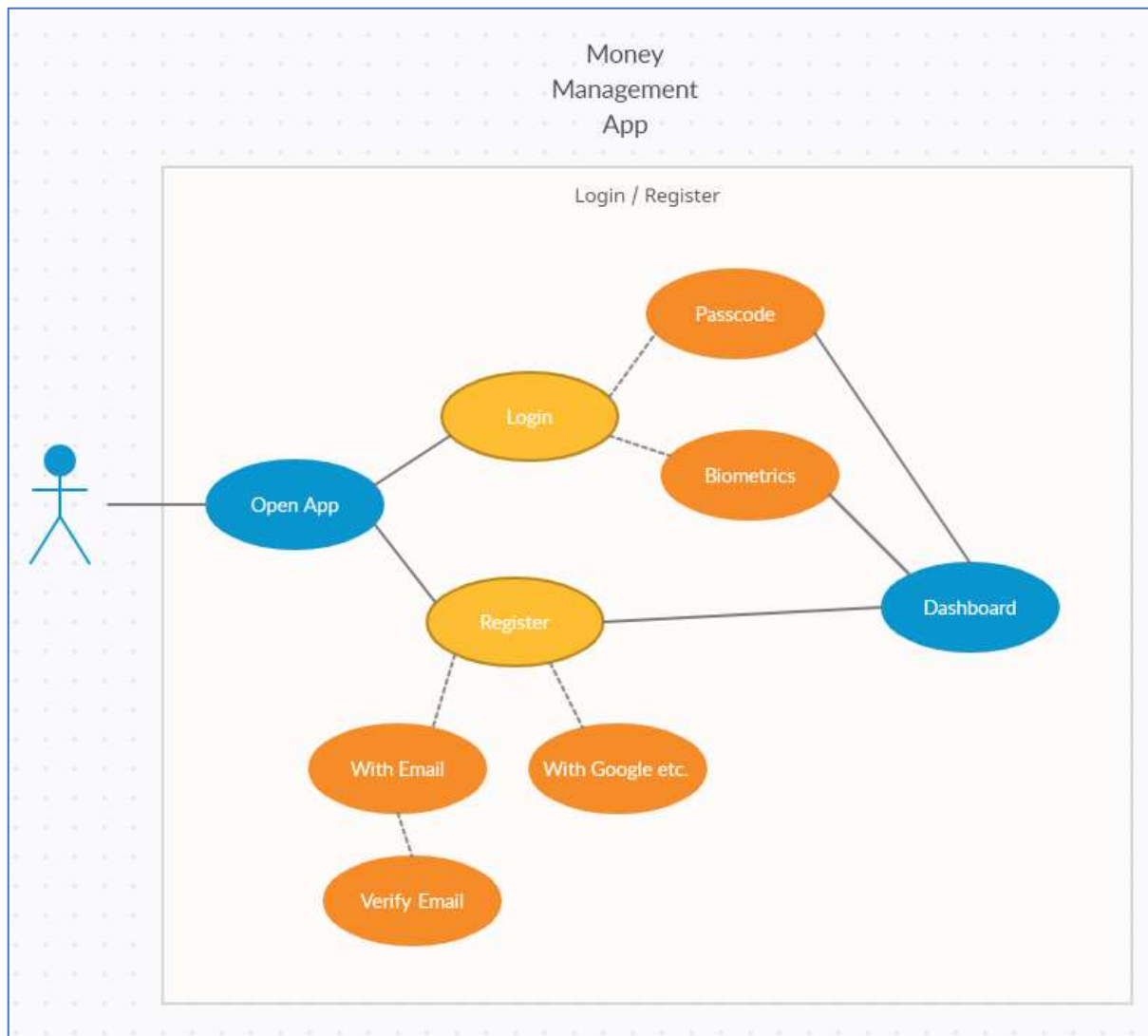


Figure 12 Use Case diagram 1

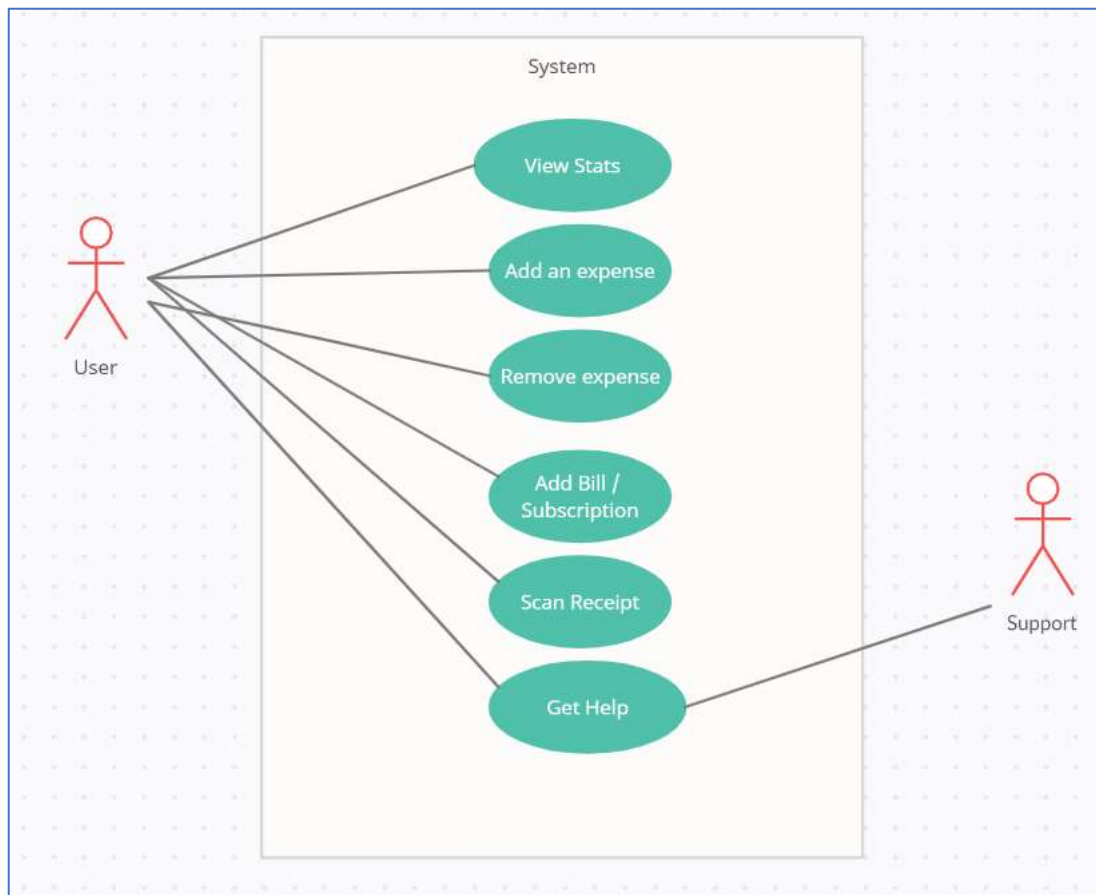


Figure 13 Use Case diagram 2

3.6 Requirements

Upon the completion of research gathering through surveys, user personas and use case diagrams, a list of requirements were derived. These requirements are categorized into four types of requirements - user, technical, functional, and non-functional requirements.

3.6.1 User Requirements

User requirements are used to describe what the user does with the software, such as what activities that the users must be able to perform.

A user should be able to:

- Sign up with an account or login to their existing account
- View their financial statistics in their dashboard such as their most recent transactions, their saving goals, bills, and subscriptions
- View expenses which is retrieved from external API
- Link one or more of their bank accounts to the application
- Create a new saving goal
- Add a new bill or subscription to their account
- Logout of their account

3.6.2 Technical Requirements

Technical requirements describe how the software should function and what its behaviour should be. These requirements describe the technical aspects and issues that the developers need to address for the project or software to execute and work successfully. These aspects can refer to how reliable the software is, performance-related concerns and how readily accessible it is.

Some of the technical requirements and considerations for this application include:

- The system will maintain availability
- Will it be technically feasible to develop a full-stack application
- Will it be possible to connect to external API's – will it be possible to retrieve information from banks such as recent transactions
- If the full stack application is not feasible, will it be possible to simulate the data and transactions
- The system will be reliable for users
- The system will have good performance with fast page load times
- The system will have good authentication and authorization standards
- The system will provide privacy for each user – the user interface will not allow anyone to view sensitive information
- The system will be maintainable – the software will maintain its integrity

3.6.3 Functional Requirements

After gathering research data from users through surveys, it was possible to derive a list of requirements for the application. Functional requirements are services or components that the software must offer. Functional requirements can be created using a numbered list to determine the most important features, with the first item being the most important feature.

The application should be able to:

1. Let users create accounts and login to existing accounts
2. Require users to authenticate their account on every login
3. Let users link their various bank accounts
4. Integrate with external APIs
5. Let users perform various actions such as viewing recent expenses, adding updating, and removing bills/subscriptions etc.
6. Let users add new saving goals
7. Store information in a database
8. Allow users to logout of their account manually

3.6.4 Non-Functional Requirements

Non-functional requirements in software define the performance and scalability of an application. If these requirements are not met, the application does not stop working, however it might not work and perform as well as it possibly could. These requirements are contrasted with functional requirements that define specific behaviour or functions.

The application should:

- Be easy to use, with pleasing interfaces and easy to remember how to use
- Load pages within 3 seconds when the user enters the app/navigates through pages
- Have strong and reliable security for users
- Be able to handle large volumes of traffic without issues
- Be available for users in different continents
- Have consistent design patterns throughout screens

3.7 Feasibility Study

There are numerous technologies that could be used to build a mobile application. Each technology has its own language and frameworks available to use. If the application is being developed to be native to Apple OS, Swift is a popular language to use. This offers advanced features with minimal coding that can be maintained. If the app being developed to be native to android devices, Java and Kotlin are popular languages that are used to accomplish this. These languages also offer advanced features and keep the app flexible.

However, in recent times, application development frameworks have been created to help developers create native (cross-platform) apps. Two popular frameworks that were explored and considered to build this application were Flutter and React Native.

Flutter is a cross-platform mobile application development technology that uses the programming language Dart. This is an open-source platform SDK created by Google that extends a wide range of plugins backed by Google and allows mobile apps to be built for both iOS and Android platforms.

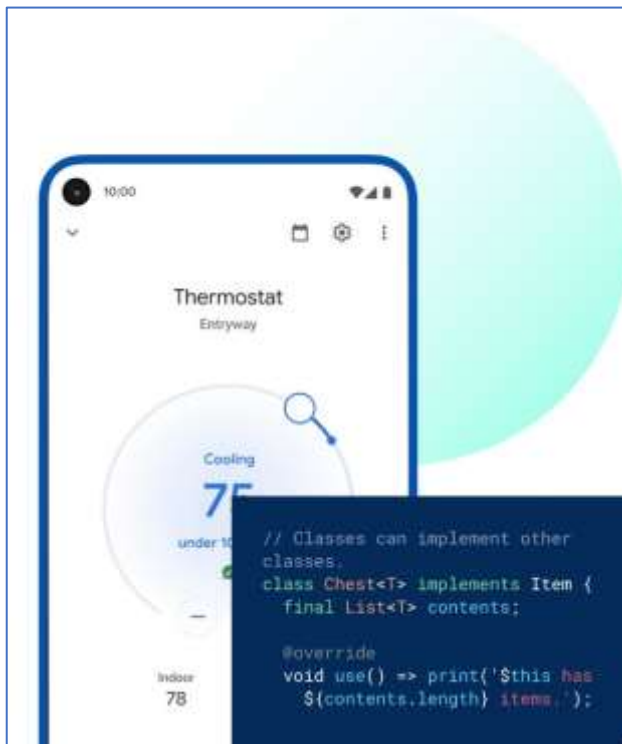


Figure 14 Flutter

React Native is an open-source framework written in JavaScript that has become one of the most popular native mobile app development technologies. React Native allows developers to create mobile apps with JavaScript using the same design as React. Apps build with React Native cannot be distinguished from an app build with Swift or Java. The likes of Instagram, Amazon and Uber are created using React Native.



Figure 15 React Native

Flutter's popularity has grown in recent years and is becoming one of the more popular native mobile app development technologies, however for this project, React Native will be used to develop the app. The reason for this is that for the backend, Node JS, MongoDB and express will be used. Therefore, React Native will complement these technologies well as it fits in with the MERN stack.

For the backend of the application, MongoDB will be used as the database, Node JS and Express will be used for the server-side and API.



Figure 16 MERN stack

3.8 Project Plan

For the project plan, a project plan schedule will be used to assist in project management. Each sprint and the sprints tasks will be laid out and the progress of them can be tracked by

the status. The possible statuses are 'Not Started', 'In Progress', 'Complete', 'Overdue' or 'On Hold'. This will help make it clear what needs to be done by the end of each sprint and this should allow me to deliver a successful product. The project will be developed over 9 sprints with each one lasting two weeks. At the end of each sprint there are a set of deliverables required.

MAJOR PROJECT PLAN

| PROJECT NAME | MAJOR PROJECT | | | PROJECT SUPERVISOR | Marian Mc Donnell | |
|------------------------------------|---------------|----------|---------------------|--------------------|-------------------|---------------------------------------|
| PROJECT DELIVERABLE | | | | | | |
| START DATE | 10/01/2022 | END DATE | TBC* | | OVERALL PROGRESS | 0% |
| TASK NAME | START DATE | END DATE | DURATION in days | % COMPLETE | STATUS | NOTES |
| SPRINT 1 – Research & Requirements | 10/01/22 | 21/01/22 | 12 | 30% | In Progress | |
| Requirements Chapter | 10/01/22 | 21/01/22 | 12 | 10% | In Progress | |
| Research Chapter | 10/01/22 | 21/01/22 | 12 | 60% | In Progress | Use research from Research module CA? |
| Create Survey | 11/01/22 | 11/01/22 | 1 | 100% | Complete | |
| Create User Persona(s) | 11/01/22 | 11/01/22 | 1 | 100% | Complete | |
| Create Use Case Diagrams | 12/01/22 | 13/01/22 | 2 | 80% | In Progress | |
| Features Backlog | | | | 0% | Not Started | |
| Paper Prototypes | | | | 0% | Not Started | |
| Define Test Plan | | | | 0% | Not Started | |
| Hi-Fi Prototype | | | | 0% | Not Started | |
| SPRINT 2 – DESIGN | 24/01/22 | 04/02/22 | 12 | 0% | Not Started | |
| Task 1 | | | | | | |

Figure 17 Project Plan Schedule

Microsoft's Tasks app is also used as part of the project plan. This works as a Kanban tool where tasks are created in the backlog, then moved to the 'To-Do' phase and then to the 'Doing' phase and finally to the 'Done' phase. This will help to make it clear what needs to be done in each sprint.

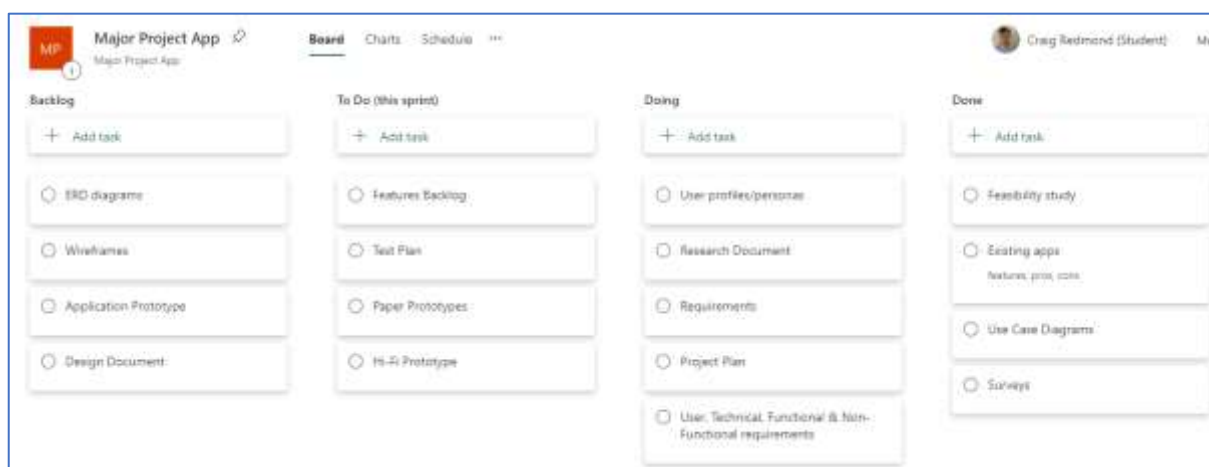


Figure 18 Microsoft Tasks Planner

4 Design

4.1 Introduction

After the completion of gathering research and requirements for the app it was then possible to move on to the design phase. This phase involves several design aspects such as application design, system architecture design, user interface design etc. This chapter describes the design of the application. The aim of the design phase is to develop the designs that can be used to achieve the desired project goals that are established in the requirements phase. The design of the app is split into system architecture, application design and user interface design.

4.2 System Architecture

System architecture design is a conceptual representation of the components and subcomponents that reflects the behaviour of the application. It functions as a blueprint for the system and developing project, laying out the tasks necessary for to be executed by the developers. The steps to designing a good system architecture are to first analyse the requirements for the system, define use cases for them and identify processes/modules to implement these use cases, then to select an operating system and hardware platform, assign requirements to individual processes/modules and then define sequence diagrams for these processes.

As shown in Figure 19 below, the systems architecture is separated into four parts – the client, API, server and back-end. The client side will be a user using the application on their mobile device, and the client will perform actions in the app that will make requests to the API for example getting an access token. The server will also communicate with the client side when a user is logging in or registering, a request will be sent to the server and some server-side code will run to login or register a user. The API that will be used is Nordigen and the server will be using Node JS and Express. The server will then communicate with the back end which will be a database created with MongoDB. The server will send documents to the back end and will in turn receive a response in JSON format.

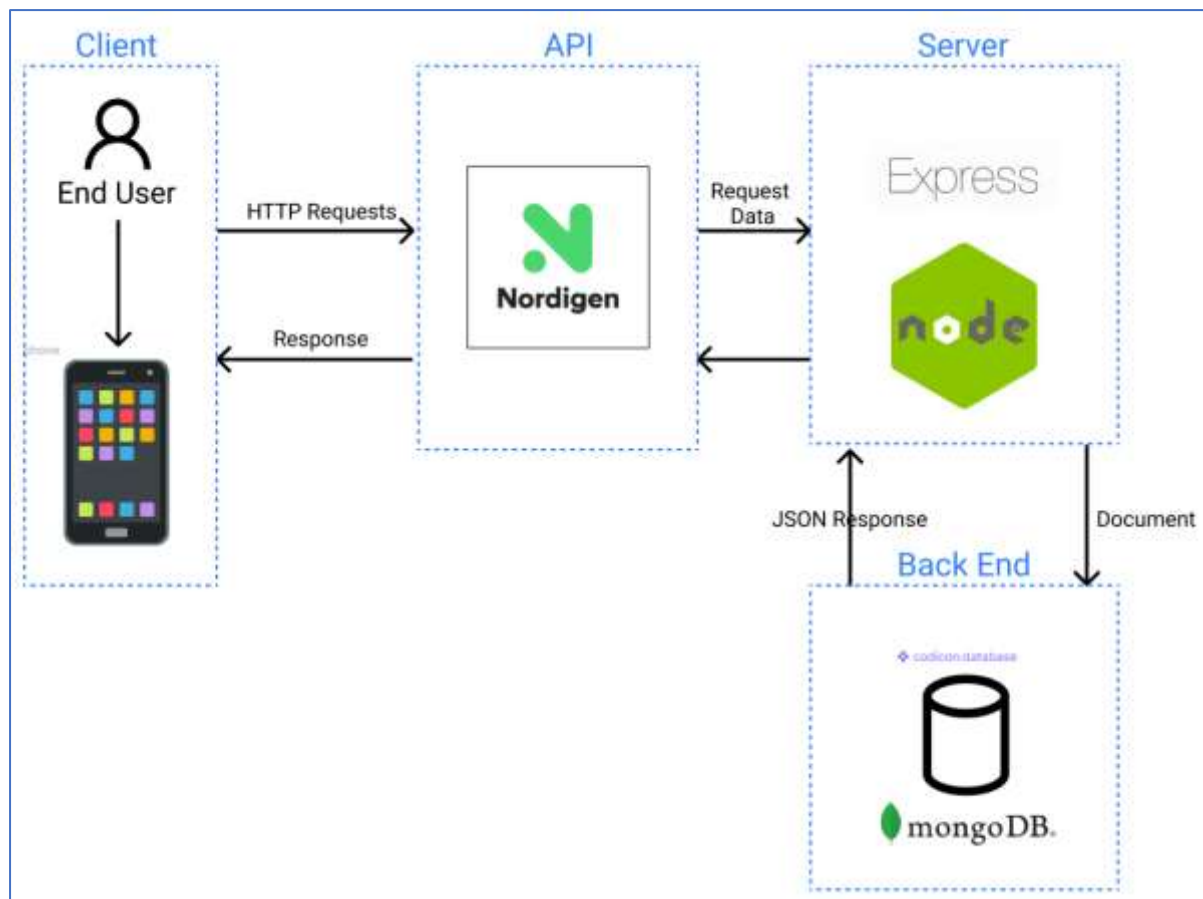


Figure 19 System Architecture Design

4.3 Application Design

4.3.1 Technologies

The technologies that will be used to develop the application are React Native, Node JS, Express, MongoDB, Nordigen API, GitHub and Figma. The main technologies that are used for creating this full stack application are part of the MERN stack (Mongo Express React Node).

React Native will be used to develop the front end of the application. React Native is an open-source framework written in JavaScript that is used for creating native mobile applications – for both Android and iOS devices. React Native is the same framework as React, however the difference is it uses native components instead of using traditional web components as building blocks. It uses components such as images, text, views etc.

Node, Express and MongoDB will be used as the back end of the application. Express is a framework for building web applications on top of Node. Node is a JavaScript runtime tool that allows developers to use the JavaScript language as a server-side language. Both tools together can communicate to the applications database – MongoDB. MongoDB is a document based that is used to store documents in JSON-like format. This database which will be used to store information for the application, such as user emails, passwords, access tokens etc.

The Nordigen API will be used as a middleman between the user interface and the backend of the application, which will be used to let users access their bank accounts and transactions. Nordigen is a freemium open banking data platform that provides free access to bank data and access to data products for analysis and insights. The Nordigen API provides a set of endpoints that allows developers to integrate their transaction categorization and insights solutions to systems.

Figma will be used for creating the design and wireframe documents. Figma is a collaborative tool where developers can work on files at the same time. It also allows developers to build libraries of reusable components such as styles, fonts etc. It makes creating consistent wireframes and prototyping easy and efficient.

GitHub will be used as a version management control tool. GitHub allows developers to upload and share code between each other. Version control helps the developers track and manage changes to a software project's code. As a software project grows, good version control becomes essential.

4.3.2 Design Patterns

Design patterns are typical solutions to common problems in software design. Each pattern acts as a blueprint that can be customized to solve design problems in code.

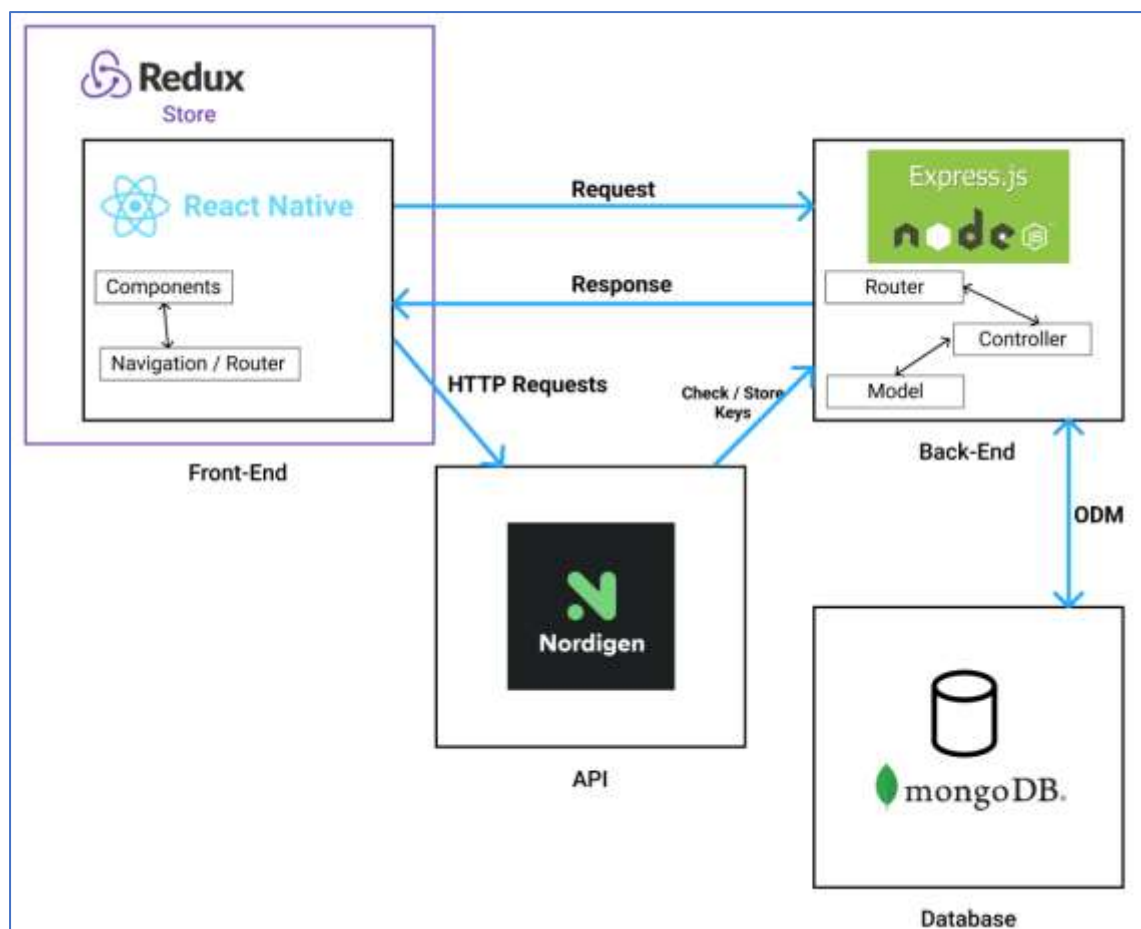


Figure 20 System Design Pattern

The pattern used in the server will be the model router controller pattern. The Express framework and Node will be used in the server (Figure 20). The router will be responsible for handling HTTP requests and for each request sent, the router will call the appropriate controller and a corresponding function. The controllers will contain classes which have functions that will run to carry out any CRUD functionality. For each controller class, it will use a model to access the database. These models are patterns that provide an interface to access the database. An example of a model could be for example 'User'. These models provide specific data operations without exposing details of the database. Usually in the model classes a connection is made to different database collections.

