EARNit: A Personal Finance Management Mobile Application

Ioanna Kougia

Diploma Thesis

Supervisor: Apostolos Zarras

Ioannina, October, 2022

Τμήμα Μηχ. Η/Υ & Πληροφορικής
Πανεπιστήμιο Ιωαννίνων
Department of Computer Science & Engineering
University of Ioannina

Acknowledgments

I would like to express my gratitude to my supervisor professor Apostolos Zarras for his unwavering guidance. I also wish to thank my partner, Lawrence, and my family for their constant support.

October, 2022

Ioanna Kougia

Abstract

The main aim of this thesis is the design and development of a mobile application that facilitates personal finance management. With EARNit the user is able to follow his/her spending plan with the budget feature, and achieve short-term and long-term saving goals, with the goal feature. The user can manage his/her income, and expenses, including recurring expenses like bills and subscriptions. Our application uses informative graphs that visualize the user's progress and helps the user understand his/her spending habits. Furthermore, the user can see monthly reports for his/her transactions, and informative comparisons with the previous month. Since personal finances are an overwhelming subject for many people, the application is trying to simplify the subject and create a user-friendly user interface. Finally, one user account can be used by two or more people, making it a good choice for families and small businesses.

Keywords: mobile application, react native, personal finance management, money, budget, saving goal

Περίληψη

Ο κύριος στόχος της διπλωματικής εργασίας αυτής είναι ο σχεδιασμός και η ανάπτυξη μιας εφαρμογής για κινητά τηλέφωνα που διευκολύνει την διαχείριση των προσωπικών οικονομικών. Με το EARNit ο χρήστης μπορεί να ακολουθήσει το πλάνο εξόδων του/της, με χρήση των budgets, και να πετύχει τους βραχυπρόθεσμους ή μακροπρόθεσμους αποταμιευτικούς του/της στόχους, με την χρήση των goals. Ο χρήστης μπορεί να διαχειριστεί τα έσοδα και τα έξοδα του, συμπεριλαμβανομένων και των επαναλαμβανόμενων εξόδων όπως είναι οι λογαριασμοί και οι συνδρομές. Η εφαρμογή μας χρησιμοποιεί κατατοπιστικά γραφήματα που οπτικοποιούν την πρόοδο του χρήστη και τον βοηθούν να κατανοήσει τις καταναλωτικές του συνήθειες. Ακόμα, ο χρήστης μπορεί να δει μηνιαίες αναφορές για τις συναλλαγές του/της, και ενημερωτικές συγκρίσεις με τον προηγούμενο μήνα. Δεδομένου ότι η διαχείριση των προσωπικών οικονομικών είναι ένα περίπλοκο θέμα για πολλούς ανθρώπους, η εφαρμογή προσπαθεί να το απλοποιήσει και να δημιουργήσει μια ευχάριστη διεπαφή χρήστη. Τέλος, ένας λογαριασμός μπορεί να χρησιμοποιηθεί από περισσότερους από έναν χρήστες, κάνοντας την εφαρμογή καλή επιλογή για οικογένειες και μικρές επιχειρήσεις.

Λέξεις Κλειδιά: εφαρμογή για κινητά τηλέφωνα, διαχείριση προσωπικών οικονομικών, χρήματα, προϋπολογισμός, αποταμιευτικός στόχος

Table of Contents

1. Introduction

Time is the only non-renewable resource. Constantly craving new stuff and working hard to acquire them, consumes peoples' creative time. Expenses accumulate and become difficult to manage. Rather than contemplating, thinking, and planning, people start blaming their bad financial situation on their money shortage instead of their bad management skills. One basic step towards achieving financial literacy is to dedicate time to understanding our spending habits and take action to change them.

Financial literacy is understanding and effectively using various financial skills, including personal financial management, budgeting, and investing [1]. Modern life's fast pace combined with inflation and lack of financial education makes it nearly impossible for the average person to grow their wealth.