The server communicates with the database using object document mapping (ODM). This is used for mapping an object model and a NoSQL database such as MongoDB. In these databases documents are stored in JSON format in a collection (Dhruw, 2020). For example, if a new user is registering an account, a new user document would be inserted into the users collection, which would be displayed in JSON format.

For the front end of the application, React Native will render the user interface through the use of components. Navigation throughout different screens in the app will be handled by the React Router package, which enables navigation among screens of various components in the app. The React Native app will use Redux for managing state in the app. State management is discussed in the following section.

4.3.2.1 REST API

A REST API will be used in this application. Representational state transfer (REST) is a software architectural style that is used to guide the design and development of the architecture for the web. An application programming interface (API) is a set of rules that define how applications or devices can connect to and communicate with each other (IBM Cloud Education, 2021). REST APIs communicate through HTTP requests to perform standard database functions such as creating updating and deleting records. For example, the REST API would use a GET request to retrieve a record from the database, a POST request to add a record, a PUT request to update a record and a DELETE request to delete a record. In this app, various HTTP requests will be sent to the server and the server code will run to handle each request. For example when a user is logging in to their account, a POST request will be sent to the server along with parameters such as the users email and passcode, the server will then communicate with the database to retrieve the user document, create an authentication token and log the user in to the app.

4.3.2.2 State Management

The application will require a structured state management, both for the local and global state. For handling the local state in the component files, the useState hook in React can be used. The useState hook is a hook integrated into React that can be used to track state in a function component. State generally refers to data or properties that need to be tracked in an application (W3 Schools, 2022). For handling global state in the application, a library called Redux can be used. Redux is an open-source JavaScript library for managing and centralizing state in applications. Redux is commonly used in React applications as it makes it easier to manage state, by allowing certain variables to be accessed anywhere in the application, no

matter how far down the component tree they may be, making the code much more structured.

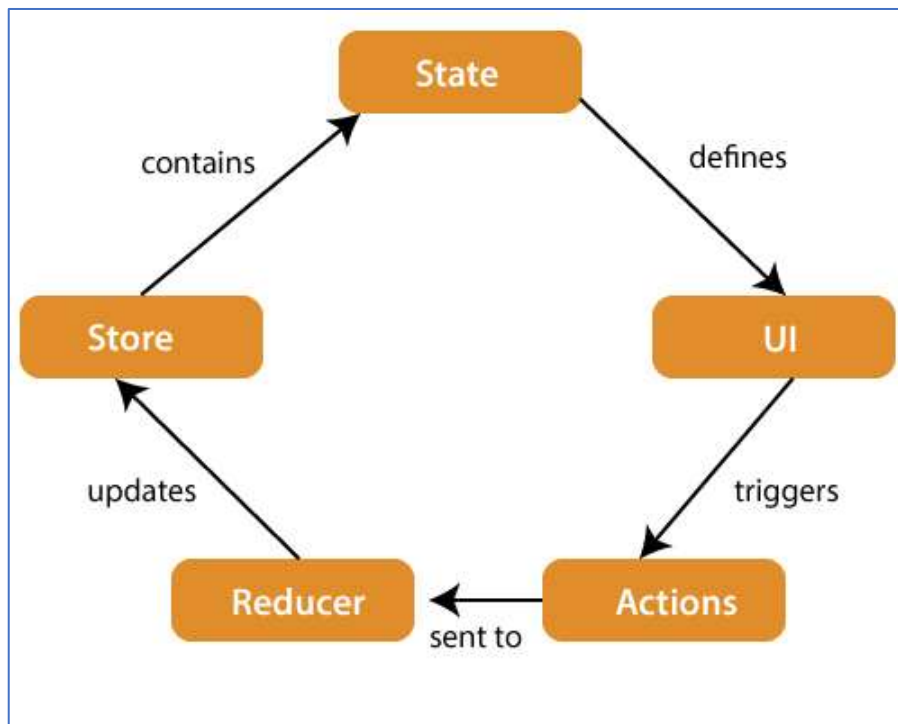


Figure 21 Redux Architecture - <https://www.javatpoint.com/react-redux>

Figure 21 shows the components of a Redux architecture. The **store** is like a container where the entire state of an application lies. It manages the status of the application and has a `dispatch(action)` function. An **action** is something that is sent or dispatched from the view which are payloads that can be read and managed by the reducers. It is an object created to store the information of the user's event. Different data that is stored in the action is the type of action, time of occurrence and the state that it looks to change. A **reducer** is a pure function that receives an action and the previous state of the application and returns a new state. The action describes what happened and it is the job of the reducer to return the new state based on that action.

4.3.3 Process Design

Process design is the process of creating various types of diagrams that gives the developers a clear picture of the way the application should function and what pieces of software need to be created to allow the application to function as expected. There are numerous types of process design that can be used such as class and sequence diagrams, user flow charts, ERD diagrams and database design diagrams.

4.3.3.1 Nordigen API Process and Integration

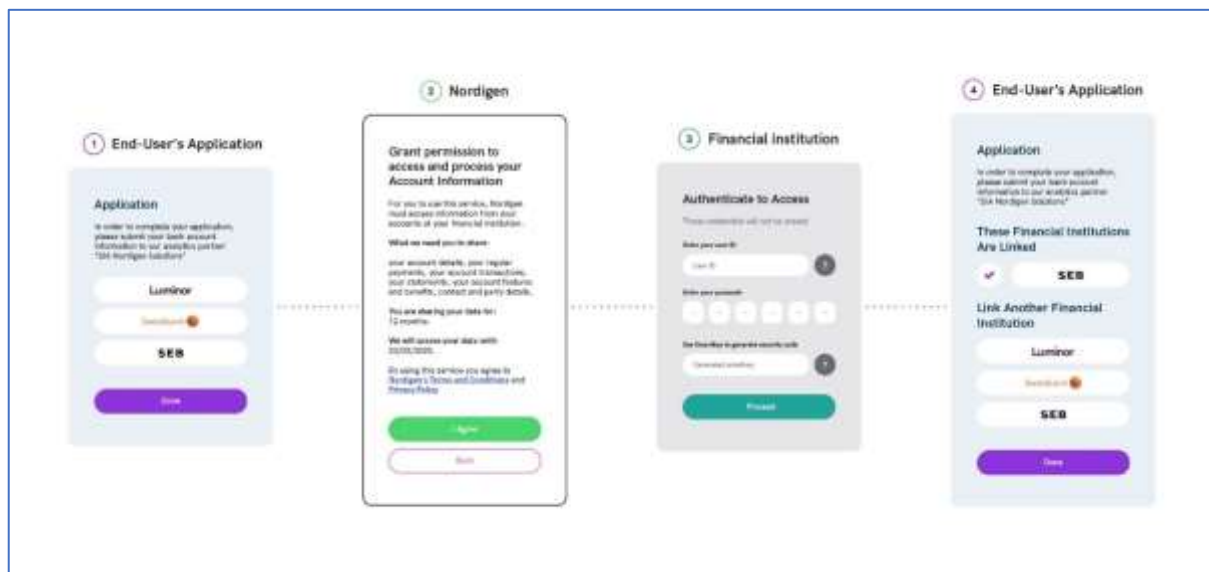


Figure 22 Nordigen Customer Journey

Figure 22 depicts an example of the customer journey for the client apps and end users. The process to integrate the Nordigen API and link bank accounts is described below:

1. The process begins within the end-user's application – this could be a website or mobile application. A HTTP request is made from the React Native application to communicate with the Nordigen servers to access private security tokens and keys. Once a successful HTTP request has been made to Nordigen, an access token is returned to the end user's application, which is a credential that is used to make requests to each of the Nordigen API endpoints. Once this request is handled, the end user is shown what financial service they can link with the application. The list of institutions available is retrieved from the Nordigen API and depends on financial institutions Nordigen supports within any given country.
2. After the user selects their institution, they are taken to a second screen that contains a consent text. This is a view that is hosted by Nordigen. Once the user accepts consent, they are directed to the next screen.
3. An interface is then provided by the financial institution for the user to link their account data. This interface is developed and hosted by the financial institution. After successful authentication, their access token is stored on the Nordigen server which enables Nordigen to fetch their bank account data from their respective financial institutions.
4. In the final step, the user is redirected to a URL specified by the developer. Once the user has successfully concluded the process, bank data such as transactions, accounts etc. can be accessed by the Nordigen API.

4.3.3.2 Class Diagrams

Class diagrams are the main building blocks of object-oriented modelling. It is used for general conceptual modelling of the structure of an application. Class diagrams can be used to model the objects that make up the system.

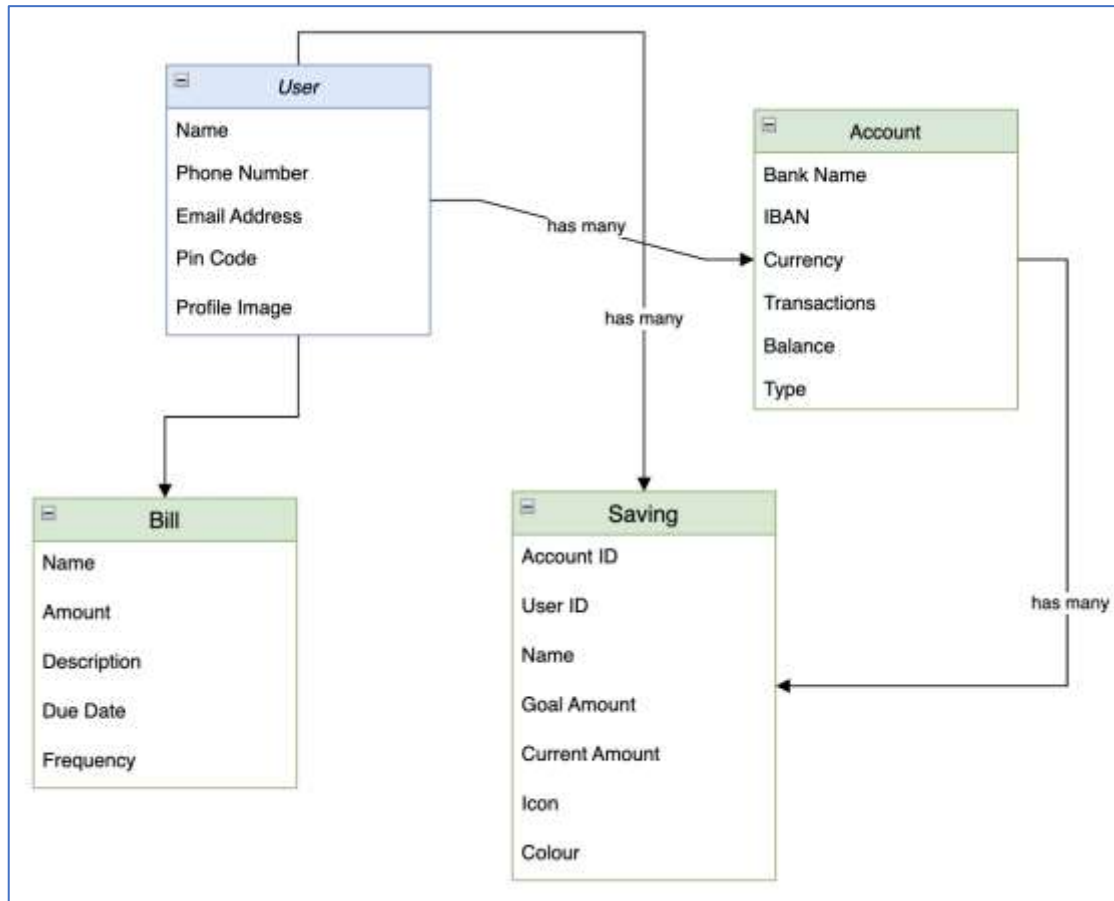


Figure 23 Class Diagram

4.3.3.3 Sequence Diagram

Sequence diagrams are interaction diagrams that detail how certain operations. Sequence diagrams capture the high-level interactions between the users of the system and the system (Figure 24).

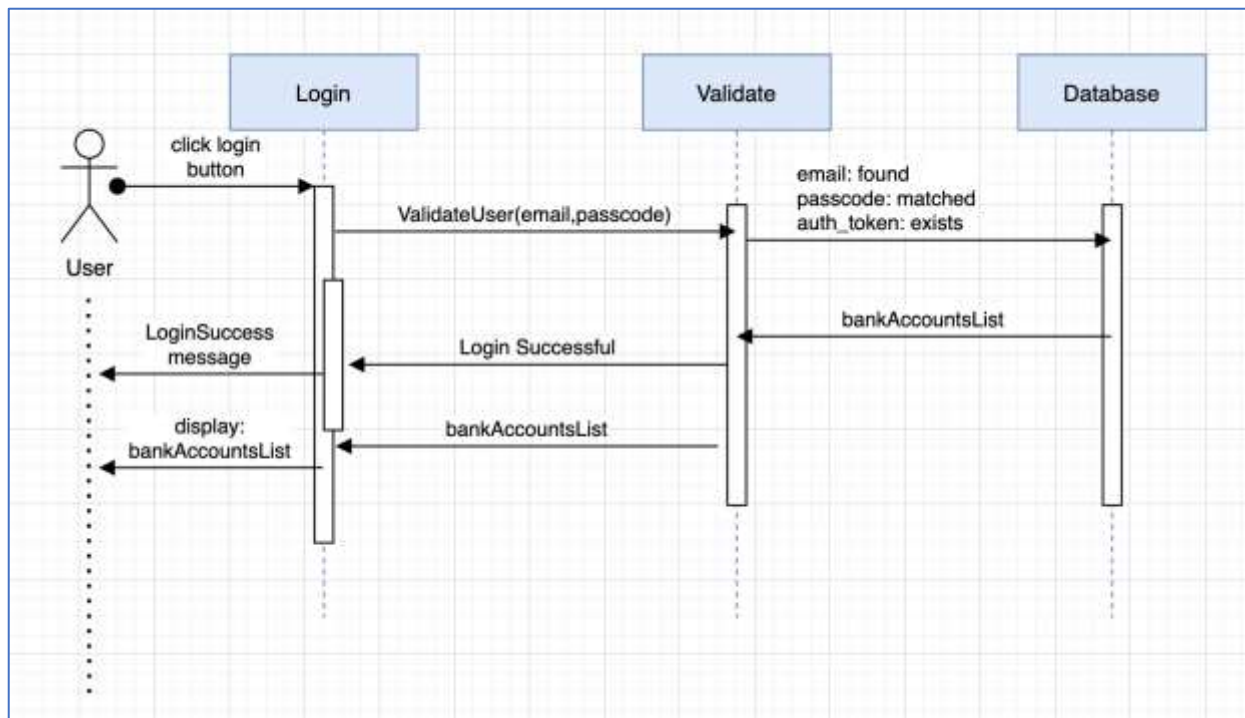


Figure 24 Sequence Diagram

4.3.3.4 Flow Charts

Flow charts are used to display the separate steps of a process in a sequential order. The flow chart below (Figure 25) shows the flow of a user logging into their account and being brought to their dashboard.

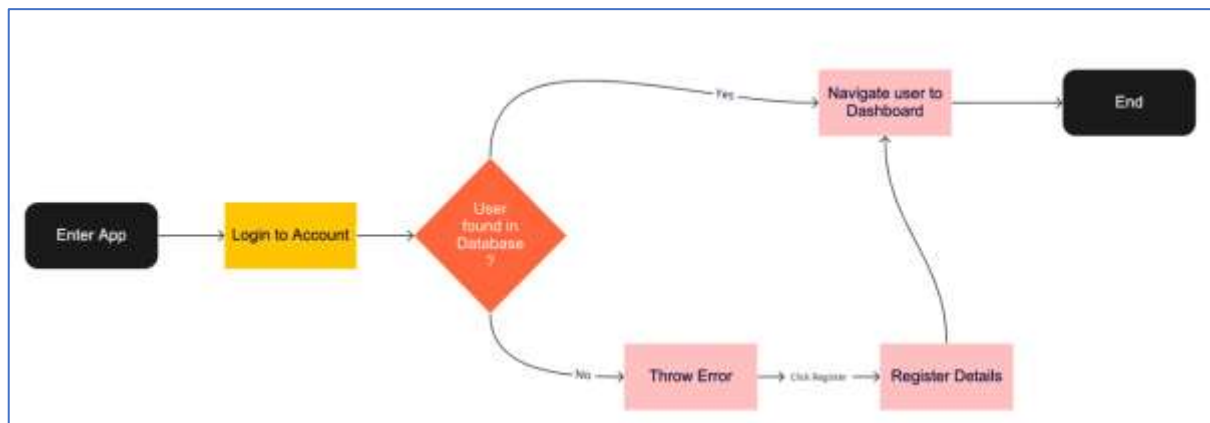


Figure 25 Flow chart - Logging into account

The flow chart in Figure 26 below shows the steps that a user would take when they are going to add a new saving goal.

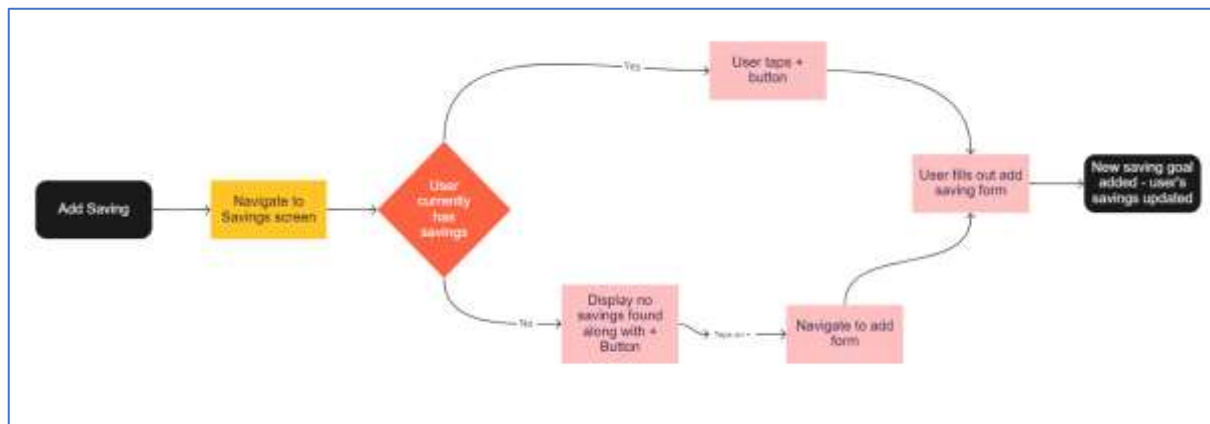


Figure 26 Flow Chart - Adding a new saving goal

4.4 User Interface Design

After completing the application design, the next stage of the design is the user interface design. This is where the paper prototypes will be sketched, then the wireframes for each screen will be designed based on the paper prototypes, a style guide will be made to encourage consistent design patterns. Various flow diagrams will be made to communicate specific activities through a sequence of actions/movements within the app.

4.4.1 Wireframe

The next task in the design process was the design of wireframes that should represent how the application should look. Figma was used as a tool for designing the wireframes.

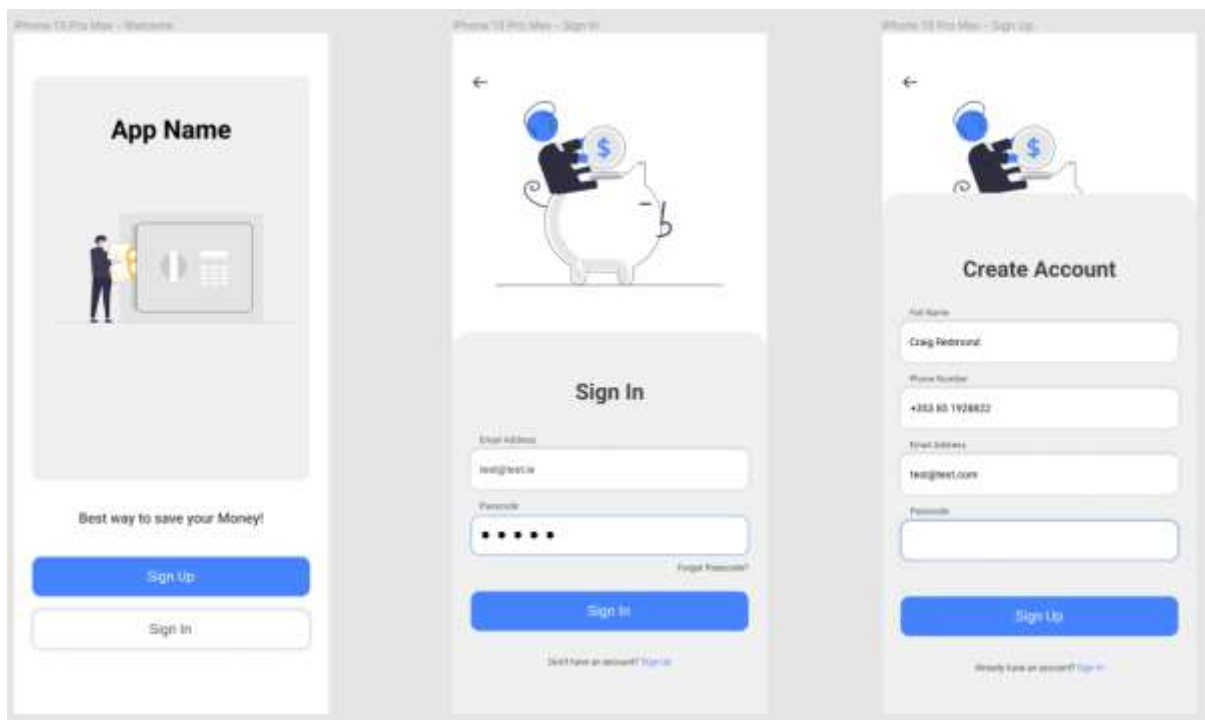


Figure 27 Authentication Wireframes

A prototype of the various steps for user authentication are displayed in Figure 27. It shows the welcome screen which will be displayed when a user enters the app, from here they can navigate to either the sign up or sign in screens. Each of these screens portray the potential style of the input boxes, buttons, icons and SVG images. Some of the colours are different as the design system/style guide were still being created. Each screen displays good readability and colour contrast.

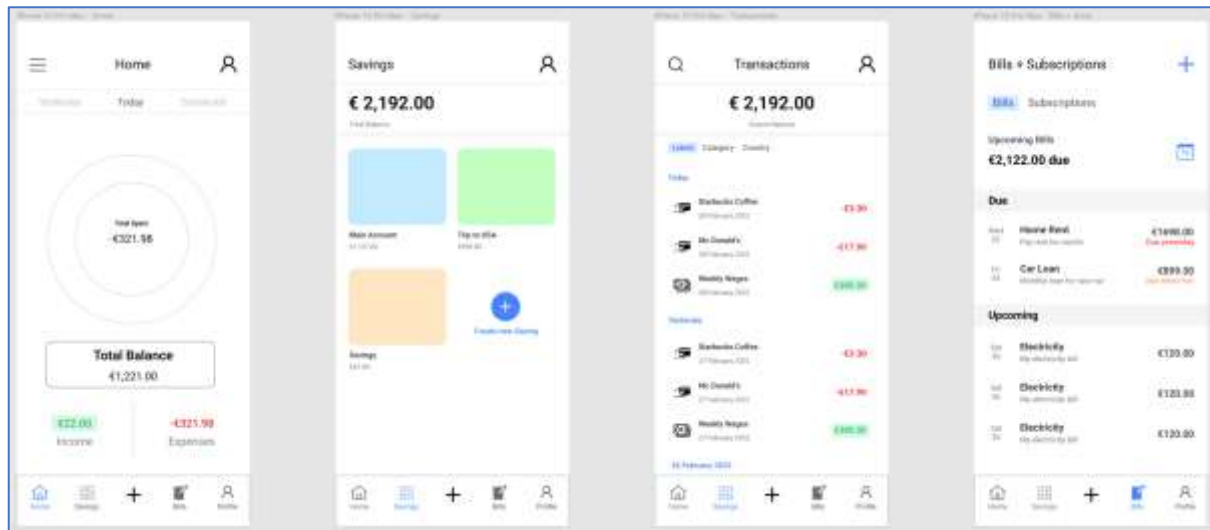


Figure 28 First iteration of main screens

The first iteration of the main screens are shown in Figure 28. These are the potential screens that a user can navigate to from the bottom navigation bar. For example, the home screen is the screen that the user will see when they login to their account, from there they can navigate to their savings, recent transactions, their bills and subscriptions page and profile page. These wireframes are quite basic with minimal colour in each, the goal of this iteration was to get a good idea of how the screens might look and how the user flow might be.