According to the S&P Global FINLIT Survey [2], conducted in over 140 countries and 150.000 people, 1 in 3 adults worldwide are financially literate. Specifically, in Greece, literacy rates are 45%, which stands to reason since Greece's primary and secondary educational system does not involve the slightest reference to money management. This results in young adults who do not understand basic financial concepts well enough. Poor financial decisions, self-insufficiency, and financial instability precisely describe most young adults in Greece. Achieving financial literacy could change that.

1.1. Motivation

Many teenagers and young adults think of financial matters as tedious but eventually some of them get intrigued by financial management. Suddenly they feel the need to understand their spending habits and make a realistic plan for their spending and savings. They might need something more automated than a notebook or a spreadsheet, but less automated than a 3rd party application that syncs to their bank accounts. A PFM

- Personal Finance Management- application that can be customized to their needs but also be user-friendly, and with as little complexity as possible. Sometimes, too much information and features can be overwhelming, especially on a subject like finances.

1.2. Envelope Budgeting System

There are multiple ways of approaching PFM. One of them, the Envelope Budgeting System, EBS, is particularly interesting. The foundational idea is that the user's spending categories are represented by actual envelopes. Every time the user has new income he/she puts the amount of money he/she wants to budget for each category, in cash, into the corresponding envelope [3].

Today, this approach but with virtual envelopes is been used in some PFM Applications. At the start of the budgeting period, the user sets limits for his/hers envelopes/budgets. During that period he/she virtually fills the envelopes with funds before spending. This envelope filling can happen once at the start of the period with the full amount, or more times with partial amounts. We can think of *envelope filling* as "taking funds out from our balance and assigning them to an envelope/budget". The goal of this process is for the user to be always left with -virtual, not actual- zero balance since all available funds are immediately allocated to envelopes. The user cannot overspend in one envelope unless he/she transfers virtual funds from another envelope or allocate money from their balance. This is quite a complex approach for an application since making an envelope (budget) and envelope filling are two different actions, with envelope filling and transfer adding to this complexity.

1.3. Method

While this thesis is organized in linear chapters, it was developed with respect to an iterative process. The following workflow has been repeated until all the requirements have been met:

- Requirements
- Design
- Implementation
- Testing

2. Related Work

This chapter describes and analyzes some popular PFM Applications, that we could be inspired by, based on a list of features listed below. In our analysis, EBS applications and non-EBS applications are included in order to understand a big range of applications.

1. Spending/ income categories: the categories that a user wants to divide their spending and income into.
2. Budgets/Envelopes: a spending plan, including one or more categories and a limit amount for spending.
3. Goals: a saving plan.
4. Accounts/wallets: a stash of money, whether it is in a bank account or in cash.
5. Transactions: transactions are added automatically when in a back synced application or by hand in an application that does not sync to bank accounts.
6. Recurring Transactions: When adding a transaction there is an option to mark that transaction as recurrent, meaning that the application will remind you of that transaction in some way.
7. Household: It is the option to share the same application account with another person, meaning it will be data persistent among devices, or in some cases share only some parts of one's account.
8. EBS: The user virtually fills his/her envelopes (budgets & goals) at the start of the budgeting period. This way, theoretically, the user "can not overspend" in one envelope unless he/she transfers funds from another.
9. Budget-Categories: This is regarding how customizable a budget is category-wise. There are three options here. It's either "1", meaning 1 budget for all categories, or "1-1", meaning one budget for one category, or "1-N", meaning one budget can have multiple categories.
10. Budging Period: This is regarding how customizable a budget is period-wise. Can weekly, monthly, yearly, or custom.
11. FINLIT: if the application offers any kind of financial literacy feature.
12. Learning Curve: Short, for an easy-to-use from the first day kind of application. Medium, for an application that takes some time for the user to learn. And, Long, for an application that is really complicated, and needs even more time to get familiar with.

13. Pro Version: The application has also paid version. If any of the features above are only in the paid version they will be marked with a “Pro”.