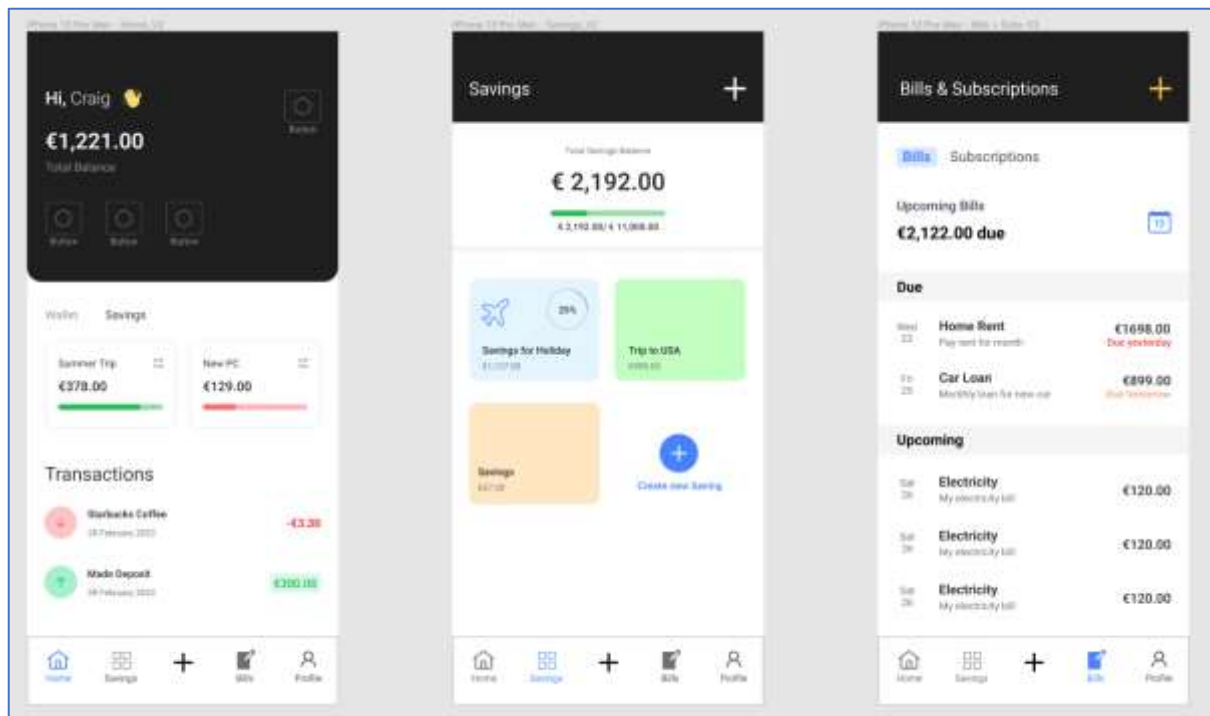


Figure 29 Second iteration of screens

In the second iteration of the wireframes (Figure 29), more colour was added to the screens and there was a slight change to their design. The primary colour which is blue was added throughout the screens and its complementary colour (yellow) was added in some parts such as the add button in the bills screen as a test to see how it looks and to see if it increases the readability. In the home screen, a bit of personalization was added, with a dark backdrop to increase contrast. Below that the user can access their saving goals which shows the amount of money in each and a status bar to show how close they are to reaching their goal. They can also switch to their 'wallet' which would display the amount of money that's available in their main account. Below this the user can see a list of their most recent transactions, where the icons were replaced by arrows with a colour surrounding them – this was to make it easier to visualize whether the transaction was an expense or income. In the savings screen, the total savings balance is displayed with a status bar to show how close the user is to completing all saving goals. Below this is each saving goal displayed in a box with an icon, the name of the goal, the amount of money currently in the saving goal and the percentage of completion.

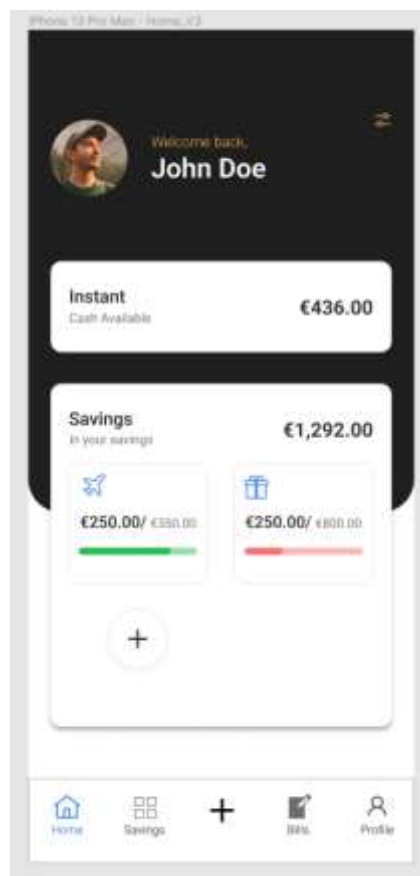


Figure 30 Different design of home screen

Another style of the home screen was designed in Figure 30. In this wireframe a user profile picture is added to add more personalization. The user can view their 'instant cash' that is currently available in their main account and below this they can see each of their saving goals and from here they can add more goals. The use of icons was used in this iteration, to potentially make it easier visually for the user to see their saving goals.

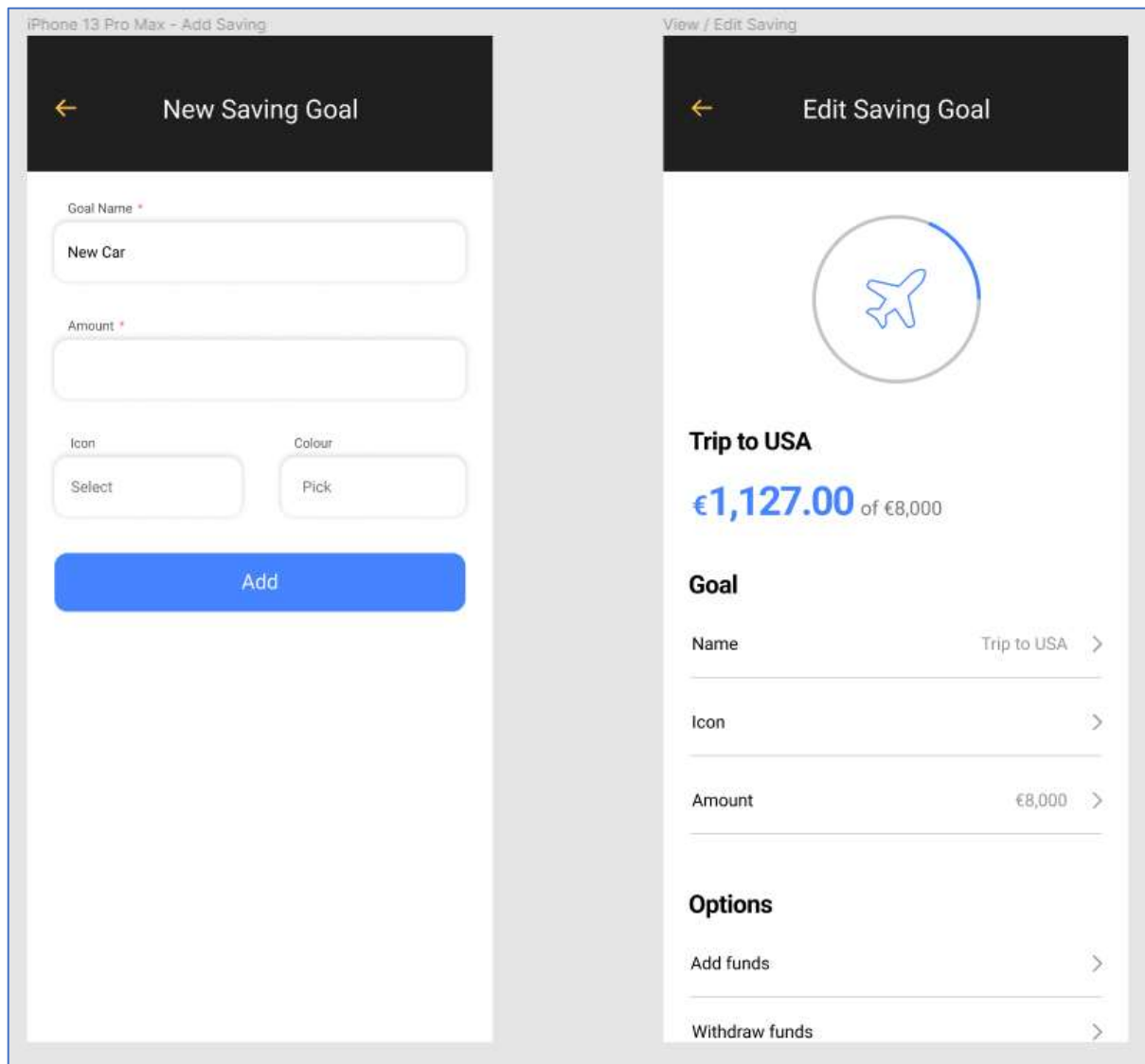


Figure 31 Add & Edit Saving goal screens

Figure 31 shows the design of the screens where users can add new saving goals and edit existing goals. The user should be able to input the name and amount of the goal, and optionally select an icon that will relate to the goal and a colour – which would show on the background of the goal in the savings screen (Figure 29). This would add more personalization for the user. When a user selects a goal, they would be taken to the edit screen and from here they could see the progress made and have the option to update the details of the saving goal. The user could also have the option to add and withdraw funds from the goal.

4.4.2 Style Guide

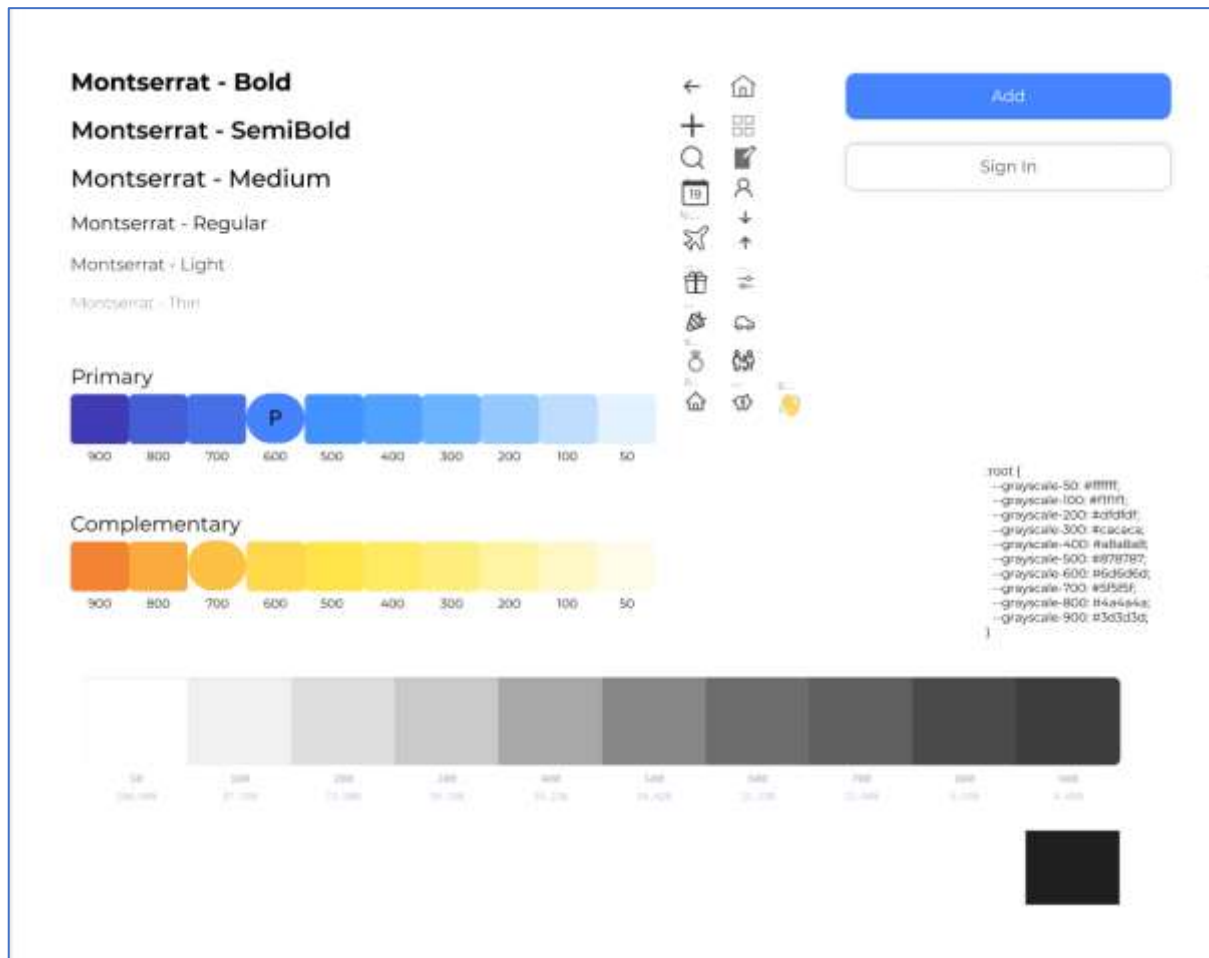


Figure 32 Style Guide

The first style guide/design system was designed for the app (Figure 32). This shows the fonts, colours, icons and components such as buttons that will be used throughout the app. Having a design system promotes good practices for having a consistent design throughout the app. The fonts were retrieved from Google Fonts, the colour scheme was generated from the Material Design colour palette creator (Material Design, 2022). Using this resource, a primary colour can be selected, and it generates various resources such as complementary and analogous colours.

4.5 Conclusion

The design of the system architecture and application design provided a better picture of what technologies should be used to develop the application and how it should function. It has shown how each system should function together in order to produce a full-stack mobile application.

This chapter has also discussed the overall user interface design of the application. The development of the wireframes has aided in showing how the user interface should look to the user and how the experience should be. A style guide has been developed that contains references such as colours and typography that can be used throughout the application's

components which in return results in the development process speeding up and consistent design throughout the application.

5 Implementation

5.1 Introduction

The next stage in the development process was to begin the implementation. This stage involved the process of implementing the first outlined design of the application. The steps for this was to set up the development environment, begin coding the front end of the app in a React Native project, set up the backend for the app which includes the MongoDB database and Express server, and the REST client for testing the endpoints for the REST APIs.

5.2 Development Environment

The code editor used to develop the application was Visual Studio code (VS Code). VS code is a code editor with support for development operations such as debugging, task running and version control. VS code has Git version control built in, so it was possible to open the app's project folder, checkout to the current branch that was being worked on and switch between branches, make commits and push to the remote origin – all without having to leave the code editor.

Git was used as version control. A separate branch would be created each time a new feature would be in development, for example a branch such as “authentication-branch” would be created to develop the authentication system for the app. Creating separate branches for features is good practice because in a professional environment the main branch of a project is rarely worked on. It is generally bad practice to work and make changes directly to the main branch, as this can cause merge conflicts. Once the feature was completed, a commit was made, the code was then pushed and merged to the main branch.

While developing the user interface for the app, a mobile phone emulator was required to see the app being developed (Figure 33). For Mac, XCode was used to run its built-in emulators and on Windows PC, Android Studio's virtual devices were used. When the project is run in the terminal, the Expo CLI runs the React Native application and it then provides the option to run on an android or iOS device. Once the device is up and running, the app automatically connects to the simulator.

To test the REST APIs endpoints, a REST client was required. Insomnia was used to test these endpoints, which can be seen in Figure 43. Insomnia is an API testing tool that allows developers to access applications without interacting with the system or user interface. This was used to test Nordigen's API and the REST API used in the backend.

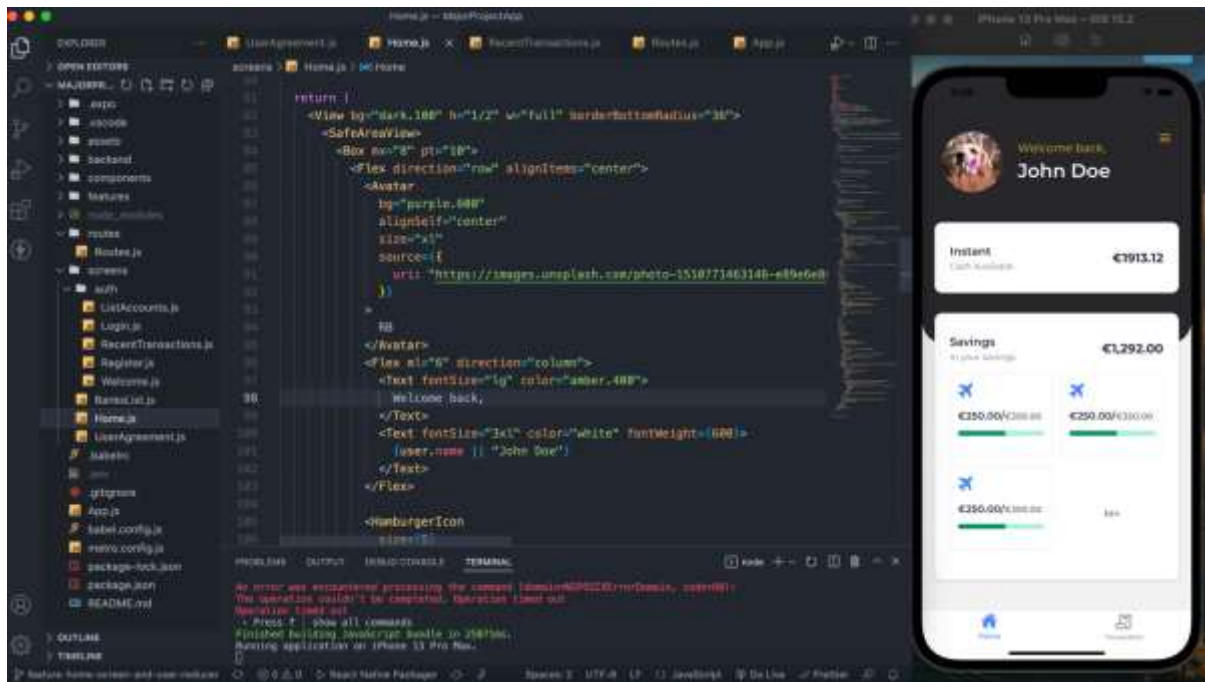


Figure 33 Development environment workflow

5.3 Database

MongoDB was used to create the database for the app. MongoDB is a document-oriented database program that is classified as a NoSQL database program. It uses JSON-like documents to store data. MongoDB atlas was used to create and host the database. Mongo Atlas is a cloud database that handles the complexities of deploying and managing a database on a cloud service provider such as AWS, Azure etc. Once the database was created, a cluster is then defined which is a NoSQL database in the public cloud. Within this cluster, the collections are then defined which resemble tables in an SQL database. An example of a collection would be a user's or sessions collection.

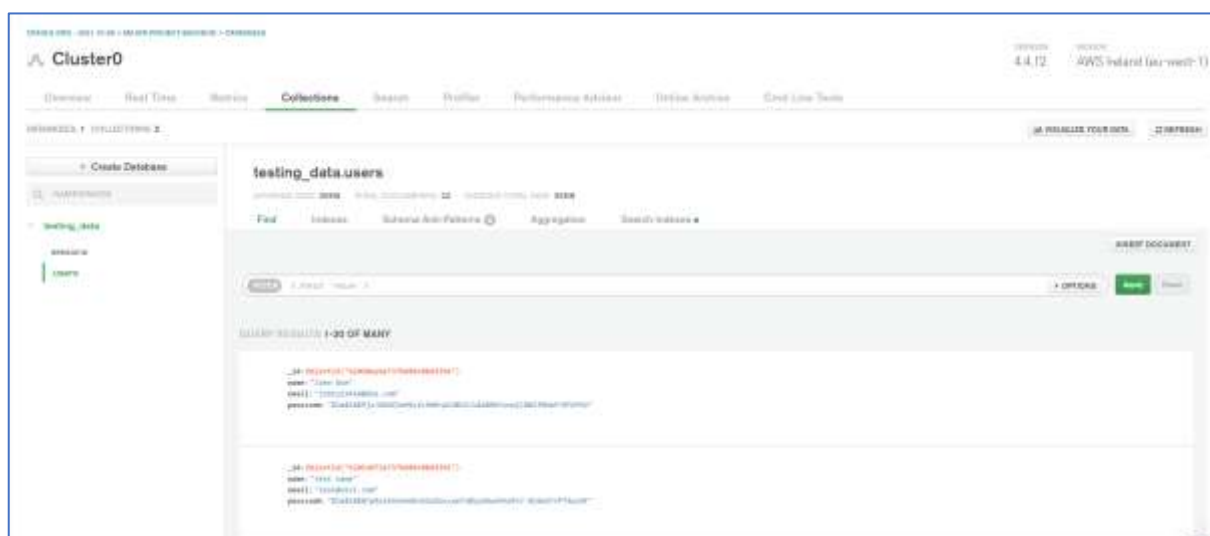
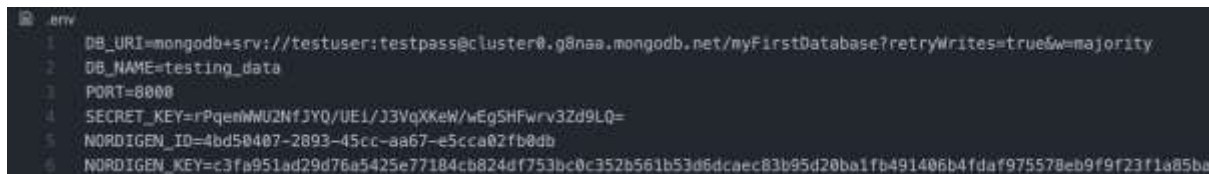


Figure 34 MongoDB cluster

5.4 Backend

For the backend of the app, an Express app was created with NodeJS for hosting a server. Express is a framework for building web applications on top of Node. This application was responsible for holding all the server-side code and handles requests being made from the client application. The backend application then communicates with the database and sends information back to the client app. This architecture creates a full-stack application.

5.4.1 Configuration – Environment Variables



```
1 DB_URI=mongodb+srv://testuser:testpass@cluster0.g8naa.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
2 DB_NAME=testing_data
3 PORT=8000
4 SECRET_KEY=rPqemMU2NfJYQ/UEI/J3VqXKeW/wEg5HFwrv3Zd9LQ=
5 NORDIGEN_ID=4bd50407-2893-45cc-aa67-e5cca02fb0db
6 NORDIGEN_KEY=c3fa951ad29d76a5425e77184cb824df753bc0c352b561b53d6dcaec83b95d20ba1fb491406b4fda975578eb9f9f23f1a85ba
```

Figure 35 Environment Variables

Certain environment variables are declared in a '.env' file. These variables contain sensitive information for the app such as the database name, URI, and the secret and IDs for the Nordigen API. This information is stored in this file so that other developers cannot view this information – this file is not pushed to any GitHub repository.

5.4.2 Application Configuration & Scripts



```
1 {
2   "name": "major-project-backend",
3   "version": "1.0.0",
4   "description": "REST API for major project",
5   "main": "index.js",
6   "type": "module",
7   "scripts": {
8     "server": "cross-env DEBUG=rest-api:* nodemon src/index.js"
9   },
10 }
```

Figure 36 Package.json file

Each Node/Express application comes with a package.json file which is where the configuration for the app can be set up. This is where the entry point for the app can be declared and when the 'npm run server' command is ran in a terminal, the code on line 8 runs which then starts up the server.

5.4.3 App/Server Initialization

In the 'server.js' file, an instance of an express object is initialized and stored in a variable. This object can then be used to set up middleware for the app such as JSON and URL encoded. Each router object is imported into this file and the routes are handled as shown in lines 19 – 21 (Figure 37). The app object is then exported.

```
c> server.js > ...
1  import express from "express";
2  import cors from "cors";
3
4  // routers
5  import usersRouter from "../routes/users.router.js";
6  import accountsRouter from "../routes/accounts.router.js";
7  import savingsRouter from "../routes/savings.router.js";
8
9  // Create app - an express object
10 const app = express();
11
12 // Middleware - recognize incoming requests as JSON
13 // recognize incoming requests as strings or arrays
14 app.use(express.json());
15 app.use(express.urlencoded({ extended: true }));
16 app.use(cors());
17
18 // handle router requests
19 app.use("/users", usersRouter);
20 app.use("/accounts", accountsRouter);
21 app.use("/savings", savingsRouter);
22
23 // export app
24 export default app;
```

Figure 37 Server JS file

5.4.4 Index File – App entry point

The 'index.js' file is where the server runs and makes a connection to the various DAO classes.

```
c> index.js > ...
1  import dotenv from "dotenv";
2  import { MongoClient } from "mongodb";
3
4  import app from "../server.js";
5
6  // DAO classes
7  import UsersDAO from "../dao/users.dao.js";
8  import AccountsDAO from "../dao/accounts.dao.js";
9  import SavingsDAO from "../dao/savings.dao.js";
10
11 // load environment variables from .env file
12 dotenv.config();
13
14 const port = process.env.PORT;
15 const dbUri = process.env.DB_URI;
16 const client = new MongoClient(dbUri);
17
```

Figure 38 Index.js configuration

The app object, libraries and DAO classes are imported into this file. Different constant variables are instantiated such as the port and database URI. A new mongo client is set up

and set to a variable called 'client' which will then be used to connect to the MongoDB database (Figure 38).

```
18  try {
19    //Connect to client, call injectDB in users DAO passing in the client object
20    await client.connect();
21    await UsersDAO.injectDB(client);
22    await AccountsDAO.injectDB(client);
23    await SavingsDAO.injectDB(client);
24
25    // App will listen on port 8000
26    app.listen(port, () => {
27      console.log(`Listening on port ${port}`);
28    });
29  } catch (err) {
30    // Log any errors to the console and exit
31    console.error(err.stack);
32    process.exit(1);
33  }
```

Figure 39 Index file - running server

A try-catch block then runs which connects to the database and DAO classes, and then the 'listen' function is ran on the app object which runs the server on the specified port.

5.4.5 App Structure

The backend project was structured in three sections – the routes, controllers and DAO (Data Access Objects). In the routes folder it contained various routers for the app such as a user's router, which listens for and handles URL requests. Each route that is defined calls a function in a controller class. For example, in Figure 40, if a post request is sent to /login, the 'login' function is called in the user's controller.

```
1  // Import router object      You, a week ago • register request stores a user in database
2  import { Router } from "express";
3
4  import UsersController from "../controllers/users.controller.js";
5
6  // create router object
7  const router = Router();
8
9  // Handle URL requests
10 // If a post request is sent using /x, run the appropriate method in user controller class
11 router.post("/register", UsersController.register);
12 router.post("/login", UsersController.login);
13 router.post("/logout", UsersController.logout);
14
15 // Export router
16 export default router;
```

Figure 40 Users router

Once the function is called in the controller class, this code is run. In the login function, it takes in a request body sent in the request – this might include an email and password for example. Some error handling is performed to reduce the chance of bad data being stored in the database. Within the controller functions, this is where the DAO objects are accessed. For

example, in Figure 41, the 'getUser' method is called in the users DAO class and the email is passed in. Within this function, a 'findOne' query is executed in the database to find a user in the database with that email (Figure 42).

```
static async login(req, res, next) {
  try {
    // destructure email and passcode values from request body
    const { email, passcode } = req.body;
    console.log(email);
    // if theres no email or if it is not a string, set status to 400 & return error
    if (!email || typeof email !== "string") {
      res.status(400).json({ error: "Bad email format, expected string." });
      return;
    }
    // if theres no passcode or if it is not a string, set status to 400 & return error
    if (!passcode || typeof passcode !== "string") {
      res
        .status(400)
        .json({ error: "Bad passcode format, expected string." });
      return;
    }

    // set variable to the result of getUser function from the usersDAO class
    let userData = await UsersDAO.getUser(email);

    // if there is no user found, return 401 error and error message
    if (!userData) {
      res.status(401).json({ error: "Make sure your email is correct." });
      return;
    }

    // set user to new user object
    const user = new User(userData);

    // if the passcode passed in does not match, return a 401 error with message
    if (!(await user.comparePasscode(passcode))) {
      res.status(401).json({ error: "Make sure your passcode is correct." });
      return;
    }

    // Login response - run loginUser method in UsersDAO, passing in the email and jwt
    const loginResponse = await UsersDAO.loginUser(
      user.email,
      user.encoded()
    );
    // if status is not success, return 500 response with error message
    if (!loginResponse.success) {
      res.status(500).json({ error: loginResponse.error });
      return;
    }
  }
}
```

Figure 41 User controller - login function

Figure 42 Users DAO – getUser()

This process is the same for developing most of the backend project. To test that the endpoints were working as expected, the Insomnia REST client was used to send requests to the server. In Figure 43 below, a POST request was sent to the server's register endpoint with an object containing a name, email and passcode. A 200 OK response is sent back from the server to the client with an authorization token and an object containing the user's information.

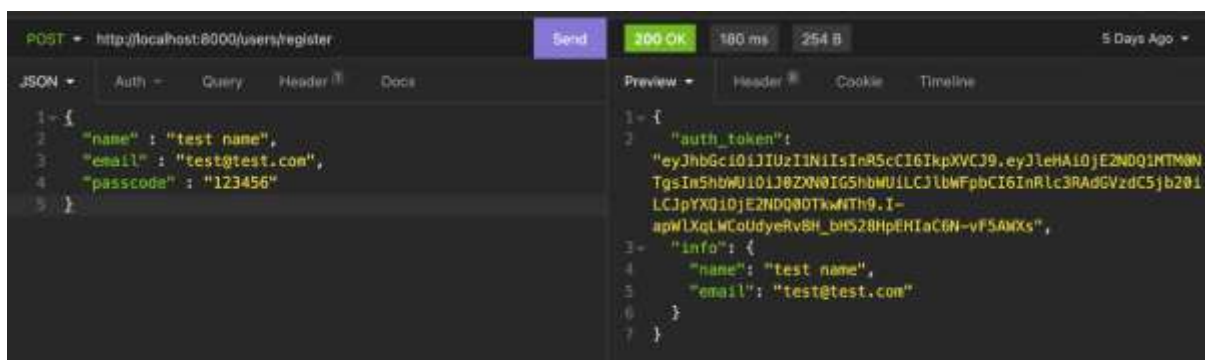


Figure 43 Testing endpoints in Insomnia

5.4.6 Linking an Account

When a user performed actions like linking an account or accessing their currently linked accounts, various functions were used to accomplish this. The accounts router, controller and DAO class were used in this process.

```
c:\routes > cd accounts.router.js > ...
1 // Import router object
2 import { Router } from "express";
3
4 import AccountsController from "../controllers/accounts.controller.js";
5
6 // create router object
7 const router = Router();
8
9 // Handle URL requests
10 // If a post request is sent using /x, run the appropriate method in user controller class
11 router.post("/", AccountsController.apiAddAccount);
12 router.get("/", AccountsController.apiGetAccounts);
13 // router.post("/logout", UsersController.logout);
14
15 // Export router
16 export default router;
```

Figure 44 Accounts router

The accounts router (Figure 44) file is responsible for handling requests sent to the '/accounts' endpoint. As shown on lines 11 and 12, if it is a POST request then the 'apiAddAccount' function is called within the controller and if it is a GET request, the 'apiGetAccounts' function is called.

```
9   static async apiAddAccount(req, res, next) {
10     try {
11       // getting token
12       const userJwt = req.get("Authorization").slice("Bearer ".length);
13       const user = await User.decoded(userJwt);
14       var { error } = user;
15       if (error) {
16         res.status(401).json({ error });
17         return;
18       }
19
20       // get nordigen auth token
21       const nordigenToken = await axios
22         .post("https://ob.nordigen.com/api/v2/token/new/", {
23           secret_id: process.env.NORDIGEN_ID,
24           secret_key: process.env.NORDIGEN_KEY,
25         })
26         .then((res) => {
27           return res.data.access;
28         })
29         .catch((err) => console.log(err));
30     }
```

Figure 45 Accounts Controller - Add function

In the add account function, the user must be authenticated to make the request, therefore a check is made to see if there was a bearer token passed in as a header. The bearer token is then passed into the 'decoded' function within the user class – which verifies the JSON Web Token (JWT) and returns a new user object. A Nordigen access token is required later in the function and this is retrieved in the function from lines 21 to 29 in the above figure.

```
31   // get request body data
32   const accountFromBody = req.body;
33
34   // get user existing accounts
35   let existingAccounts = await AccountsDAO.getAccountsByEmail(user.email);
36
37   // check if user has account already with the account ID
38   const accountExists = existingAccounts.accountsList.some((element) => {
39     if (element.account_id === accountFromBody.account_id) {
40       return true;
41     }
42   });
43
44   if (accountExists) {
45     res
46       .status(400)
47       .json({ error: "This account has previously been linked." });
48     return;
49   }
50 }
```

Figure 46 Retrieving accounts

The data from the request body is then stored in a variable on line 32. This data includes the bank name and account ID. A function is then called in the Accounts DAO class to retrieve the user's currently linked accounts if they have any. This function takes in the users email and retrieves the account documents that contain the provided email. A function is then created that is used to see if the user has already linked an account with the provided account ID on lines 38-42. A check is then made to see if the account exists and if it does, an error is returned to the user as a JSON response with a status code of 400.

```
51 // get account details from Nordigen API
52 const accountDetailsResult = await axios
53   .get(
54     'https://ob.nordigen.com/api/v2/accounts/${accountFromBody.account_id}/details/',
55     {
56       headers: {
57         Authorization: `Bearer ${nordigenToken}`,
58       },
59     }
60   )
61   .then((res) => {
62     return res.data.account;
63   })
64   .catch((err) => console.log(err));
65
66 let fullAccountDoc = { ...accountFromBody, accountDetailsResult };
67
68 // call dao method to add new account
69 const accountResponse = await AccountsDAO.addAccount(
70   fullAccountDoc,
71   user
72 );
```

Figure 47 Getting account details

If no errors have occurred, the function continues to compile. A request is made to the Nordigen API to get the details of the account. The account ID is passed in the URL and the access token is passed in as a header. This result is then stored in a variable as an object. This variable is then combined with the previous variable (accountFromBody) where the JavaScript spread operator is used (...) and is combined into a new variable called 'fullAccountDoc'. A constant variable is created to store the response where the 'addAccount' method is called on line 69. This function takes in the account document and user object.

```
19 static async addAccount(accountDoc, user) {
20   try {
21     const updatedAccountDoc = {
22       name: user.name,
23       email: user.email,
24       account_id: accountDoc.account_id,
25       bank_name: accountDoc.bank_name,
26       iban: accountDoc.accountDetailsResult.iban,
27       currency: accountDoc.accountDetailsResult.currency,
28       owner_name: accountDoc.accountDetailsResult.ownerName,
29       account_name: accountDoc.accountDetailsResult.name,
30       product: accountDoc.accountDetailsResult.product,
31     };
32
33     return await accounts.insertOne(updatedAccountDoc);
34   } catch (e) {
35     console.error('Unable to post account: ${e}');
36     return { error: e };
37   }
38 }
```

Figure 48 Accounts DAO - Add account

The 'addAccount' function in the accounts DAO class is responsible for inserting the document into the accounts collection in the database. A constant variable is defined as 'updatedAccountDoc' which is an object containing the data to be inserted as a document. An 'insertOne' action is then performed on the accounts collection with the document passed in. This is then returned back to the controller class.

```
74 const updatedAccounts = await UsersDAO.getAccountsByEmail(user.email);
75
76 res.json({ status: "success", accounts: updatedAccounts.accounts });
77 } catch (error) {
78   console.error(error);
79   res.status(500).json({ e });
80 }
81 }
```

Figure 49 Returning response with accounts

A list of the users accounts is then retrieved by calling the 'getAccountsByEmail' function in the user DAO class. This array is then passed into a JSON response which indicates is a successful request and the completion of the process. An example of a successful request and response is displayed in Figure 50.

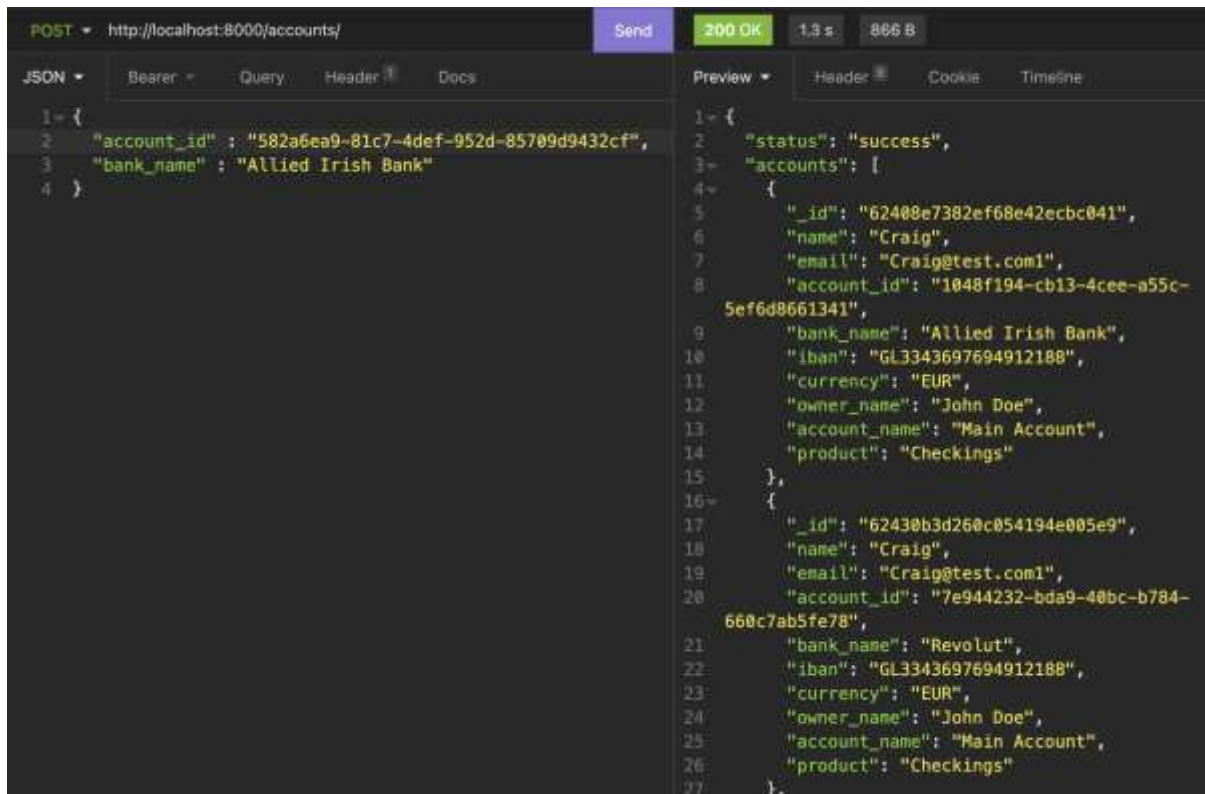


Figure 50 Successful request

5.4.7 Retrieving Accounts

A request can be made to retrieve the accounts for any specific user. A GET request is made to the `/accounts` endpoint and the `apiGetAccounts` function is called, which is previously shown in Figure 44.

```
83 static async apiGetAccounts(req, res, next) {
84   try {
85     const userJwt = req.get("Authorization").slice("Bearer ".length);
86     const user = await User.decoded(userJwt);
87     var { error } = user;
88     if (error) {
89       res.status(401).json({ error });
90       return;
91     }
92
93     const { accountsList, totalNumAccounts } =
94       await AccountsDAO.getAccountsByEmail(user.email);
95
96     let response = {
97       accounts: accountsList,
98       total_results: totalNumAccounts,
99     };
100
101     res.json(response);
102   } catch (error) {
103     console.log(error);
104     res.json(error);
105   }
106 }
```

Figure 51 Accounts controller - get accounts

An authentication check is made again as the user must be authenticated to make the request. The user object is then stored in a variable. The 'getAccountsByEmail' function is then called from the accounts DAO class and the users email is passed into the function. A destructuring assignment takes place on line 93 above, where the variables 'accountsList' and 'totalNumAccounts' are unpacked from the response from the DAO function, and stored into their own distinct variables. These variables are then used to create an object, 'response'. This response is then returned to the user in JSON format.

```
40     static async getAccountsByEmail(  
41         email,  
42         page = 0,  
43         accountsPerPage = 20,  
44         query = {}  
45     ) {  
46         let cursor;  
47         try {  
48             cursor = await accounts.find({ email: email });  
49         } catch (error) {  
50             console.error('Unable to issue find command, ${error}');  
51             return { accountsList: [], totalNumAccounts: 0 };  
52         }  
53  
54         const displayCursor = cursor  
55             .skip(accountsPerPage * page)  
56             .limit(accountsPerPage);  
57  
58         try {  
59             const accountsList = await displayCursor.toArray();  
60  
61             return { accountsList };  
62         } catch (error) {  
63             console.error(  
64                 'Unable to convert cursor to array or problem counting documents, ${error}'  
65             );  
66             return { accountsList: [], totalNumAccounts: 0 };  
67         }  
68     }
```

Figure 52 Accounts DAO – getAccountsByEmail

For the function in Figure 52, a cursor object is created, which is used to perform the 'find' query on the accounts collection. This code which is on line 48 tries to find the documents in the accounts collection where the email matches the email that was passed into the function. The lines from 54 to 56 provides the possibility for pagination, by only allowing a maximum of 20 accounts per page in the response. A try-catch block is then executed to transform the cursor into an array and store it in a variable. This variable is then returned to the controller.

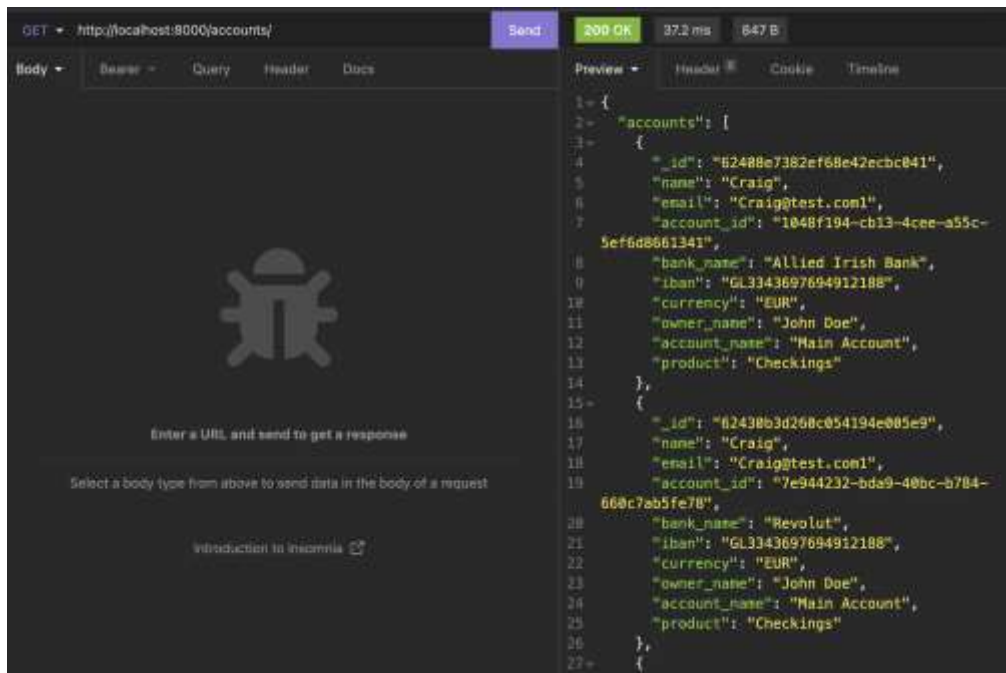


Figure 53 Successful GET accounts response

5.5 Frontend

The frontend application is the application that the end-users interact with. For the frontend of the application, it was developed using React Native. React Native is an open-source framework for creating mobile applications. It uses the same framework as React however for creating mobile applications native components are used rather than traditional web components.

To implement the design of the application, a library called 'NativeBase' was used. NativeBase is an accessible utility-first component library that is used to help developers build consistent UI across iOS, Android, and Web. This framework provides an abundance of native components such as buttons, alerts, boxes, styled form inputs etc. Each component supports various utility props which helps the developers style the components to their liking. Examples of these utility props would be padding, margin, background colour, size etc. The use of this library significantly speeds up the development process as building with React Native from scratch can become a tedious process with the requirement of certain steps like styling, adding interactions, responsiveness, accessibility etc. This is a similar library to popular frontend frameworks for the web such as Bootstrap and Tailwind CSS. NativeBase ships with their own colour palette for styling each component and provides the ability to customize the default colour palette and each component.



Figure 54 NativeBase themes - <https://www.npmjs.com/package/native-base>

5.5.1 Project Dependencies

```
package.json > ...
1
2 "dependencies": {
3   "expo-google-fonts/montserrat": "^0.2.2",
4   "expo/vector-icons": "^12.0.5",
5   "@react-native-async-storage/async-storage": "^1.15.17",
6   "@react-navigation/bottom-tabs": "^6.2.0",
7   "@react-navigation/native": "^6.0.6",
8   "@react-navigation/native-stack": "^6.2.5",
9   "@reduxjs/toolkit": "^1.7.1",
10  "async-storage": "^0.1.0",
11  "axios": "^0.25.0",
12  "expo": "^44.0.0",
13  "expo-app-loading": "^1.3.0",
14  "expo-font": "~10.0.4",
15  "expo-web-browser": "^10.1.0",
16  "moment": "^2.29.1",
17  "native-base": "^3.3.4",
18  "react": "17.0.1",
19  "react-dom": "17.0.1",
20  "react-hook-form": "^7.27.1",
21  "react-native": "0.64.3",
22  "react-native-dotenv": "^3.3.1",
23  "react-native-keyboard-aware-scroll-view": "^0.9.5",
24  "react-native-progress-circle": "^2.1.0",
25  "react-native-safe-area-context": "3.3.2",
26  "react-native-screens": "~3.10.1",
27  "react-native-svg": "^12.1.1",
28  "react-native-svg-transformer": "^1.0.0",
29  "react-native-web": "0.17.1",
30  "react-redux": "^7.2.6",
31  "redux": "^4.1.2"
32 }
```



```
32 },
33 "devDependencies": {
34   "@babel/core": "^7.12.9"
35 },
36   > Debug
37 "scripts": {
38   "start": "expo start",
39   "android": "expo start --android",
40   "ios": "expo start --ios",
41   "web": "expo start --web"
42 },
43 "version": "1.0.0",
44 "private": true,
45 "name": "majorprojectapp"
46 }
```

Figure 55 Frontend dependencies

To start the application, the React Native project comes with a 'package.json' file that is written in JSON. This file contains the dependencies that are required to install to develop the project, for example the react package comes with all the tools provided with React. This file also contains certain scripts that can be run in the terminal within the root of the project. These scripts run the expo server and allows the developer to connect to their virtual device.

5.5.2 Parent Component

```
App.js > -
1  import { NavigationContainer } from "@react-navigation/native";
2
3  // Redux
4  import { configureStore } from "@reduxjs/toolkit";
5  import { Provider } from "react-redux";
6  import authReducer from "../features/auth";
7  import userReducer from "../features/user";
8
9  // components
10 import Routes from "../routes/Routes";
11 import CustomThemeContainer from "../components/CustomThemeContainer";
12
13 const store = configureStore({
14   reducer: {
15     auth: authReducer,
16     user: userReducer,
17   },
18 });
19
20 export default function App() {
21   return (
22     <CustomThemeContainer>
23       <Provider store={store}>
24         <NavigationContainer>
25           <Routes />
26         </NavigationContainer>
27       </Provider>
28     </CustomThemeContainer>
29   );
30 }
```

Figure 56 Parent - App component

For every React Native project, it comes with an 'App.js' file that acts as the entry point for the application and it is the single component that is rendered and displayed to the user. Within the App.js file, the rest of the components are stored such as the various screens in the app. Within this file, the NativeBase, Redux and React Navigation packages were also

implemented. In order to implement a custom theme for the application, a custom component was created called the 'CustomThemeContainer' and inside this component, any fonts and colours etc. that might be used throughout the app can be initialized. This component simply returns a custom provider component provided by NativeBase, where the theme object is passed in as a prop and the children props are returned within this component:

```
13 const CustomThemeContainer = (props) => {  
14   // customizing native base theme  
15   > const theme = extendTheme({  
44     });  
45  
46   let [fontsLoaded] = useFonts({  
47     Montserrat_100Thin,  
48     Montserrat_300Light,  
49     Montserrat_400Regular,  
50     Montserrat_500Medium,  
51     Montserrat_600SemiBold,  
52     Montserrat_700Bold,  
53   });  
54   if (!fontsLoaded) {  
55     return <AppLoading />;  
56   }  
57  
58   return (  
59     <NativeBaseProvider theme={theme}>{props.children}</NativeBaseProvider>  
60   );  
61 };  
62  
63 export default CustomThemeContainer;
```

Figure 57 Custom theme container

In this case, the rest of the components from the App.js file are rendered inside the 'custom theme' component. This allows the custom fonts and colours to be used throughout the application using NativeBase components.