2.1. Money Manager, Expense & Budget

Money Manager Expense & Budget is one of the most popular android applications with more than 10 million downloads. The application does not sync to banks and the user inputs transactions manually. It facilitates efficient asset management and accounting by applying a double-entry accounting system and supports recurrent transactions. There is the option for viewing filtered transactions. Its charts include pie charts, one for income and expenses, and a monthly asset line chart. Through the credit card management function, the user can track his payable and outstanding balance, meaning the amount he still owes after payment and the total amount he owes, respectively. The user can set one budget per spending category for the current month. It displays daily weekly and monthly financial data, and statistics. It has a backup/restore function which indicates that data is stored locally.

Strengths: easy-navigated, transaction-focused
Weaknesses: rigid UI, inflexible budgeting, not for households

2.2. Monefy

Monefy is a cross-platform application with more than 5 million downloads. The user can manually input new records quickly through its intuitive interface. It features easy-to-read spending distribution charts and a record list, using different icons for every category. It displays data in fixed and custom periods, grouped by account. It can be synchronized with a cloud account and used on different devices. The user makes a monthly budget. Some features like recurring payments, creating custom categories, and syncing with multiple devices are only available in the paid edition.

Strengths: easy-navigated, user-friendly UI, short learning curve
Weaknesses: inflexible budgeting, limited features

2.3. Good Budget

Good Budget is a cross-platform, and web application with more than 1 million downloads, that uses the EBS. Users can create monthly and annual envelopes to plan their spending or savings. With the EBS, the user needs to allocate their available funds into the envelopes and transfer between envelopes adapting to changing situations. It

features two kinds of reports. One is analyzing spending by envelope using a pie chart and the other is monitoring cash flow using bar charts. The web application has some more features including bank account statement import and more reports.

Strengths: short learning curve, simple, bank account statement import
Weaknesses: limited features, envelope filling

2.4. Wallet

Wallet is a cross-platform application with more than 5 million downloads. Unlike the previous applications, Wallet can be synchronized with bank accounts, automatically import transactions and factor them into the user's budget. It has really flexible budgets, with custom budgeting periods and multiple categories, featuring notifications when budgets are exceeded. Users can track their dept through the Depts feature, set saving goals through the Goals feature, and organize their bills through the Planned Payments feature. Selected accounts can be shared between friends and family who need to cooperate on a budget. The Statistics feature has plenty of detailed charts. Last but not least, it has a simple feature where they propose financial literacy books. Their claim that "whatever it is you need to accomplish this app offers the flexibility to meet your goals" stands true.

Strengths: bank sync, flexible budgets & goals, planned payments, financial literacy feature, depts
Weaknesses: medium learning curve, overwhelming charts

2.5. YNAB

YNAB is a cross-platform, and web application with more than 1 million downloads, using the EBS. It is the only application without a free version included in this analysis.. It can be synchronized with the banks but this feature is not available in Europe. It has features for flexible budgets, saving goals, debt pay-down, and detailed reports and charts. Additionally, It offers personalized support and some free resources like courses, videos, and podcasts. YNAB is more expensive than most paid apps and there is a good reason for it. It is not just a very detailed PFM Application, but also a whole support system for people with any kind of financial issues.

It is built around "The YNAB Method", which has four rules. The first rule is: "Give Every Dollar A Job", and it is referring to allocating funds to budgets once they become

available. The second rule is: “Embrace Your True Expenses” meaning dividing big expenses into monthly manageable “bills”. The third rule is “Roll With The Punches”, and it is about being flexible when overspending on a budget, and adjusting as needed. The fourth and last rule is “Age Your Money”, and it refers to using the last month's pay on this month's expenses. Their method is proven to work and reduce money stress.

Strengths: bank sync, flexible budgets & goals, planned payments, financial literacy feature, personal support
Weaknesses: long learning curve, envelope filling, too complicated

2.6. ExpenLess!

ExpenLess! is an android application with more than 50 thousand downloads. Adding transactions is done manually, and by receipt scanning, and it supports recurrent transactions and transaction filtering. Additionally, transactions can be labeled with tags. Its budgeting feature supports monthly and annual budgets with multiple categories. Its chart feature includes income & spending comparison bar charts and pie charts by category. Finally, it supports local and cloud backups.