The next component in the App.js file is the 'Provider' which is part of the react-redux library. Redux is responsible for handling state management throughout the app. The Provider component takes in a store prop which is an object that contains various reducers. A reducer is a function that takes the current state and an action as arguments, and returns a new state value as the result. For example, if the application requires an authentication token to be used throughout the app, a reducer can be used such as an 'auth reducer' that contains a function that takes in the arguments (auth token) and the function then returns the updated state. This state can then be accessed in any component in the app – no matter how far down the component tree it may be. This makes the structure of the app and its variables cleaner, as certain variables can be stored globally, rather than being passed through multiple components as props.

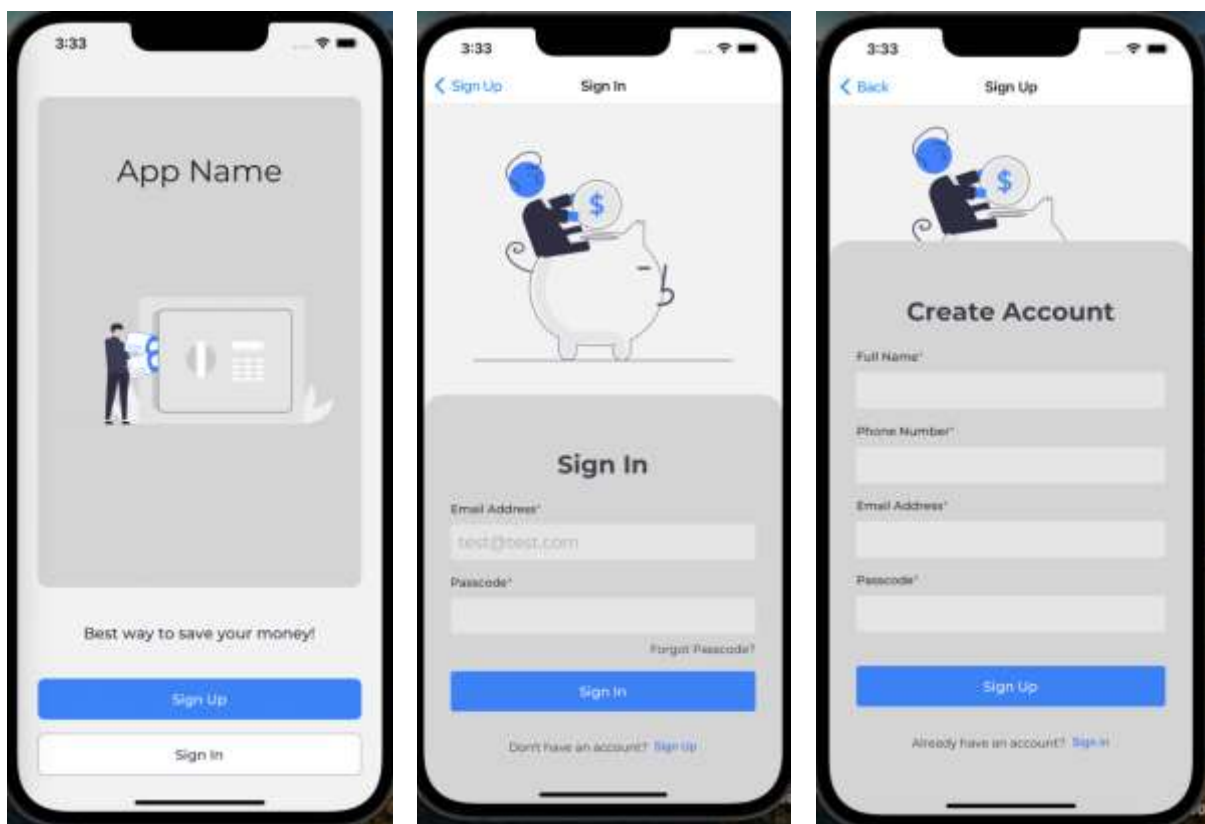
The next component in the App component tree is the 'Navigation Container' which is a component provided by the react navigation library. This is a parent component that wraps around the various routes/screens for the app. This allows users to navigate to different screens in the app. Inside this component, a custom component called 'Routes' has been created in order to tidy up the main App.js file. The Routes component returns the screens stack for the application. An example of this screen stack is shown in Figure 58.

```
138 <Stack.Screen
139   // options={{ headerShown: false }}
140   name="Register"
141   component={Register}
142   options={{ title: "Sign Up" }}
143 />
144 <Stack.Screen
145   // options={{ headerShown: false }}
146   name="Login"
147   component={Login}
148   options={{ title: "Sign In" }}
149 />
```

Figure 58 Screens Stack

5.5.3 Authentication Screens

In the initial sprint, the authorization screens began being developed such as the welcome screen, login and register screens and the home screen. The UI was developed first with some hard-coded data and then the API and backend were gradually integrated into the app, with the user's details getting stored in the database when they registered and receiving an authentication token when they logged in. For the purpose of testing the application, dummy data was used with the help of Nordigen's sandbox data. This essentially acts as a fake bank and provides access to a list of fake transactions that are used for testing in the app.



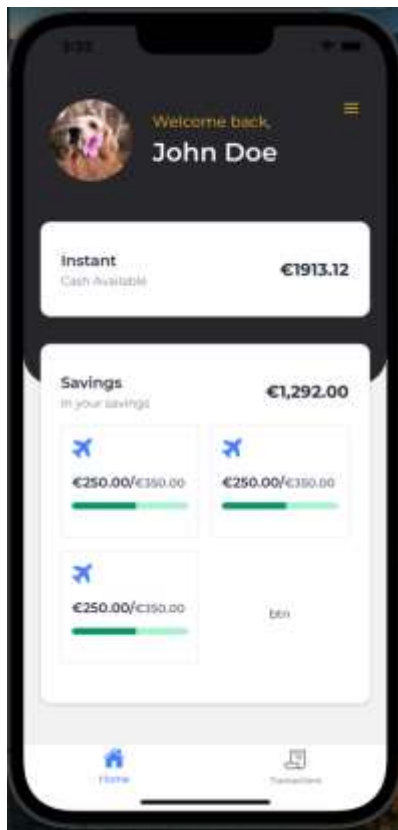


Figure 59 Development of First Screens

One issue that was faced when developing these screens was that when the user tapped on some of the inputs, the keyboard was covering the input which hindered the view and experience of the user as they could not see what they were typing. As shown in Figure 61 below, the user is attempting to input their password however the keyboard covers the entire input field. This issue was solved by using a package called react native keyboard aware scroll view. This provides a container where the form can be inserted and when the user taps on an input field it checks to see if the keyboard will cover the input and if it is, the screen automatically scrolls down to show the input field (Figure 60).

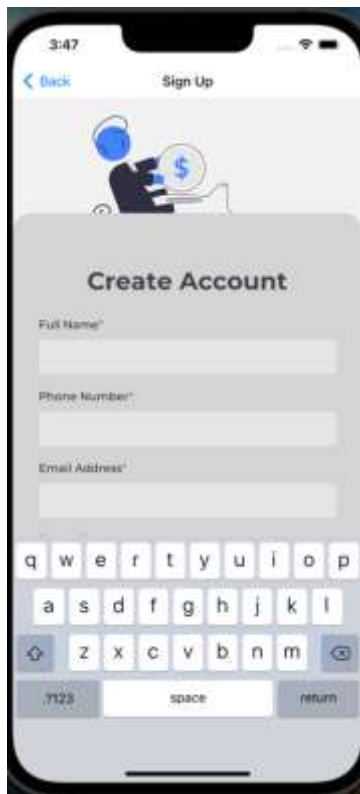


Figure 61 Keyboard blocking password input



Figure 60 Keyboard aware scroll view

The main major implementation issue faced was the task of getting familiar with building an app with the native components rather than the traditional web component blocks. However, once this type of development is learned, the process of developing the app became fluent and quicker.

For the frontend handling for the user authentication system, it connected to a custom Express backend application and also made requests to the Nordigen API. When the user makes request to login or register, requests are made to the backend which executes a certain function that then communicates with the MongoDB database.

```

24  const Register = ({ navigation }) => {
25    const dispatch = useDispatch();
26    const [responseErrors, setResponseErrors] = useState(null);
27
28    const {
29      control,
30      handleSubmit,
31      formState: { errors },
32    } = useForm({
33      defaultValues: {
34        name: "",
35        number: "",
36        email: "",
37        passcode: "",
38      },
39    });

```

Figure 62 Register Component

In the component above, the Register screen is displayed to the user, which is a form with various fields such as name, passcode, email etc. To handle errors on the frontend, a package called 'react-hook-form' was introduced. This allows the developer to initialize the field names and default values and for each field, it is possible to set up certain rules – for example: the name field must be a minimum length of 3 characters. This object is initialized at the top of the component, as shown in lines 28 to 39 above. On line 25, the dispatch object is set up which is a redux object that is eventually used to make updates to the state. On line 26 a React useState variable is set up and initialized as null. This variable is used to store errors returned from the response sent back from the API if there is any.

Once the user presses the sign up button and there are no errors on the frontend, the register() function is run (Figure 63).

```

41  const register = (data) => {
42    // store user in database
43    axios
44      .post("http://localhost:8000/users/register", {
45        name: data.name,
46        email: data.email,
47        passcode: data.passcode,
48        number: data.number,
49      })
50      .then((res) => {
51        console.log(res.data);
52        dispatch(setAuthToken({ authToken: res.data.auth_token }));
53        dispatch(
54          setUser({ name: data.name, email: data.email, number: data.number })
55        );
56
57        // create a nordigen access token
58        // navigate to banks list screen
59        axios
60          .post("https://ob.nordigen.com/api/v2/token/new/", {
61            secret_id: SECRET_ID,
62            secret_key: SECRET_KEY,
63          })
64          .then((res) => {
65            dispatch(setNordigenToken({ nordigenToken: res.data.access }));
66            console.log(res.data.access);
67            navigation.navigate("BanksList");
68          })
69          .catch((err) => console.log(err));
70      })
71      .catch((err) => {
72        if (err) {
73          setResponseErrors(
74            "Error - User with this email already exists in the database."
75          );
76        }
77      });
78  };

```

You, a month ago • Register sends request to backend

Figure 63 Register function

This function takes in data as a parameter which is the data filled in from the form fields. An axios request is then sent to the backend to the /register route and the user object is passed in as the request body. Axios is promised based and runs the '.then' block if the request has been fulfilled, or else the '.catch' block is run to handle any errors. If the request has been successful, a user gets stored in the database and a response is sent back with the user data and an authentication token. The dispatch() object is then used to change the state in the Redux store which is shown on lines 52 to 55 – where an authentication token and a user

object are being initialized as they can be used later throughout the application. An axios request is then sent to the Nordigen API to create a new token that will be used to access their list of banks, accounts etc. This request requires a secret ID and secret key which have been initialized in a .env file in the project route that contains environment variables. If this request has been successful, an access token is returned, which has been set in the global store on line 65. The user is then navigated to the banks list screen where they select their bank.

5.5.4 Linking Bank Accounts

```
19 const BanksList = ({ navigation }) => {  
20   const { nordigenToken } = useSelector((state) => state.auth.nordigenToken);  
21   const [banksList, setBanksList] = useState([]);  
22  
23   useEffect(() => {  
24     getBanks();  
25   }, [banksList]);  
26  
27   const getBanks = () => {  
28     axios  
29       .get("https://ob.nordigen.com/api/v2/institutions/?country=ie", {  
30         headers: {  
31           Authorization: `Bearer ${nordigenToken}`,  
32         },  
33       })  
34       .then((res) => {  
35         setBanksList(res.data);  
36       })  
37       .catch((err) => console.log(err));  
38   };  
}
```

Figure 64 Banks List Component

The banks list component is responsible for displaying a list of banks to the user so that they can choose which one to link to their account. At the top of the component an empty array is initialized as a variable called 'banksList' on line 21. The 'nordigenToken' is also retrieved from the global state by using the Redux useSelector hook, which is required in the header for each request to the Nordigen API. On line 23, a useEffect hook is used which is a hook provided from React that allows side effects to be performed in components such as fetching data, updating the DOM. In this example the getBanks() function is run when the component renders, which makes an axios request to the institutions endpoint. The Nordigen token is passed in as the header and in the response, the banks list variable is set to an array from the data returned in the response.


```

56      {banksList ? (
57        banksList.map((bank, index) => {
58          <Box key={index}>
59            <TouchableOpacity
60              style={{
61                flexDirection: "row",
62                alignItems: "center",
63              }}
64              onPress={() => selectBank(bank)}
65            >
66              <Image
67                style={{ width: 48, height: 48 }}
68                source={{ uri: bank.logo }}
69              />
70              <Text mL="3">{bank.name}</Text>
71            </TouchableOpacity>
72            <Divider my="4" />
73          </Box>
74        })
75      ) : (
76        <View>
77          <Text>No banks found</Text>
78        </View>
79      )}

```

Figure 65 Banks List JSX

As shown above, JSX is rendered and a check is made to see if there is data in the 'banksList' variable. If there is each bank is rendered using the JavaScript map() function that renders each bank in the array. Each item is a box that contains an image and text, the item is wrapped in a TouchableOpacity which is a component provided by React Native that allows the user to press on items (Figure 66). When a user presses on a bank item, the selectBank() function is executed and the bank item is passed in.

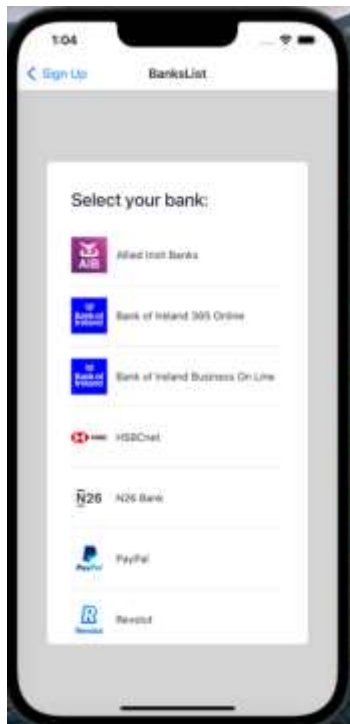


Figure 66 List of banks


```
40   const selectBank = (props) => {  
41     console.log("selected", props);  
42     navigation.navigate("UserAgreement", {  
43       name: props.name,  
44       id: props.id,  
45     });  
46   };
```

Figure 67 selectBank function

The selectBank function navigates the user to the user agreement screen and passes props to the next screen. These props are the bank name and ID which will be used to access the bank information on the next screen.

The next step in the process of linking accounts from the Nordigen API is for the user to complete the user agreement (Figure 68), which grants the access of their bank transactions and accounts. Once the user presses the agree button, the 'createAgreement()' function is executed.

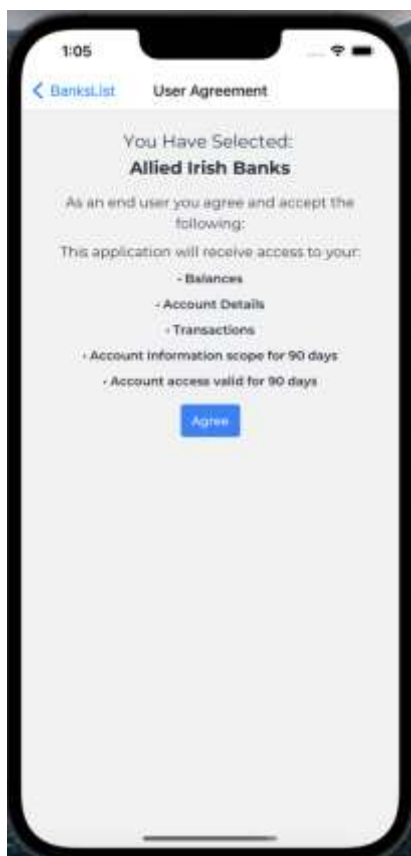


Figure 68 User Agreement screen

```

50  const createAgreement = () => {
51
52    axios
53      .post(
54        "https://ob.nordigen.com/api/v2/agreements/enduser/",
55        {
56          // institution_id: route.params.id,
57          institution_id: "SANDBOXFINANCE_SFIN0000", // test bank
58        },
59        {
60          headers: {
61            Authorization: `Bearer ${nordigenToken}`,
62          },
63        }
64      )
65      .then(() => {
66        axios
67          .post(
68            "https://ob.nordigen.com/api/v2/requisitions/",
69            {
70              redirect: "https://dreamy-ptolemy-483fe8.netlify.app/",
71              // institution_id: route.params.id,
72              institution_id: "SANDBOXFINANCE_SFIN0000", // test bank
73            },
74            {
75              headers: {
76                Authorization: `Bearer ${nordigenToken}`,
77              },
78            }
79          )
80          .then((res) => {
81            console.log(res.data.id);
82            setAgreementId(res.data.id);
83            let result = WebBrowser.openBrowserAsync(
84              res.data.link
85              // "https://ob.nordigen.com/psd2/start/2be17da5-2b1e-4881-91f9-f4b22b570096/SANDBOXFINANCE_SFIN0000"
86            );
87            setProcessComplete(true);
88          })
89          .catch((err) => console.log(err));
90      })
91      .catch((err) => console.log(err));
92  };

```

Figure 69 createAgreement function

This create agreement function makes an axios request to Nordigen's agreements endpoint, the institution ID is required in the request body, which is the bank ID that is passed in from the previous screen. In this example and for testing purposes, the Nordigen sandbox finance bank ID is passed in on line 56. Once this request has been completed, another axios request is sent to the requisition's endpoint, which is used to pass in the bank ID and a redirect link where the user will be redirected once they have completed the process of linking their account through Nordigen. In the response of this request, a link is returned in the response that is used to begin the process of linking the bank institute through Nordigen. The web browser of the phone will open up for the user when line 82 is executed. This is possible by using the openBrowserAsync function that is provided by the 'expo-web-browser' library. The variable 'processComplete()' is then set to true in order to update the state of the component and display a message to the user telling them to refresh the screen in order to proceed (Figure 70).

```

130 [processComplete ? (
131   <>
132   <Heading>Please Refresh The Screen</Heading>
133   <Image
134     source={{
135       uri: "https://docs.microsoft.com/en-us/windows/apps/design/controls/images/pull-to-refresh.gif",
136     }}
137     w="300"
138     h="400"
139     alt="gif"
140   />
141   </>
142 ) : (
143   <></>
144 )

```

Figure 70 Conditionally rendering

```

27 // on refresh screen
28 const onRefresh = useCallback(() => {
29   setRefreshing(true);
30   wait(1000).then(() => {
31     axios
32       .get('https://ob.nordigen.com/api/v2/requisitions/${agreementId}/', {
33         headers: {
34           Authorization: `Bearer ${nordigenToken}`,
35         },
36       })
37       .then((res) => {
38         navigation.navigate("ListAccounts", {
39           accounts: res.data.accounts,
40         });
41       })
42       .catch((err) => console.log(err));
43     setRefreshing(false);
44   });
45 }, [agreementId]);

```

Figure 71 onRefresh function

When the user refreshes the screen, the 'onRefresh' function is executed. This makes an axios request to the requisitions endpoint with the 'agreementId' passed in that was retrieved from the previous request. The purpose of this request is to get a list of the accounts that the user can link to, which is then passed in to the next screen – List Accounts screen. The accounts array is passed in as props to the list accounts screen.

```
33 // Get account details
34 useEffect(() => {
35   checkUserExistingAccounts();
36 }, []);
37
38 const checkUserExistingAccounts = () => {
39   axios
40     .get("http://localhost:8000/accounts/", {
41       headers: {
42         Authorization: `Bearer ${authToken}`,
43       },
44     })
45     .then((res) => {
46       res.data.accounts.forEach((account) => {
47         setExistingAccounts(account.account_id);
48       });
49       getAccounts();
50     })
51     .catch((err) => {
52       getAccounts();
53       console.log("error", err);
54     });
55   };

```

Figure 72 List Accounts functions

The purpose of the list accounts screen in the figure above is to allow the user to select the account they wish to link. To prevent the user from being able to link the same account more than once, a function 'checkUserExistingAccounts()' was implemented. This function is run in the use effect hook once the component has rendered and, in this function, a request is made to the Express backend to get the users existing accounts. This returns an array with the account IDs that relates to the user, and for each account in the response, a variable called 'existingAccounts' is set to the account ID. The function 'getAccounts()' is then called (Figure 73).

```

57  const getAccounts = () => {
58    // account IDs from previous route
59    setAccountIds(route.params.accounts);
60
61    if (!existingAccounts.includes(accountIds[0])) {
62      axios
63        .get(
64          `https://ob.nordigen.com/api/v2/accounts/${accountIds[0]}/details/`,
65          {
66            headers: {
67              Authorization: `Bearer ${nordigenToken}`,
68            },
69          }
70        )
71        .then((res) => {
72          setAccountOne(res.data.account);
73          console.log(accountOne);
74        })
75        .catch((err) => console.log(err));
76    } else {
77      setAccountOne(null);
78    }

```

Figure 73 getAccounts function

Once this function is called, the account IDs from the previous screen is stored in the state variable 'accountIds'. A check is then made to see if the user already has the account linked, this is handled by checking if the existing accounts array contains the accountId at index 0, and if it doesn't, the axios request is made to get the account details – which is then stored in the 'accountOne' variable on line 72, otherwise the 'accountOne' is set to null. This process above is then repeated for the second value from the account IDs array.

```

176  {accountOne ? (
177    <Pressable onPress={() => selectAccount(0)}>
178      <Box
179        mx="8"
180        borderWidth="1"
181        borderColor="coolGray.300"
182        rounded="xl"
183        px="6"
184        py="4"
185        bg="white"
186      >
187        <Flex direction="row" alignItems="center" justify="space-between">
188          <View>
189            <Text
190              color="coolGray.500"
191              fontWeight={500}
192              textTransform="uppercase"
193            >
194              {accountOne.name}
195            </Text>
196            <Text fontSize="xl" fontWeight={600}>
197              {accountOne.ownerName}
198            </Text>
199          </View>
200          <Icons name="enter-outline" size={24} />
201        </Flex>
202        <Text mt="3">Currency: {accountOne.currency}</Text>
203        <Text>Product: {accountOne.product}</Text>
204      </Box>
205    </Pressable>
206  ) : (
207    <</>
208  )}

```

Figure 74 Conditionally render account

Each account is then conditionally rendered in the component. As displayed in line 176, a check is made to see if 'accountOne' is not null. If it does exist, the account is rendered displaying its information and is wrapped around a 'pressable' component (Figure 75), and when the user presses on the account, the 'selectAccount' function is called.

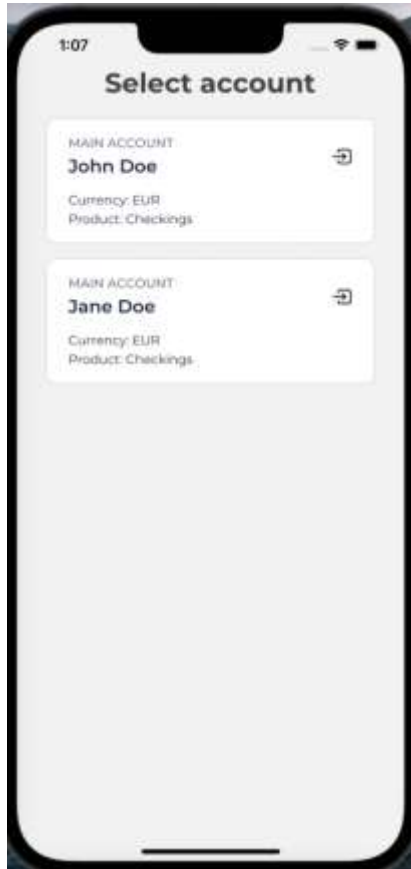


Figure 75 Select Account Screen

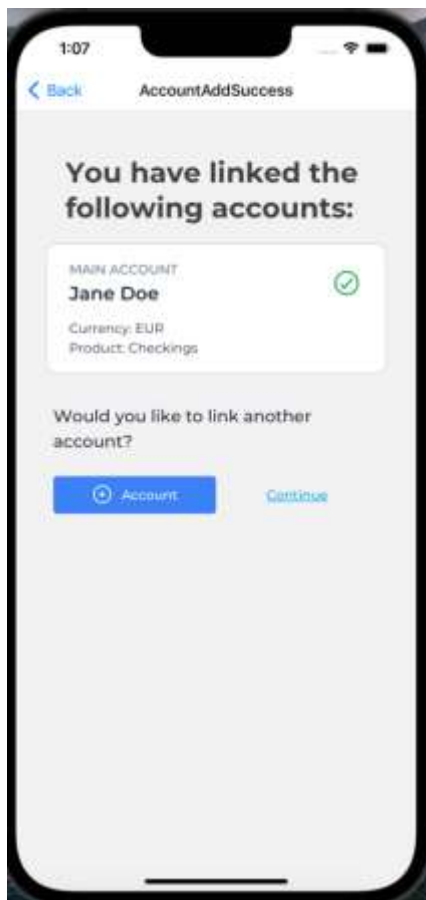
```
101   const selectAccount = (accountNumber) => {
102     axios
103       .post(
104         "http://localhost:8000/accounts/",
105         {
106           account_id: accountIds[accountNumber],
107         },
108         {
109           headers: {
110             Authorization: `Bearer ${authToken}`,
111           },
112         }
113       )
114       .then((res) => {
115         //last element of array
116         dispatch(
117           getUserAccount({
118             accountID:
119               res.data.accounts[res.data.accounts.length - 1].account_id,
120           })
121         );
122
123         // navigate to add successfull screen with the account ID passed
124         navigation.navigate("AccountAddSuccess", {
125           accountID: res.data.accounts[res.data.accounts.length - 1].account_id,
126         });
127       })
128       .catch((err) => console.log(err));
129   };
```

Figure 76 selectAccount function

This function takes in the account number that was selected, in the above example in Figure 74, the number 0 is passed into the function as a parameter, this represents the index of the value in the accountIDs array – as shown in line 106. A request is sent to the Express backend with the account id, and this gets stored in the database as an account document – that contains the account ID and user's email. In the response of this request, each of the user's accounts are returned as an array. To access the most recent account that was added, the code on line 119 was used. This account ID is then stored in the global store as a variable in case it needs to be accessed later in the application. The user is then navigated to the 'AccountAddSuccess' screen with the accountID passed in as a parameter.


```
17 const AccountAddSuccess = ({ navigation, route }) => {
18   const [accounts, setAccounts] = useState([]);
19   const { nordigenToken } = useSelector((state) => state.auth.nordigenToken);
20   const userAccountID = useSelector((state) => state.user.accountID.accountID);
21
22   useEffect(() => {
23     getAccounts();
24   }, []);
25
26   const getAccounts = () => {
27     axios
28       .get(
29         `https://ob.nordigen.com/api/v2/accounts/${route.params.accountID}/details/`,
30         {
31           headers: {
32             Authorization: `Bearer ${nordigenToken}`,
33           },
34         }
35       )
36       .then((res) => {
37         setAccounts(res.data.account);
38       })
39       .catch((err) => console.log(err));
40   };
41 }
```

Figure 77 Account add success screen



This screen gives feedback to the user that they have successfully linked their bank account to the app. When the component is rendered, the 'getAccounts()' function is called in the

useEffect hook. This makes a request to the Nordigen APIs account details endpoint, and passes in the account ID from the previous screen. This returns an object containing the details about the account such as the name, type, currency etc. This is then displayed to the user as the account that they have linked. The user is then offered a choice – if they would like to link another account or else continue. If the user chooses to link another account, they are taken through the process again beginning at the list of banks screen. If the user presses on the continue button, they are taken to the home screen.

5.5.5 Home Screen – User Index

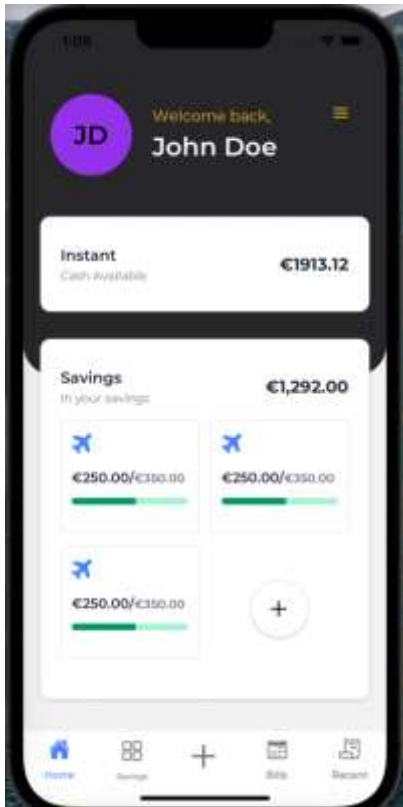


Figure 78 Home Screen

The user index screen (Figure 78) is where users are brought when they initially register or login to their account. This screen shows them a welcome back message, and displays their currently selected bank accounts total available balance, and a list of their saving goals underneath. There is also a hamburger icon that when selected, shows a modal popup that provides various links such as a link to the user's profile and the logout button. From this screen, the user can navigate to the tabs on the bottom tab navigator. These tabs include the savings index, bills index and recent transactions. There is also a '+' button that takes the user to a screen that prompts them to add a new saving goal, bill or link a new account.

The user index component contains different functions to perform actions such as getting the account balance and logging the user out.

```

27 const Index = ({ navigation, route }) => {
28   // State
29   const userAccountID = useSelector((state) => state.user.accountID.accountID);
30   const user = useSelector((state) => state.user.value);
31   const nordigenToken = useSelector(
32     (state) => state.auth.nordigenToken.nordigenToken
33   );
34   const authToken = useSelector((state) => state.auth.authToken);
35   const [accountBalance, setAccountBalance] = useState("");
36   const [showModal, setShowModal] = useState(false);
37
38   // get account balance
39   useEffect(() => {
40     getAccountBalance();
41   }, []);
42
43   const getAccountBalance = () => {
44     axios
45       .get(
46         `https://ob.nordigen.com/api/v2/accounts/${userAccountID}/balances/`,
47         {
48           headers: {
49             Authorization: `Bearer ${nordigenToken}`,
50           },
51         }
52       )
53       .then((res) => {
54         console.log(res.data.balances[0]);
55         setAccountBalance(res.data.balances[0]);
56       })
57       .catch((err) => console.log(err));
58   };

```

Figure 79 User Index component

As shown in Figure 79, there are state variables initialized in the component such as the account ID, user object, auth token etc. as these variables are required within this component. When the component renders, the 'getAccountBalance()' method executes, which makes a request to the account balances endpoint from Nordigen with the account ID passed in. This returns an object with details of the account such as the currency and total amount.

```

60 const logoutUser = () => {
61   axios
62     .post(
63       "http://localhost:8000/users/logout",
64       {},
65       {
66         headers: {
67           Authorization: `Bearer ${authToken.authToken}`,
68         },
69       }
70     )
71     .then((res) => {
72       navigation.navigate("Welcome");
73       console.log(res);
74     })
75     .catch((err) => console.log(err));
76 };

```

Figure 80 Logout function

Another function within the user index screen is the 'logoutUser()' function, which makes a POST request to the Express application and the user's authentication token is passed in as a header. Within the backend, the session is removed from the database and a 200 OK response is returned to the frontend with a success message. The user is then navigated to the Welcome screen on line 72.

5.5.6 Savings Screen – Index

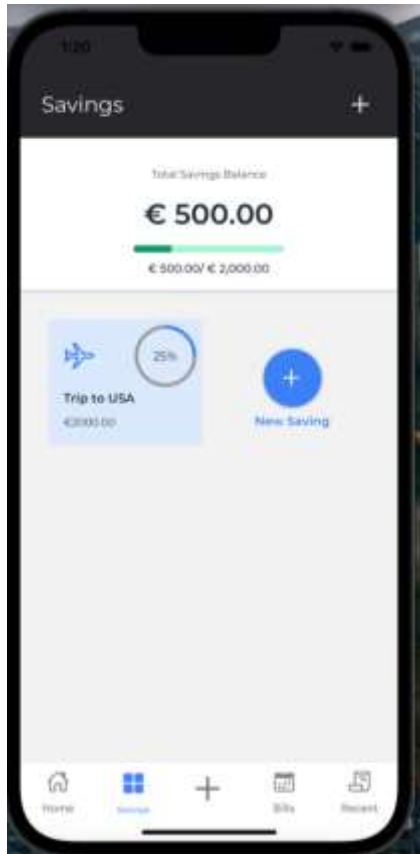


Figure 81 Savings index screen

The savings index screen is where the user can view their list of saving goals, add a new goal and it displays the current savings total amount, the total saving goals amount combined and a progress bar to give the user feedback on how much progress they have made on the goal. Within this component, there are multiple calculations and features implemented.

```
48  const getTotalSavingGoal = (array) => {  
49    let sum = 0;  
50    for (var i = 0; i < array.length; i++) sum += parseInt(array[i].amount);  
51    setInitialSavingsGoal(sum);  
52    setTotalSavingsGoal(currencyFormatter(sum));  
53  };  
54  
55  const getCurrentSavingsTotal = (array) => {  
56    let sum = 0;  
57    for (var i = 0; i < array.length; i++)  
58      sum += parseInt(array[i].current_amount);  
59    setInitialSavingsTotal(sum);  
60    setCurrentSavingsTotal(currencyFormatter(sum));  
61  };
```

Figure 82 Functions to calculate totals

In the above figure, two methods were implemented to calculate the total savings balance and the current total amount that the user has saved. The reason for this is to display to the user for example: "total saved = €699/€1000". In the 'getTotalSavingGoal()' function, an array is passed in and a variable 'sum' is initialized as 0. A for loop is then ran on the array and it loops over the array by the number of items in the array. An addition assignment (+=) is then ran on the sum variable which adds each of the 'amount' values from the array together and assigns it to the sum variable. The variable 'initialSavingsGoal' gets assigned the sum variable. Another assignment of a variable is made on line 52, where the 'totalSavingsGoal' is assigned to the same variable, however this time it is passed into another function called 'currencyFormatter()' which is described below. The reasoning behind separating these variables is because the total savings combined returns a long value with no punctuation such as €121344, this initial value is eventually used in a function to get a percentage value. The currency formatted value will return a value which would look like €121,344.00.

The 'getCurrentSavingsTotal()' function is similar to the one above, however different variables get assigned and the sum variable gets assigned to the total values of the 'current_amount' variables added together.

```
63  const currencyFormatter = (amount) => {  
64      if (!amount) return null;  
65      return amount.toFixed(2).replace(/\d(?=(\d{3})+\.)/g, "$&," );  
66  };
```

Figure 83 Currency formatter function

The currency formatter function is used to make the amount string more readable to the user. It takes in an amount as a parameter and returns a formatted value. An example of this would be currencyFormatter(1299) => €1,299.00.

```
68  const getPercentage = (partialValue, totalValue) => {  
69      return (100 * partialValue) / totalValue;  
70  };
```

Figure 84 Percentage function

The get percentage function was created to return a percentage of two values passed in as parameters. The partial and total values are passed in and a calculation is made to generate the percentage. An example use case of this function is to calculate the total completion percentage for a user's saving goal – if the goal is €1000 and they have saved €500 currently, therefore getPercentage(500,1000) => 50.

```
25  useEffect(() => {  
26    getSavings();  
27  });  
28  
29  const getSavings = () => {  
30    axios  
31      .get("http://localhost:8000/savings/", {  
32        headers: {  
33          Authorization: `Bearer ${authToken.authToken}`,  
34        },  
35      })  
36      .then((res) => {  
37        setSavingsList(res.data.savings);  
38        getTotalSavingGoal(savingsList);  
39        getCurrentSavingsTotal(savingsList);  
40        setSavingsStatus(  
41          getPercentage(initialSavingsTotal, initialSavingsGoal)  
42        );  
43      })  
44      .catch((err) => console.log(err));  
45  };  
46
```

Figure 85 Savings index - `getSavings()`

When the saving index screen has rendered, the function 'getSavings()' is called. This makes a request to the backend and a list of the saving goals that belong to a user are returned as an array. The variable 'savingsList' is set to the response data and then the functions from the figures above are called by passing in the array. These variables are then used in the return section of the component and are displayed to the user.

5.5.7 Savings Screen – Show

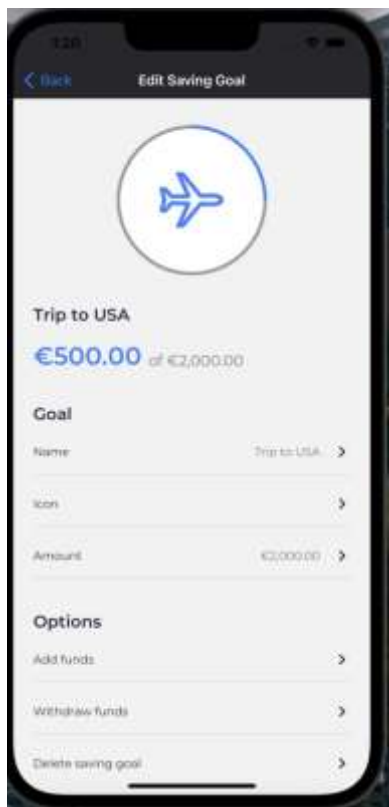


Figure 86 Savings show screen

When a user taps on a saving item, they are brought to the saving show screen (Figure 86). This is where the information of the saving item can be viewed. This screen is where the user should be able to update the saving's name, icon, amount etc. and perform certain actions for the saving like adding or withdrawing funds from the saving.

5.5.8 Transactions Screen – Index

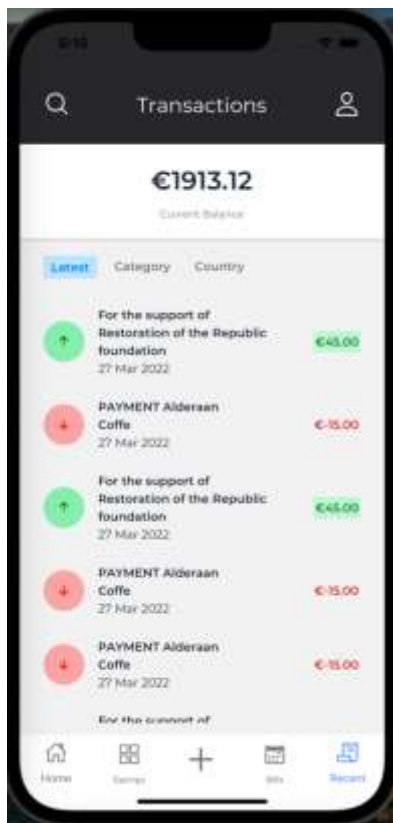


Figure 87 Transactions Index screen

The transactions index screen is where the user can go to view their most recent transactions for the current account they have selected. This screen makes use of a couple of endpoints provided by the Nordigen API.


```
17   useEffect(() => {
18     // get account transactions & balance
19     getTransactions();
20     getAccountBalance();
21   }, [userAccountID]);
22
23   const getTransactions = () => {
24     axios
25       .get(
26         `https://ob.nordigen.com/api/v2/accounts/${userAccountID}/transaction
27       {
28         headers: {
29           Authorization: `Bearer ${nordigenToken}`,
30         },
31       }
32     )
33     .then((res) => {
34       console.log(res.data.transactions.booked.slice(0, 5));
35       setAccountTransactions(res.data.transactions.booked.slice(0, 20));
36     })
37     .catch((err) => console.log(err));
38   };
```

Figure 88 Transactions index - useEffect hook & getTransactions

As shown in Figure 88, when the screen has rendered, two functions are called within the use effect hook – ‘getTransactions’ and ‘getAccountBalance’. Within the get transactions function, a request is made to the account transactions endpoint provided by Nordigen, where the account ID is passed in the URL. This account ID is stored in the applications Redux global store. The Nordigen access token is required as a header in the request, which is also stored as a global variable.

Upon success of this request, a list of recent transactions is returned in the response. As shown on line 35, the first 20 transactions are stored in a variable which is an array called ‘accountTransactions’. These transactions are then rendered on the screen.

```
40   const getAccountBalance = () => {
41     axios
42       .get(
43         `https://ob.nordigen.com/api/v2/accounts/${userAccountID}/balances/`,
44         {
45           headers: {
46             Authorization: `Bearer ${nordigenToken}`,
47           },
48         }
49       )
50       .then((res) => {
51         setAccountBalance(res.data.balances[0]);
52       })
53       .catch((err) => console.log(err));
54   };
```

Figure 89 getAccountBalance function

The user's account balance is retrieved from the Nordigen API balances endpoint. The balance is stored in a variable on line 51 and is then displayed at the top of the screen as shown in Figure 87.

```
142 {accountTransactions ? {
143   accountTransactions.map((transaction, index) => {
144     return (
145       <React.Fragment key={index}>
146         <Flex direction="row" justify="space-between" mb="6">
147           <Flex direction="row" alignItems="center">
148             <TransactionIcon
149               color={
150                 transaction.transactionAmount.amount.includes("-")
151                   ? "red.300"
152                   : "green.300"
153               }
154               direction={
155                 transaction.transactionAmount.amount.includes("-")
156                   ? "down"
157                   : "up"
158               }
159             />
160             <Box maxW="2/3">
161               <Text fontWeight={600}>
162                 {transaction.remittanceInformationUnstructured}
163               </Text>
```

Figure 90 Transactions loop

A check is made to see if the 'accountTransactions' variable contains any data and if it does, the array is displayed through a loop. A custom component is implemented (TransactionIcon) on line 148 to 159 which takes in two custom props, 'colour' and 'direction'. Conditional rendering is then performed to see if the transaction amount includes a '-' symbol, which indicates if it is an expense or not. If it does include a minus symbol, the red colour is passed into the component or else it is set to the green colour to indicate is income. For the direction prop, either the string "up" or "down" is passed in which determines which direction the arrow icon faces.

```

56   const TransactionIcon = ({ color, direction }) => {
57     return (
58       <Box
59         w="12"
60         h="12"
61         mr="4"
62         bg={color}
63         rounded="full"
64         alignItems="center"
65         justifyContent="center"
66       >
67         <Ionicons
68           name={`arrow-${direction}`}
69           size={18}
70           color={direction == "up" ? "green" : "red"}
71         />
72       </Box>
73     );
74   };

```

Figure 91 Transaction Icon component

```

162     <Text fontWeight={600}>
163       {transaction.remittanceInformationUnstructured}
164     </Text>
165     <Text>
166       {moment(transaction.bookingDate).format("D MMM YYYY")}
167     </Text>
168   </Box>
169 </Flex>
170 /* if transaction has - then it should be RED */
171 <Box
172   bg={
173     transaction.transactionAmount.amount.includes("-")
174     ? "transparent"
175     : "green.200"
176   }
177   p={0.5}
178   alignSelf="center"
179 >
180   <Text
181     fontWeight={600}
182     color={
183       transaction.transactionAmount.amount.includes("-")
184       ? "red.600"
185       : "green.700"
186     }
187   >
188     €{transaction.transactionAmount.amount}
189   </Text>
190 </Box>
191 </Flex>

```

Figure 92 Transactions Markup

For the rest of the transaction component, the date, details and amount are displayed. The JavaScript library moment.js is used to format the date, as shown on line 166. More

conditional rendering is performed to check if the transaction amount includes a minus symbol to determine if it is an expense or income. If the transaction is an expense, the text colour is red and if it is income, the text is displayed as green with a green box around it – which can be seen in Figure 87.

5.5.9 Switching Accounts

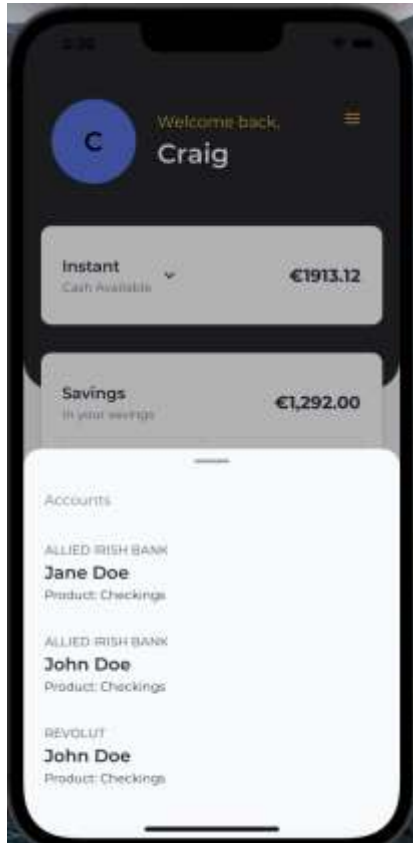


Figure 93 Account Selection

Various functionality was implemented to allow the user to switch between the bank accounts that they have linked. When a user presses on the arrow next to the 'Instant Cash Available' text in the box (Figure 93) an action sheet is displayed at the bottom of the screen with a list of the user's accounts. The action sheet is a component provided by the NativeBase framework.

```
55   const getAccounts = () => {  
56     axios  
57       .get(`http://localhost:8000/accounts`, {  
58         headers: {  
59           Authorization: `Bearer ${authToken.authToken}`,  
60         },  
61       })  
62       .then((res) => {  
63         setUserBankAccounts(res.data.accounts);  
64       })  
65       .catch((err) => console.log(err));  
66   };
```

Figure 94 getAccounts function

When the user index screen is rendered, the 'getAccounts' function is called (Figure 94). This makes a request to the Express backend to the accounts endpoint which returns an array with the user's linked accounts. This array is stored in a variable on line 63. This array is then rendered on the screen to allow the user to switch between accounts.

```
179   {userBankAccounts ? ||  
180     userBankAccounts.map((account, index) => {  
181       <ActionSheet.Item  
182         key={index}  
183         display="flex"  
184         onPress={() => {  
185           selectAccount(account.account_id);  
186           onClose();  
187         }}  
188       >  
189         <Text  
190           color="coolGray.500"  
191           fontWeight={500}  
192           textTransform="uppercase"  
193         >  
194           {account.bank_name}  
195         </Text>  
196         <Text fontSize="xl" fontWeight={600}>  
197           {account.owner_name}  
198         </Text>  
199         <Text>Product: {account.product}</Text>  
200       </ActionSheet.Item>  
201     )  
202   }
```

Figure 95 Account loop in Action Sheet

The bank accounts array is looped over and rendered on the screen, allowing the user to press on the account item and two functions are called – 'selectAccount' and 'onClose'. The on close function closes the action sheet, and for the select account function, the account ID is passed as a parameter (Figure 95).

```
154     const selectAccount = (props) => {  
155         dispatch(getUserAccount({ accountID: props }));  
156         getAccountBalance();  
157     };
```

Figure 96 Select Account function

When the select account function is called, the redux state is updated and the user's account ID is set to the ID that was passed into the function. The reason for this is the ID can then be used throughout the application, for example if the user's transactions for that account need to be retrieved. On line 156 the 'getAccountBalance' function is called to update the home screen and display the balance for the selected account. This function can be seen in Figure 89.

5.5.10 Add Item Screen

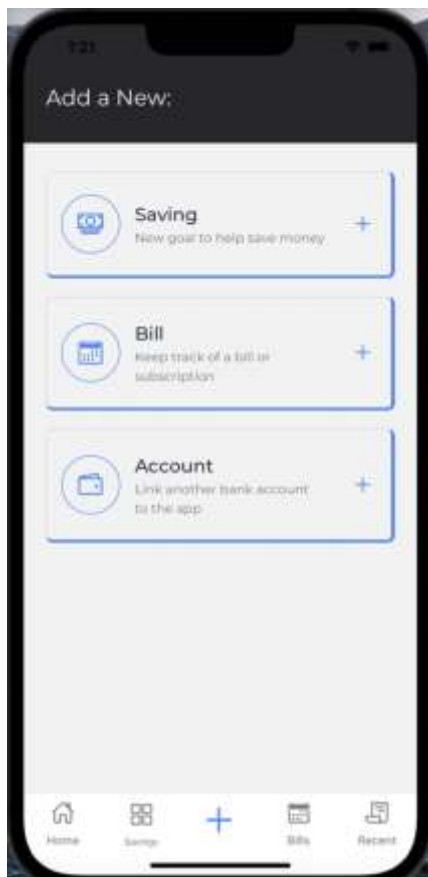


Figure 97 Add item Screen

When the user presses on the middle button on the bottom tab navigator, which is a plus button, they are brought to the screen shown in Figure 97. From here the option is provided to the user to add a new saving, bill or to link a new account.

6 Testing

6.1 Introduction

This chapter describes the tests that were carried out for the application. Testing the application includes the process of evaluating that the software does what it is supposed to do. Testing is very beneficial as it can help the prevention of bugs, it can enhance the development process and increase the overall quality of the system.

Several testing methods were used throughout this phase of the project. These testing methods included functional and user testing on the frontend and backend of the application.

Some of the tests on the application were based off the requirements that were defined within the requirements chapter, where multiple functional, user and technical requirements were outlined. The goal of these tests was to outline any issues or bugs within the application that needed to be addressed before the deployment phase. The goal of the user testing was to identify any issues in relation to the frontend/user interface of the application. It could be tested to see if there were any problems with the user flow, navigation and to get an idea of how the overall user experience of the application was.

The functional requirements were tested through functional and user tests. A functional test is a type of software test that determines if a piece of software is acting in accordance with the pre-determined requirements. It uses black box testing techniques, where the user has no prior knowledge to the internal systems logic (Bose, 2021). Functional tests can show if the piece of software is functioning and works as expected, but it does not indicate if it is easy to use.

A user test is a testing technique used in user-centred interaction design to evaluate an application by giving it to users to test. It gives input on how users interact with an application and can be used to show if the application is easy to use and intuitive for the user.

6.2 Functional Testing

Functional tests were carried out to see if certain pieces of the application worked as expected. The black box testing technique was used to test. This meant the tests would indicate whether the pieces of software functioned and worked as expected, but not whether the app was easy to use. For the tests, a description of the test case was described, and then the actual output of the test was tested against the expected output.