Strengths: aesthetic UI, receipt scanning, tags for transactions
Weaknesses: complex navigation, medium-long learning curve, inflexible budgets

2.7. Summary

Table 1 shows the aforementioned applications with their features and our application proposal.

	Money Manager Expense & Budget	Monefy	Good Budget	YNAB	ExpenLess!	Wallet	EARNit
Downloads	10mil+	5mil+	1mil+	1mil+	50.000+	5mil+	
Recurring Transactions	x	X (Pro)	x	x	x	x	x
Household		X (Pro)		x		X (Pro)	x

EBS			x	x			
Goals			x	x		x	x
Budget - Categories	1-1	1	1-1	1-N	1-N	1-N	1-N
Budgeting Period	M	M	M,Y	custom	M,Y	custom	custom
FINLIT				x		x	x
Learning curve	medium	short	short	long	medium	medium	short

Table 1. Money management applications and their features

3. Technologies Used

This chapter lists the technologies, frameworks, and programming languages used in this thesis.

3.1. JavaScript

JavaScript (JS) [4] is a lightweight, interpreted programming language. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js. JavaScript is now used by an incredible number of high-profile applications, showing that deeper knowledge of this technology is an important skill for any web or mobile developer.

3.2. React Native

React Native [5] is a front-end framework that allows us to build mobile apps using only JavaScript. It uses the same design as React, letting us compose a rich mobile UI from declarative components. With React Native, we don't build a mobile web app, an HTML5 app, or a hybrid app; we build a real mobile app that's indistinguishable from an app built using Objective-C or Java. React Native uses the same fundamental UI building blocks as regular iOS and Android apps. We just put those building blocks together using JavaScript and React.

3.3. JSX

JSX [6] is a syntax extension to JavaScript, used by React Native. It is used to describe what the UI looks like.

3.4. Expo

Expo [7] is a set of tools built around React Native. It is easier to set up and configure, eliminating the need for Xcode or Android Studio, and its only requirements are Node.js and a smartphone. Additionally, the Expo Go mobile application [8] allows application testing during development.

3.5. Node.js

Node [9] allows developers to write JavaScript code that runs directly in a computer process itself instead of in a browser. Node can, therefore, be used to write server-side applications with access to the operating system, file system, and everything else required to build fully-functional applications.

3.6. Express

Express [10] is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

3.7. Axios

Axios [11] is a promise-based HTTP Client for node.js and the browser. It is isomorphic, which means it can run in the browser and node.js with the same codebase. On the server side, it uses the native node.js HTTP module, while on the client (browser) it uses XMLHttpRequests.

3.8. MySQL

MySQL [12] is a relational database management system (RDBMS) developed by Oracle that is based on SQL. SQL (structured query language) is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is treated as a set of tables.

4. Requirements

In this chapter we present the User Stories and the Use Cases for our application.

4.1. User Stories

A user story is a sort, simple description of a feature told from the perspective of the user. In this section, the user stories are listed in the form of cards for each application feature. Table 2 contains the application account User Stories.

4.1.1. Application account

User Story	**As a user I want to be able to**	**So that**
1.	create an account using only a username, an email, and a password.	I can use the application in a more secure way.
2.	use the app from different devices	I can change devices without worrying about losing my progress.
3.	stay logged in, in my devices	I don't waste time logging in every time I want to use the application.

Table 2. User Stories of the application account

4.1.2. Wallets

A wallet can be a virtual or a physical stack of money, like bank accounts or cash. Table 3 contains the User Stories about the wallet feature.

User Story	**As a user I want to be able to**	**So that**

4.	add a wallet	I can use it in the application.
5.	view all my wallets	I know the balance of each one of my wallets.
6.	see the transactions made from a wallet	I can view my transactions filtered by a wallet.
7.	edit a wallet	if I stopped tracking for a while I can easily start tracking again.
8.	delete a wallet	if I stop using a wallet it will not appear on my wallets.