The functional tests were used to test a variety of parts of the app. They were split into the following sections:

- Authentication
- Navigation
- Calculation

6.2.1 Authentication

Tests were carried out on the authentication system to ensure the authentication system was working as expected. The expected outputs were users being stored in the database, helpful errors being provided to users and the users being directed to the flow to begin linking their bank institutions.

| Test No. | Description | Input | Expected Output | Actual Output | Comments |
|----------|--|--|---|--|--|
| 1 | User Login & Registration – error handling | Leave input fields blank | Errors provided to users | Errors are provided on each input field | Working as expected |
| 2 | Error handling on register – user cannot register with existing email | Register an account with an existing email | Error provided to user – user with email already exists | Error thrown – user already exists with that email | |
| 3 | Error if user tries to set passcode with letters | Input a passcode as a string when registering | Error to user – that passcode can only contain numbers | Works as expected, user is not allowed register if passcode contains letters | An error saying “this field is required” is still present when the user has typed in the input |
| 4 | User gets stored in the database after registering | Register a new user | User gets stored in the ‘users’ collection in MongoDB | User gets stored in the user’s collection | |
| 5 | User registers and link account flow begins | Provide correct details in register form, select bank and begin flow | User is brought to screen showing they have linked account successfully | Issue when the account selection screen loads – accounts are not loaded, and refresh is required | Still using test sandbox finance bank accounts for testing purposes |
| 6 | User with linked accounts gets directed to the home screen after login | User logs in to their account | User is directed straight to the home screen | User gets directed to home screen | A React warning is displayed when the screen loads |
| 7 | User that has no linked account gets directed to link account flow after login | User does not finish linking accounts flow. Then the user tries to login again | User gets directed to the link accounts flow | Works as expected. User gets directed to link account flow | Bug was noticed during this process – on register, spaces are allowed in email |

6.2.2 Navigation

Tests were conducted on the navigation of the application to make sure the user-flows were working and not complicated. Various tests were carried out to test the overall navigation of the user interface to ensure the navigation worked as expected.

| Test No. | Description | Input | Expected Output | Actual Output | Comments |
|----------|--|--|---|--|---|
| 1 | Logging out of account | User taps on 'Logout' button | User redirected to 'welcome' screen. Session is deleted from database | User redirected to welcome screen. Session was deleted from database | A React warning was given on welcome screen on some logouts |
| 2 | User is navigated to 'Savings Index' when they tap on 2 nd icon in bottom tab navigation | User taps on icon in bottom navigation | User is navigated to Savings Index screen | Works as expected. | |
| 3 | User is navigated to screen where they can add a new item when they tap on '+' icon in bottom tab navigation | User taps on '+' button in bottom navigation | User is navigated to screen where they can choose to add a new item | Working as expected | This screen's design might need to be improved |
| 4 | User is navigated to 'Bills index' when they tap on 3 rd icon in bottom tabs | User taps on bills icon in bottom navigation | User is navigated to bills index screen | Working as expected | Some design issues on this screen |
| 5 | User is navigated to 'Transactions index' when they tap on the last icon in bottom tabs | User taps on the transaction's icon in bottom navigation | User is navigated to transactions index screen | Working as expected | |
| 6 | From 'Bills index' screen, user can switch to subscriptions | User switches bills to 'subscriptions' by tapping button | Switch from bills to subscriptions | Functionality not implemented yet | |
| 7 | From 'Transactions index' screen, users can filter by category/country etc. | User presses 'Category' or 'Country' filter button | Filter transactions | Functionality not implemented. Users can only view transaction list | |

6.2.3 Calculation

Tests were conducted to check the calculations of the application. These tests were performed to ensure important features of the application were working as expected in order

to provide a positive user experience. Some of these tests included updating the user's total account balance when they switched between accounts, or the calculations being performed to update the total savings goal when the user added a new saving goal.

| Test No. | Description | Input | Expected Output | Actual Output | Comments |
|----------|---|---|--|---|---|
| 1 | User's total account balance is shown on home screen | Navigate to home screen | Total balance should be displayed | Total balance is shown on home screen | Total balance shows for the currently selected account |
| 2 | User's transactions are changed when account is switched | From home screen, switch to a different account | When account is switched, transactions should be different | Works as expected, transactions are changed | Possibly would be easier to notice if real account data was used |
| 3 | Total saving goal is updated when a new saving goal is added | Add a new saving goal | Total saving goal should be updated with the previous amount added | Calculation successful, saving goal gets updated | Bug – warning displayed when user selects icon |
| 4 | Current savings amount is updated when user adds funds to saving goal | Add funds to a saving goal | The current amount in the saving goal should change | User cannot add funds to saving goals | If the savings 'current funds' gets updated manually in DB, works as expected |
| 5 | Saving goal progress % is updated when funds are added to saving goal | Add funds to a saving goal | Progress % on saving goal should be updated | User cannot add funds to saving. However, progress % is updated if manually updated from the DB | When navigating back to home screen, saving goal was not updated instantly |

6.2.4 Analysis of Functional Tests

Upon the analysis of the functional testing results, it was found that a lot of the functionality was working as expected and met the defined requirements. These tests also highlighted slight issues and bugs that were present in the application. Some of these issues simply required a UI tweak and some of them required a process of debugging to resolve the issue. Most of these issues weren't issues that would make the app un-useable, however they would possibly hinder the user's experience while using the app. For example, users might add a space in their email while registering, and since this was a bug in the app, the account

would be created and the user might not notice the issue, which could then lead to some frustration when trying to log in again. After completion and analysing the functional tests, the focus was then set on the user tests so that it could be determined if the app was easy to use and intuitive for the users.

6.3 User Testing

User tests were performed to ensure that the app was easy to use, intuitive for all users and provided a positive user experience overall. Firstly, the objectives for the usability tests were decided, these objectives included determining what users enjoy about the application – such as if it is easy to use, or if the colours in the app are pleasing. Once the objectives were defined, the tasks could then be designed. These are tasks that the users are asked to do. These tasks help provide a better sense of how users navigate the application and decide on what information is important.

Revisiting the goal of the application - it was to provide users with an application that is pleasing for users with an easy-to-use interface, making it easy for users to link multiple financial institutions to their account and to make saving easier. When carrying out the primary research for the application, it was found that there have been several design problems in recent years within multiple Irish banking applications that are used. The likes of AIB and Bank of Ireland have suffered from poor usability and design however each bank has been striving to improve their application's user experience. Some of these applications have resulted in a tedious user experience at times.

Examining some of the flaws of these applications, the goal of this application is to provide users with a simple, organized and easy to use interface and to allow users to view their financial state within one application. The application is aimed at users that want to increase their skills in saving money. The goal of the user tests was to help determine if any changes would need to be made to the user interface and user flows to make it a more pleasing and intuitive experience for the user.

6.3.1 Test Participants

There were a total of four test participants that took part in the user tests. The test participants for the user testing tasks varied in age groups and financial literacy level. The application is targeted at users from teenagers to adults who know how to use mobile apps and want to make it easier to organize their finances and savings. Generally, young adults can sometimes struggle to save money and as a result, young users would most likely be the most active users on the application.

Getting feedback from various types of users on the application was beneficial as it pointed out which parts of the application are most popular, which sections of the app stand out and which parts need to be improved further.

6.3.2 Test Environment

The testing environment that was used to carry out the user testing consisted of the application running on a laptop, the test participant, and the tester that explained tasks for the participant to carry out and who recorded the feedback.

6.3.3 Test Methods

The user testing method that was decided upon was ease of use. The reason for choosing this method is because the goal of the application was to make it easier for users to analyse their financial data within the app and to make it easier to save money.

The test tasks were then designed, and the test participants were recruited. The tests were then run, which involved the tester giving the participant a description of the application and how it works, the test task would then be described to the participant and the test would begin. As the participant attempted to complete the tasks, the tester observed the user and noted any feedback or difficulties that were encountered. After completion of the tests the participants were asked some post-test questions such as what they liked most/least about the app and if they had anything they would change about the app. The testing data was then summarized and analysed to draw conclusions and formulate any recommended design changes.

6.3.4 User Testing Tasks & Results

| Test No. | Description of Task | Comments/Feedback |
|----------|---|--|
| 1 | Registering an account, linking a bank institution to the app | <ul style="list-style-type: none">- Likes the look of the app, register form- Keyboard didn't show on simulator device, had to tell them to input through keyboard- Wondered what type/length passcode had to be- Bug: Accounts list didn't show up to select account- Found flow of linking accounts to be easy- Warning displayed when brought to home screen- Some users found it difficult to locate account information – it is required to press on the down arrow rather than the whole card- Not much information available for account, just bank name, account name- When user selected account from dropdown, the popup appeared with "Account Switched" even though it is the same account |

| | | |
|---|---|---|
| 1 | Finding information about transactions | <ul style="list-style-type: none"> - Some users found it easy to find transactions - Some found it difficult: the link on bottom tabs says “Recent”, with icon. Could be misleading - Users attempted to switch from ‘latest’ to ‘category’ or ‘country’ however this functionality is not implemented - Users attempted to tap on a single transaction to possibly show more details, this does not work - Easy to distinguish between income and an expense - Search icon is there however it does not currently work to search - Only shows a certain number of recent transactions |
| 2 | Navigating to profile screen | <ul style="list-style-type: none"> - Some users looked at the bottom navigation tabs for a profile link - A user struggled to find a way to the profile. Pressing on the hamburger icon to find the profile link may not be intuitive - User pressed on the profile image to navigate to profile - User pressed on the name to attempt to navigate – this is not a link currently - Once on the profile, users liked the minimalistic design, showing their name and email - These fields should be made editable - User tried to press on the profile image to change it, functionality not implemented |
| 2 | From the profile screen, the task is to add a new saving goal | <ul style="list-style-type: none"> - Users found it easy to navigate back to home using the back button - Some users used the + button within the savings card from the home screen to create - User tapped on the + button from the bottom navigation to add new saving - Limited number of icons in form - When icon chosen, a react warning appeared. The icon did not show up as a selected icon - User was unsure of what the colour input was for - Information about the saving goal was easy to find - Fields should be editable when saving goal is selected |

| | | |
|---|---------------------------------------|---|
| 3 | Setting up a bill to track on the app | <ul style="list-style-type: none"> - Users navigated to bills tab to find the new bill form - Some used the + button in navigation tabs to add new bill, easy to find - From the bills index screen, it was clear how to add a new bill - Screen needs a bit more design, the calendar input particularly - Warning shown on the screen about calendar input becoming deprecated - Form was easy to fill out - Users get navigated back to home screen rather than bills index -could be confusing - When navigating back to the bills screen, the total amount due is not updated – this is currently hardcoded - Currently all new bills get put in the 'Due' section with a message 'due yesterday' – this is incorrect as functionality not fully implemented - Users attempted to tap on the bill to see more information/interact with it - Bills should be editable - While exploring this section, some users were curious about switching to the 'Subscriptions' tab, which currently is not implemented |
| 4 | Linking another account to the app | <ul style="list-style-type: none"> - Some users attempted to find a button to link account from the accounts action sheet - Users used the + button from the bottom tabs to find an option to link account - Process to link account was easy to remember - Using real accounts might be less confusing than test accounts - Users were navigated back to the + option screen, confusing - unsure if the process was successful - Bug: when navigating back to home screen to find new account, it still shows the first account, and a screen refresh is required - Switching account is easy, feedback with a modal is good, possibly add to modal which account has been selected - When trying to compare accounts it was confusing, as the balances are the same, transaction names are the same also |

6.4 Conclusion

In conclusion of this chapter, it has helped gain an insight into what parts of the app worked as expected and highlighted any slight issues or bugs within the app. The functional tests helped identify if the pre-determined requirements were met and the user tests helped identify if the app was intuitive and easy to use. Both types of tests combined helped come up with some slight refinements for the app. The testing of the application showed why it is so important to conduct various tests on the app while it is in the development phase as it helps pinpoint any bugs that need to be addressed within the app to make it ready for production.

7 Project Management

This chapter discusses how the project was managed overall to make sure that there was steady and consistent progress made throughout the course of the project. Good project management was important to ensure that each deadline was met. The project was developed and managed through a number of stages, starting from the proposal/idea phase where the idea for the project was structured, to the requirements phase to determine the project's requirements, then the design phase where the designs were outlined to meet the requirements. The next stages were the implementation and testing phases. There were numerous project management tools that were used to ensure that the project was being developed efficiently. The tools that were used to manage the project were GitHub, a Kanban board on Microsoft's Tasks app, and a journal to record the progress made.

The project was developed over 4 months, from January 2022 to May 2022, with the SCRUM methodology being used. SCRUM is a framework for project management that emphasizes teamwork, accountability, and iterative progress towards a well-defined goal. Sprints are an agile software development concept used by SCRUM. A sprint is a period of time when the software development is done. Each sprint is usually short – one to two weeks and during each sprint you can work on completing items from the product backlog. Each sprint can end with a sprint review and then you choose another item from the backlog to develop. Sprints are continued until the deadline has passed.

The project was divided into 8 sprints overall, with each sprint lasting 2 weeks. At the beginning of each sprint, different items from the project backlog would be worked on and the aim was to have them completed by the end of the sprint. At the end of each sprint, a review would take place to establish how it went and if more time was needed for any particular tasks and what was required next. This schedule of sprints made it much easier to plan out and develop the application.

The main sprint deadlines were the following:

1. 23/01/22 - Requirements, Research & Prototype Development
2. 06/02/22 - Design Document v1
3. 20/02/22 – Implementation Document v1
4. 24/02/22 – Interim Presentation
5. 06/03/22 – Final Design Document
6. 30/03/22 – Final Implementation Document
7. 03/04/22 – Testing Document v1
8. 17/04/22 – Thesis v1
9. 08/05/22 – Final Thesis & Application versions
10. 10/05/22 – Project Presentations

Weekly meeting were carried out to discuss the progress of the project and any issues that were faced, and the project deliverables. The project was planned and managed using the Kanban method and using Microsoft's Tasks app. This is where the items for the project backlog could be stored and there was a list for the items to-do in the current sprint, the

‘Doing’ list that contained items that were currently being worked on, and the ‘Done’ list for items that were completed (Figure 98).

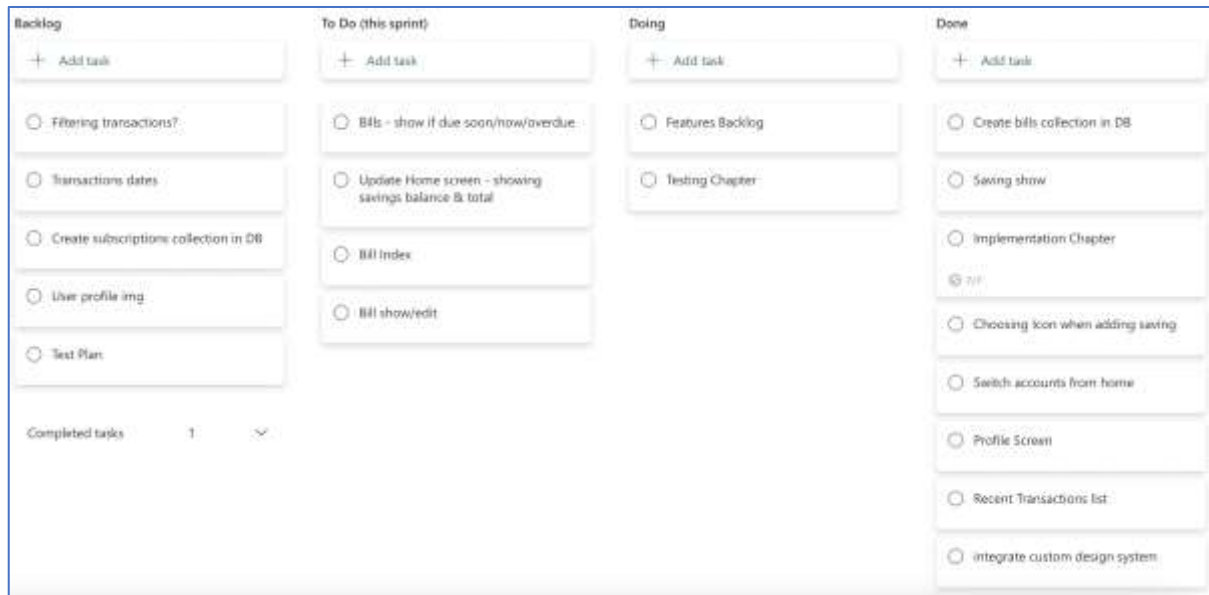


Figure 98 Microsoft Planner

7.1 GitHub

A repository on GitHub was created to host the code base and enable a version control system for the app (Figure 100). GitHub is a platform where you can host your code and is used for version control and collaboration. It lets you and others work on projects anywhere at any time (GitHub, 2022). The project was developed on GitHub using a number of branches. A branch would be created for most new features being developed on the app, for example a branch called ‘Authentication’ where the authentication system would be worked on. Branching is the way to work on different versions of the repository at the same time. One the feature was complete; the branch would be merged into the main branch. Using branches on a repository is a positive way to develop because they allow developers to develop features, work on bug fixes and safely experiment with new ideas within a contained area of the repository (GitHub, 2022).

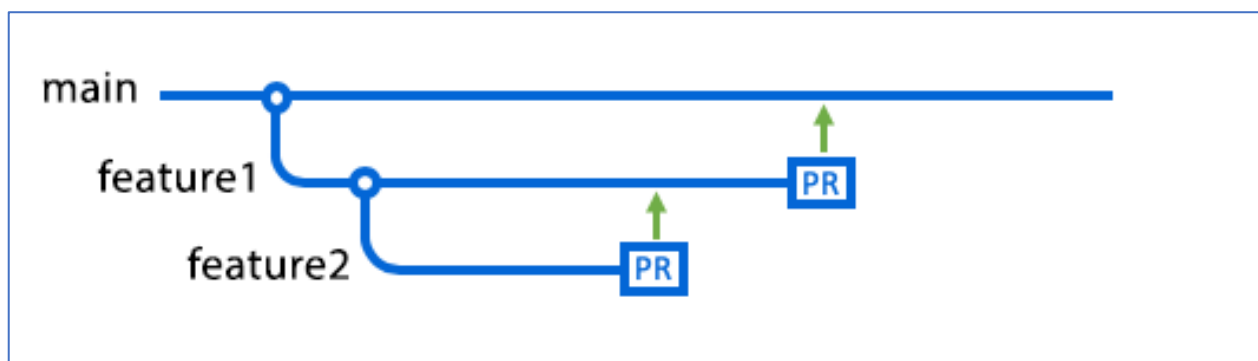


Figure 99 GitHub Branches

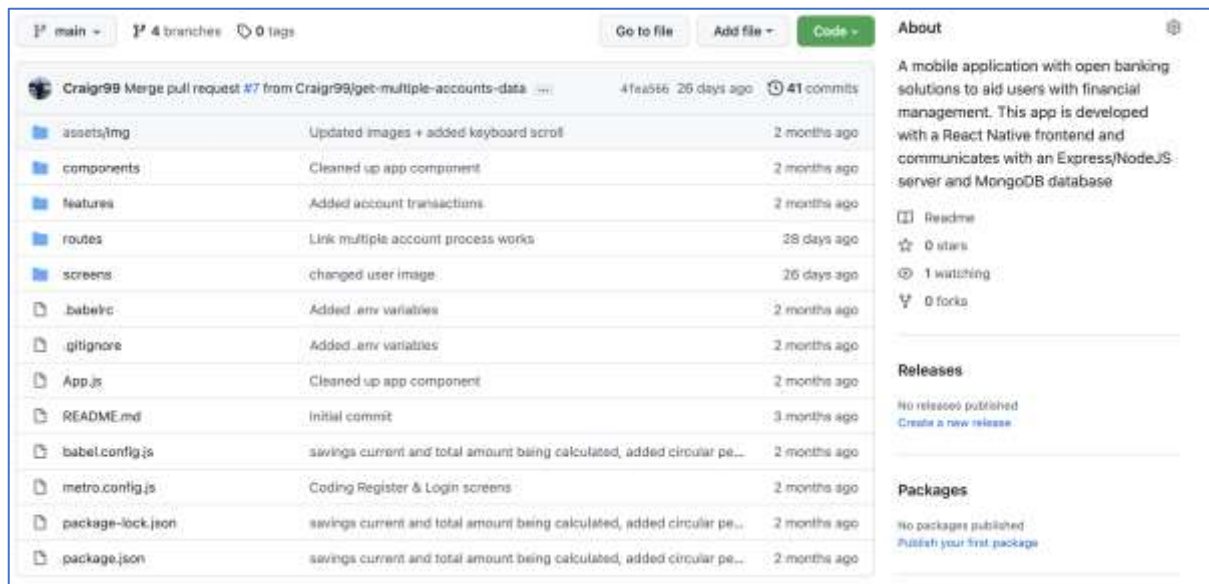


Figure 100 GitHub Repository

Each time an addition or change to the codebase was added a commit would be made to the repository. The frequency of these commits can be seen in Figure 101 below. This chart helps visualize the development of the project by showing the number of commits made throughout the course of the 4 months. It is visible that the highest point in the chart is in the month of February, which as described above, this was around the time of the implementation sprint - where a lot of the coding was done.

The chart shows that the number of commits decreases after February. This is due to the fact that a lot of the app's main functionality had been implemented, and that the focus then switched to testing the app, while making small changes to the code and integrating less important functionality. A lot of documentation for the app was then completed and updated at this stage.



Figure 101 Commits made to the repository over 3 months

7.2 Journal

A journal was used to document the development progress throughout the course of the project. The platform that was used is called Notion. Notion is an application that provides components such as notes, databases, Kanban boards, wikis, calendars and reminders.

Journal entries would be made at the end of most days/weeks to describe what progress was made, how things went and any difficulties that arose. Using the journal was good practice as it clarified what was complete and what was to be completed throughout each sprint.

7.3 Conclusion

The project was managed well throughout each sprint and as a result, a working application has been produced that meets the project requirements. Working on the project while being supervised has led to the development of a number skills in relation to project management such as communication, teamwork and leadership skills. Using professional methods and tools such as a journal, and project management tools like GitHub has provided valuable experience in working on a large software project. It was required to develop the application in a timely manner while using the SCRUM methodology which as a result helped improve project management skills overall.

8 Conclusion

This chapter discusses and summarizes how the development of a major software project went overall, any limitations that were encountered and the strengths and skills that were developed.

8.1 Project Summary

The overall goal of the project application was to research into a topic and then design, implement, test and deploy a full-stack application that would attempt to solve a problem. The application was to have numerous features which would help users to manage their finances overall, and the application was to be designed to become a useful tool which could be deployed for users to use on their mobile phones in the future.

The project began with researching into a topic and writing a literature review on the information available online that relates to the topic. The topic was around the area of mobile banking applications, user experience design and how the two relate to each other. The idea for the project was then developed after this stage. The requirements for the project were determined after performing research into various similar mobile applications that related to the project idea. The possible technologies that could be used to develop the application were then researched and the MERN stack was the chosen stack to use. The design phase of the application then commenced in which the system architecture was designed, as well as the user interface design of the application. This phase of the project was beneficial as it provided a better understanding on the user flows and design patterns required throughout the application. The implementation phase then consisted of the outlined designs being implemented through code. Once the major features of the application were implemented, the testing phase of the project began. This phase consisted of numerous types of tests – both user and functional tests, which provided an overview of how well the application functioned and outlined any bugs that were present.

As stated in the research section of this document, there are certain aspects of a mobile application that can be used to provide a better user experience within banking apps. One method that can be used is gamification which is the use of game principals and mechanics being applied to something to motivate and encourage users to perform activities. Another method is the use of personalisation within the app, which is when users can customize certain aspects of an application to their liking. The use of personalization was implemented in this project, by allowing users to customize their saving goals as they can choose a colour and an icon which represents the saving goal. An example of this personalized item can be seen in Figure 81 above. Gamification methods were originally planned to be integrated within the app however this was not possible due to the time restraint. This could be a possible area for development in the future work of the app.

8.2 Future Work

This section describes the possible future work that could be done on the application and the limitations that were encountered throughout the previous development process. This application could be further developed in numerous ways, as an application is constantly in a stage of development iterations and therefore is never usually a finished product. There is

potential for more features to be added and refined throughout the app, updates that can be made from user testing and bug fixes and also design updates to the user interface.

8.2.1 Savings Feature integrated into real bank accounts

One of the main ideas/features of the application was to allow users to create and manage saving goals. There were limitations to this as the user can currently only create the saving goal, however they cannot add or withdraw funds from their bank account to and from the saving goal. The idea behind this feature was to allow the user to add funds from their main account into the saving goal and then they cannot access these funds from their main bank account. This is to encourage better saving habits and discipline. This is a possible future development that could be integrated into user's real bank accounts.

8.2.2 Current features enhanced

There are a number of features currently working within the application with most of them having basic functionality. These features could be further developed to improve the apps overall functionality and usability. The transactions feature could be improved by adding filters to the transactions, allowing users to filter transactions by category, country etc. as currently the users can only view a list of the transactions. A feature could be added to the transactions where when the user taps on a single transaction it shows additional information about the transaction. The bills and subscriptions feature could also be developed further, to allow users to perform CRUD functionality on their bills and subscriptions.

8.2.3 Modifications from User Testing results

Following the results and analysis from the results of the user testing, there are multiple improvements that can be made to the app. These improvements would be made to the current functionality to ensure each feature works as expected and to make sure that the user experience is positive. There are also tweaks that would be made to the user interface to make the app intuitive for each user. It was found that there were various bugs throughout the application such as warnings being shown to the user that is from the React App. These bugs should be resolved to make it a smooth user experience and these fixes would be a priority. Some of these modifications would involve simple UI tweaks which could overall make some of the user flows less confusing and more natural.

8.2.4 Further User Testing

The process of user testing throughout the application has been beneficial, since it has provided a much clearer picture on how well each feature in the application functions, as well as how the user experience and user flows are in the app. When new features are added to the app it is important that each feature goes through a series of functional and user tests. Carrying out these tests on new features is an important part of the development process as it ensures the features provide a good experience for the users using the application.

8.2.5 Providing support for multiple countries

Currently the application only supports users from Ireland. When users are choosing what bank to link, a list of banks in Ireland are provided to the user. This is because when the

request is made to Nordigen APIs institutes endpoint, a “country” parameter is required in the URL. As shown in **Error! Reference source not found.** below, the country is set to ‘ie’ for Ireland. In a future development, this country parameter could be programmed to change based on the user’s country.

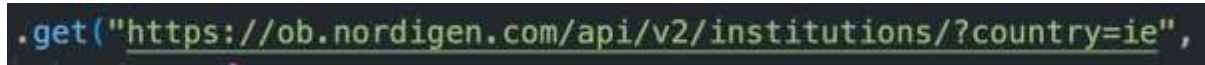


Figure 102 Bank List for Ireland

8.2.6 Providing the app for different mobile devices

Another possible further development of the application would be to develop and test it for multiple mobile devices. While developing the app, a simulator was used for testing the app and the simulator device was an iPhone 13 Pro Max. It is important that mobile apps are compatible for both iOS and Android devices, and for phones of different sizes. This process would involve testing the app on different devices for both iOS and Android, and tweaking the code to make it responsive for different device sizes.

8.3 Limitations

A few limitations became present while developing the app. The main limitation for the project was related to the API that was used. The Nordigen API provides different endpoints that allows developers to implement their account information solution in their applications. The endpoints provide accounts, account information and transactions. The information provided from these endpoints were limited and did not provide many endpoints to allow advanced features in the app such as filtering transactions. This was a limitation since the app relied on the APIs information for the majority of the functionality.

Another limitation for the project would be the time restraint. The timeline for the project was scheduled over a period of 4 months. Although a functioning mobile app has been produced within this timeframe, more time for the development of the app would be beneficial. According to various studies, a mobile app usually takes between 3 to 9 months to be developed for public release. This would generally just be the period for the implementation stage of the app, not including the research and design stages of building a mobile app (Moazed, 2015). For this major project, the 4-month timeframe included the research, design, implementation and testing phases.

8.4 Learning Outcomes

This project covered several aspects of the development of a major software project such as researching, designing, implementing, and testing a mobile application. The process in which the application was developed, with the project being developed and managed over a given period of time and in different sprints with a supervisor providing support and input, provided experience of what it is like to work on a real-world software project at a company. Some aspects of the project were not covered in the course curriculum, which required me to work on my own initiative and self-teach myself new skills and concepts. This is very beneficial as many successful software developers must work on their own initiative to problem-solve and teach themselves new things. Working on this project has improved my skills in programming,

researching and writing, my design skills, time-management and communication skills. I have enjoyed working with the technologies that were used and I have become more knowledgeable on each technology. I can now say that I would be comfortable working on a full MERN stack application, from the initial idea/researching phase to the deployment of the product.

8.5 Final Words

To conclude, applications that have a pleasing user interface and that are easy to use for users can make managing finances easier and more enjoyable for users. Some banking applications can tend to be confusing to users and as a result, make finances and numbers difficult for some users to understand and work with. This application could be used as a tool in the future with further development and testing and there are many directions in which the development of the app can take. User's bank accounts could be integrated with this app to allow users to manage their finances and savings from a single application. Overall, the development of this project has been a joy and has taught me various new concepts and skills, and has improved my current skillset, and I would enjoy seeing how far the application could be taken in future developments.

9 References

- Amin, M., Rezaei, S., & Abolghasemi, M. (2014). User satisfaction with mobile websites: the impact of perceived usefulness (PU), perceived ease of use (PEOU) and trust. *Nankai Business Review International*.
- Azwa, M., & Azrul, H. J. (2017). User Experience Design (UXD) of Mobile. *Faculty of Computer Science and Information Technology*, 197-200.
- Babrovich, N. (2017, July 10). *How to use gamification in banking to engage customers and employees*. Retrieved from Scnsoft.com: <https://www.scnsoft.com/blog/gamification-in-banking>
- Batchelor, B. (2017, September 26). *The History of E-Banking*. Retrieved from Bizfluent: <https://bizfluent.com/about-5109945-history-ebanking.html>
- Bose, S. (2021, May 11). *Functional Testing : Definition, Types & Examples | BrowserStack*. Retrieved from BrowserStack: <https://www.browserstack.com/guide/functional-testing>
- Clearbridge Mobile. (2018, December 19). *How Biometric Authentication is Shaping the Future of Mobile Banking | Clearbridge Mobile*. Retrieved from Clearbridge Mobile: <https://clearbridgemoible.com/biometric-authentication-shaping-future-mobile-banking/>
- Dhruw, D. (2020, November 11). *ORM and ODM — A Brief Introduction - Spider - Medium*. Retrieved from Medium: <https://medium.com/spidernitt/orm-and-odm-a-brief-introduction-369046ec57eb>
- Dyslexia Ireland. (2020, December). *Dyscalculia and Maths Difficulties - Dyslexia Ireland*. Retrieved from Dyslexia Ireland: <https://dyslexia.ie/info-hub/about-dyslexia/dyscalculia-and-maths-difficulties/>
- Enginess. (2016, September 14). *How Designing for Mobile is Different from Desktop | Enginess Insights*. Retrieved from Enginess.io: <https://www.enginess.io/insights/6-ways-designing-for-mobile-sites-is-different-from-desktop>
- Finextra Editorial Team. (2019, April 7). *3 Trends That Will Shape Digital Banking's Future*. Retrieved from Finextra Research: <https://www.finextra.com/blogposting/17038/3-trends-that-will-shape-digital-bankings-future>
- GitHub. (2022). *About branches - GitHub Docs*. Retrieved from GitHub Docs: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches>
- GitHub. (2022). *Hello World - GitHub Docs*. Retrieved from GitHub Docs: <https://docs.github.com/en/get-started/quickstart/hello-world>
- GoodID team. (2021, April 26). *Biometric data*. Retrieved from Good ID: <https://www.good-id.org/en/glossary/Biometric-data/>
- Hennessy, N. (2021, March 7). *Changes in banking landscape only just beginning*. Retrieved from Irish Examiner: <https://www.irishexaminer.com/news/spotlight/arid-40238024.html>
- IBM. (2021, March 04). *IBM Docs*. Retrieved from Ibm.com: <https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-use-case>
- IBM Cloud Education. (2021, April 6). *rest-apis*. Retrieved from Ibm.com: <https://www.ibm.com/cloud/learn/rest-apis>

- Joo, H. (2017). A Study on Understanding of UI and UX, and Understanding of Design According to User Interface Change. *International Journal of Applied Engineering Research*.
- Keivani, S. F., Jouzbarkand, M., Khodadadi, M., & Sourkouhi, K. Z. (2012). A General View on the E-banking. *International Proceedings of Economics Development & Research*, 62.
- KPMG. (2019). *THE FUTURE OF DIGITAL BANKING*. KPMG.
- Material Design. (2022). *Material Design*. Retrieved from Material Design: <https://material.io/design/color/the-color-system.html#tools-for-picking-colors>
- Meadows, M. (2019, August 23). *The State of Mobile Banking and Digital UX*. Retrieved from <https://www.finastra.com/viewpoints/articles/state-mobile-banking-and-digital-ux>
- Moazed, A. (2015, July 29). *How Long Does It Take to Build an iOS or Android Mobile App?* Retrieved from Applico | Platform Experts: <https://www.applicoinc.com/blog/long-take-build-ios-android-mobile-app/#:~:text=It%20will%20usually%20take%203,of%20building%20a%20mobile%20a>pp.
- NALA. (2019, October 8). *Literacy and numeracy in Ireland - Nala*. Retrieved from Nala: <https://www.nala.ie/literacy-and-numeracy-in-ireland/>
- Perea, P., & Giner, P. (2017). *UX Design for Mobile*. Packt Publishing Ltd.
- Rahi, S., Ghandi, M. A., & Alnaser, F. M. (2017). Predicting customer's intentions to use internet banking: the role of technology acceptance. *Management Science Letters*, 513-524.
- Rodrigues, L. F., Oliviera, A., & Costa, C. J. (2016). Does ease-of-use contributes to the perception of enjoyment? A case of gamification in e-banking. *Computers in Human Behavior*, 114-126.
- Statista Research Department. (2021, October 19). *Online banking users worldwide by region 2020* / Statista. Retrieved from Statista: <https://www.statista.com/statistics/1228757/online-banking-users-worldwide/>
- Svilar, A., & Zupančič, J. (2016). User Experience with Security Elements in Internet and Mobile Banking. *Organizacija*, 49.
- The Interaction Design Foundation. (2014). *What is Mobile User Experience (UX) Design?* Retrieved from The Interaction Design Foundation: <https://www.interaction-design.org/literature/topics/mobile-ux-design>
- The Interaction Design Foundation. (2021). *What is User Experience (UX) Design?* Retrieved from The Interaction Design Foundation: <https://www.interaction-design.org/literature/topics/ux-design>
- W3 Schools. (2022, January). *React useState Hook*. Retrieved from W3Schools.com: https://www.w3schools.com/react/react_ustate.asp
- Wang, M., Cho, S., & Denton, T. (2017). The impact of personalization and compatibility with past experience on e-banking usage. *International Journal of Bank Marketing*, 45-55.

10 Appendices

10.1 Appendix A (Survey)

1

What age category do you belong in? *

☐ 18-24

☐ 25-34

☐ 35-45

☐ 46-60

☐ Over 60

☐ Prefer not to say

2

Have you ever used an online banking application such as AIB / Revolut etc.? *

☐ Yes

☐ No

3

What is your preferred device to use for managing your finances? *

☐ Phone

☐ Desktop Computer

☐ Laptop

☐ Tablet

☐ Other

4

Do you ever find it difficult to manage your finances? *

☐ Yes

☐ No

☐ Sometimes

5

Do you ever find banking applications difficult to use? *

☐ Yes

☐ No

☐ Sometimes

6

If yes, can you describe the difficulty you faced?

Enter your answer

7




Would you use an app like this to manage your finances?

1 = Not Likely
5 = Very Likely

1 2 3 4 5

☐ ☐ ☐ ☐ ☐

8



Would you use an app like this to keep track of your bills/subscriptions?

1 = Not Likely
5 = Very Likely

1 2 3 4 5

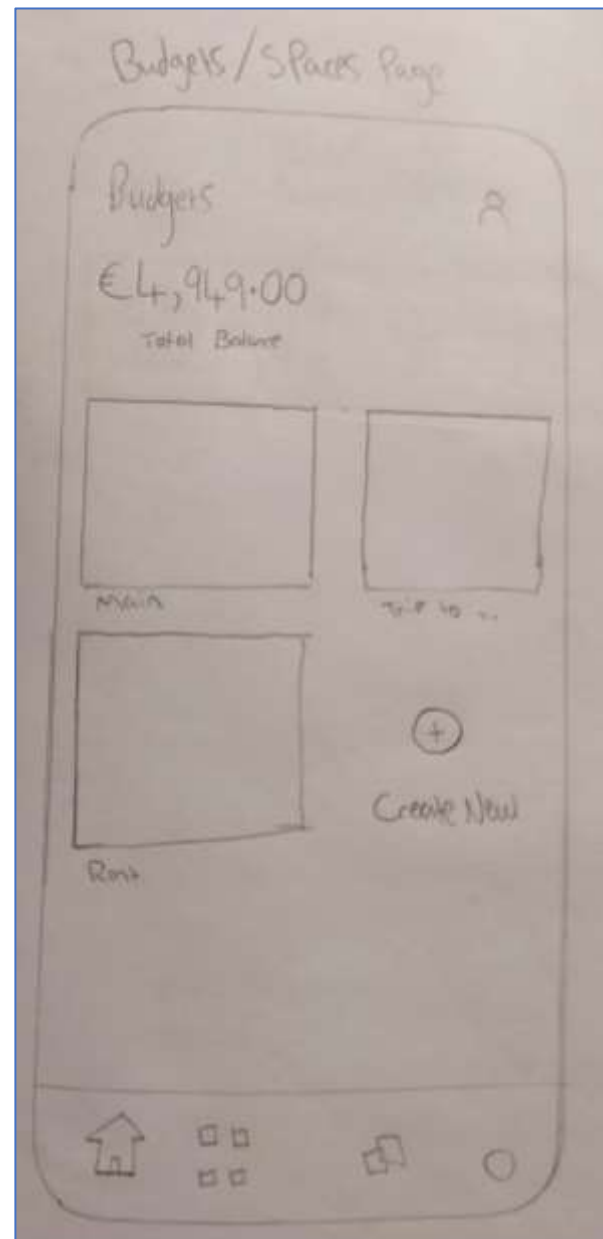
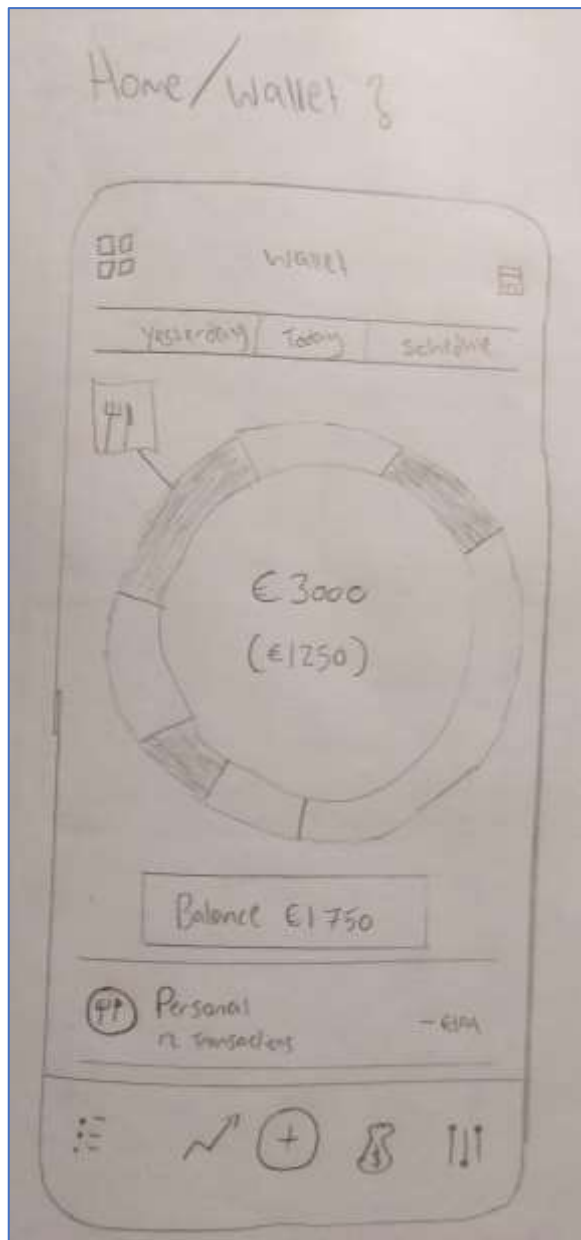
☐ ☐ ☐ ☐ ☐

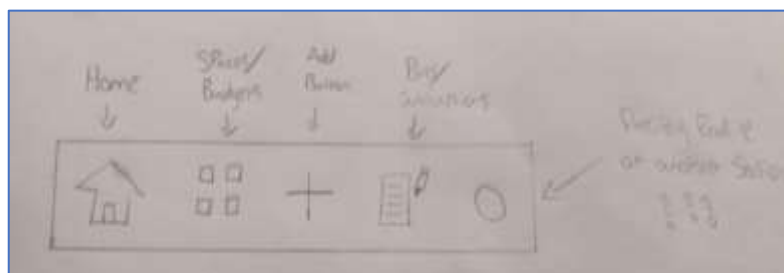
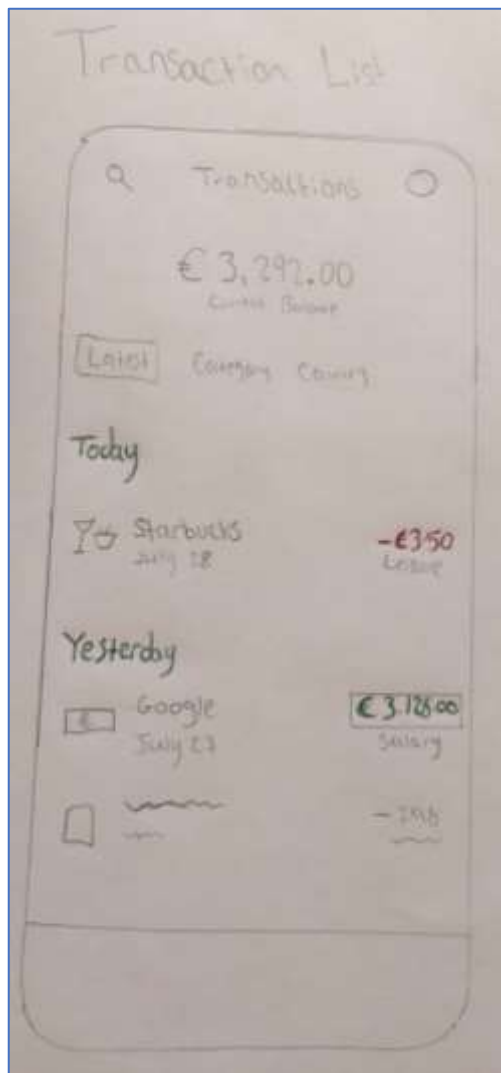
9

Do you use any application(s) to keep track of your bills/subscriptions? *

☐ Yes
☐ No


10.2 Appendix B (Paper Prototypes)





Bills and Subscriptions +

☐ Bills ☐ Subscriptions

Upcoming Bills 



€1,192 due

DUE

| | | |
|-----------|-----------|-------------------------|
| WED 23 | Home rent | €1,198 Due Yesterday |
|-----------|-----------|-------------------------|

UPCOMING

| | | |
|-----------|----------|------|
| Fri 25 | Car Loan | €252 |
|-----------|----------|------|

Subscription
↓

| | |
|---------|----------|
| Spotify | €8.50/mo |
|---------|----------|

10.3 Appendix C (Survey Documents)

10.3.1 Test Introduction

Major Project app usability test

Hello, welcome to the usability test for my final year major project application. Thank you for taking the time to carry out this test, your feedback is valuable, and it will help determine if my app functions efficiently. The test should take between 10 – 20 minutes.

My project is called "FundNest". FundNest is a mobile app where users can register an account and link various banking institutions to the app such as AIB, Bank of Ireland, Revolut etc. And can analyze their expenses and income. Users can also create saving goals, where they determine the goal amount and name, and it is possible to personalize these goals with features such as adding an icon and a colour to represent each goal. Users can also add and keep track of their bills and subscriptions in the app. The ultimate goal of the app is to make it easier to manage finances and savings.

During this session, you will be asked to fill out a consent form, then you will be asked a few simple pre-test questions. I will then bring you to the computer screen where the app is running on a virtual simulator. This is where you will use the app. You will then be presented with a couple of tasks that we ask you to complete. As you are going through the app, I would ask you to think aloud as much as possible – describe what you are seeing and what you are trying to do. You can be honest with your thoughts and opinions, please note that you do not have to worry that you are going to hurt our feelings, as we are doing this to help improve the application, so your honest reactions will be a great help!

If you have any questions throughout or at the end, feel free to ask and I will be happy to answer them. At the end of the test, I will ask you to fill out a post-test questionnaire; these are simply short and simple questions to describe your overall thoughts and feedback on the test.

Thank you for taking your time to take part in this usability test.

10.3.2 Consent Form

Usability Testing - Consent Form

Please read and sign this form.

In this test:

You will be asked to perform tasks on a computer.

You will be asked questions based on the given tasks.

The screen will be recorded for research purposes only.

Participation in this usability test is voluntary. All information will remain strictly confidential. You can withdraw your consent to the test and stop participation at any time. If you have any questions, feel free to contact me at the email: N00171313@student.iadt.ie

"I understand that participation in this usability test is voluntary and I have read and understood the information on this form and had all of my questions answered."

Thank you!

Your participation is much appreciated.

...

Hi, Craig. When you submit this form, the owner will see your name and email address.

* Required

1. I agree *

☐ Yes

2. Enter your name

Enter your answer

Submit

10.3.3 Test Tasks

Task 1.

Enter the app as a new user. Register an account. Link your banking institution to the app. Once this flow has been successful, see if you can find information about the linked account, for example: transactions, account details etc.

Task 2.

Once logged into the app, and currently on the home screen, navigate to the profile section. Once you are on the profile screen, describe your thoughts about the profile – does it represent you and have all the information that you would like? Attempt to add a new saving goal. Once this has been added, try to find information about this saving goal

Task 3.

Scenario: you have recently started paying a direct debit every month for donations to a charity. You want to keep track of these bills, set up a new bill on the app.

Task 4.

You now want to link another bank account to the app. Attempt to do this once you are logged into the app. Once you have linked another account, try to compare and switch between the accounts that you have linked.

10.3.4 Post-Test Questionnaire

Major Project - post test questionnaire

Thank you for taking part in this usability study! It would be greatly appreciated if you could provide some feedback on the app

...

Hi, Craig. When you submit this form, the owner will see your name and email address.

1. What was the thing you liked the LEAST about the app?

2. What was the thing you liked the MOST about the app?

3. Is there anything you would change about the app?

4. What was your overall impression the app?

Submit

10.3.5 Testing Notes

| Test No. | Description of Task | Comments/Feedback |
|----------|---|--|
| 1 | Registering an account, linking a bank institution to the app | <ul style="list-style-type: none"> - Likes the look of the app, register form - Keyboard didn't show on simulator device, had to tell them to input through keyboard - Wondered what type/length passcode had to be - Bug: Accounts list didn't show up to select account - Found flow of linking accounts to be easy - Warning displayed when brought to home screen - Some users found it difficult to locate account information – it is required to press on the down arrow rather than the whole card - Not much information available for account, just bank name, account name - When user selected account from dropdown, the popup appeared with "Account Switched" even though it is the same account |
| 1 | Finding information about transactions | <ul style="list-style-type: none"> - Some users found it easy to find transactions - Some found it difficult: the link on bottom tabs says "Recent", with icon. Could be misleading - Users attempted to switch from 'latest' to 'category' or 'country' however this functionality is not implemented - Users attempted to tap on a single transaction to possibly show more details, this does not work - Easy to distinguish between income and an expense - Search icon is there however it does not currently work to search - Only shows a certain number of recent transactions |
| 2 | Navigating to profile screen | <ul style="list-style-type: none"> - Some users looked at the bottom navigation tabs for a profile link - A user struggled to find a way to the profile. Pressing on the hamburger icon to find the profile link may not be intuitive - User pressed on the profile image to navigate to profile - User pressed on the name to attempt to navigate – this is not a link currently - Once on the profile, users liked the minimalistic design, showing their name and email - These fields should be made editable - User tried to press on the profile image to change it, functionality not implemented |

| | | |
|---|---|---|
| 2 | From the profile screen, the task is to add a new saving goal | <ul style="list-style-type: none"> - Users found it easy to navigate back to home using the back button - Some users used the + button within the savings card from the home screen to create - User tapped on the + button from the bottom navigation to add new saving - Limited number of icons in form - When icon chosen, a react warning appeared. The icon did not show up as a selected icon - User was unsure of what the colour input was for - Information about the saving goal was easy to find - Fields should be editable when saving goal is selected |
| 3 | Setting up a bill to track on the app | <ul style="list-style-type: none"> - Users navigated to bills tab to find the new bill form - Some used the + button in navigation tabs to add new bill, easy to find - From the bills index screen, it was clear how to add a new bill - Screen needs a bit more design, the calendar input particularly - Warning shown on the screen about calendar input becoming deprecated - Form was easy to fill out - Users get navigated back to home screen rather than bills index -could be confusing - When navigating back to the bills screen, the total amount due is not updated – this is currently hardcoded - Currently all new bills get put in the 'Due' section with a message 'due yesterday' – this is incorrect as functionality not fully implemented - Users attempted to tap on the bill to see more information/interact with it - Bills should be editable - While exploring this section, some users were curious about switching to the 'Subscriptions' tab, which currently is not implemented |
| 4 | Linking another account to the app | <ul style="list-style-type: none"> - Some users attempted to find a button to link account from the accounts action sheet - Users used the + button from the bottom tabs to find an option to link account - Process to link account was easy to remember - Using real accounts might be less confusing than test accounts - Users were navigated back to the + option screen, confusing - unsure if the process was successful - Bug: when navigating back to home screen to find new account, it still shows the first account, and a screen refresh is required - Switching account is easy, feedback with a modal is good, possibly add to modal which account has been selected - When trying to compare accounts it was confusing, as the balances are the same, transaction names are the same also |