Table 3. User Stories of the Wallet feature

4.1.3. Categories

There are two types of categories, income, and spending categories. Their differentiation is only logical, as they both have the same functions. Table 4 contains the User Stories about the categories feature.

User Story	**As a user I want to be able to**	**So that**
9.	add income and spending categories	I will have as many categories I need.
10.	change my income and spending categories' name and/or icon	I may correct a mistake or make changes that accommodate my needs better.
11.	view my spending and income categories	I can manage them.
12.	delete an income or spending category	if a category does not accommodate my needs it

		will not appear on my categories.

Table 4. User Stories of the Category feature

4.1.4. Transactions

A transaction is a physical or a virtual exchange of money. When a user makes a transaction in real life, he/she then needs to add the transaction to the application. A transaction is made through a wallet and with a spending or an income category, depending on whether it is an expense or an income. The user can also create a recurring income or expense so that the application will remind him/her of it and add the transaction record with the click of a button. Table 5 contains the User Stories about the transaction feature.

User Story	**As a user I want to be able to**	**So that**
13.	create a transaction in an easy and timely manner	I can keep track of a transaction on the application.
14.	view my transactions	I can remember what I spent/earned.
15.	filter transactions by date range	I can quickly find what I spend/earned at a specific time.
16.	change some of the details of a transaction	I can easily correct a mistake.
17.	undo a transaction	I can easily correct a mistake.
18.	schedule a recurring income or expense	I can be reminded of it by the application, and create it faster with a click of a button.

19.	view my recurring transactions	I can plan ahead and be ready for them.

Table 5. User Stories of the Transaction feature

4.1.5. Budgets

A budget is an estimate of how much money a user is going to spend over a certain period of time. Depending on the person, a user might want to make a budget for all his/her spending categories, one budget for each spending category, or anything in between. There are no standard budgeting periods. A user chooses the end date of his/her budget, and the start date can be at least the day of the budget's creation, meaning that a user can make a budget for a future period, but not for a past one. Table 6 contains the User Stories about the budget feature.

User Story	**As a user I want to be able to**	**So that**
20.	make a budget	I will plan my expenses.
21.	change some of the budget's details	I can adjust to changing situations.
22.	add multiple categories to a budget	I will have a budget that is fully customized to my needs.
23.	customize the duration of my budget	I am not restricted in monthly and annual budgets.
24.	see a visual representation of my budget's progress	I can understand my progress at a glance.
25.	see a visual representation of the money i spent during the budgeting period	I can understand my spending habits.
26.	see the details (initial budget, spend amount, balance amount, a bar chart of the total	I can further analyze my expenses.

	spending amount of each day of the budget, and transactions) of a budget	
27.	view all my budgets in one place	I can check their progress.
28.	get notified when I overspend on a budget	it comes to my attention.

Table 6. User Stories of the Budget feature

4.1.6. Goals

A goal is a saving goal. Generally, a saving goal is about setting a target amount and having that amount until a target date. Reality is more complicated than that since not all goals have a target amount or date. The application's goals are customizable regarding target amounts and dates. A goal, unlike a budget, is not connected with spending categories. Table 7 contains the User Stories about the Goal feature.

User Story	**As a user I want to be able to**	**So that**
29.	make a goal with or without a target amount and date	I can plan for my big expenditures and/or save for my future.
30.	see a percentage visual representation of my goal's progress	I can understand my progress at a glance.
31.	see the details of a goal	I can review my progress and make estimations for the future.
32.	view all my goals in one place	I can check their progress.
33.	add or subtract an amount from a selected goal	I can adapt to changing situations.
34.	see all my reached goals in one place	I can review my achievements.

Table 7. User Stories of the Transaction feature

4.1.7. Insights

The insights show the user some charts regarding his/her spending and income. Table 8 contains the User Stories about the Insights.

User Story	As a user I want to be able to	So that
35.	see pie charts of my expenses and income in spending and income categories respectively	I can get a visual representation.
36.	choose any date range for the pie charts	I can see the ones of older periods and compare.
37.	see simple monthly reports	I can easily see a spending and income summary of a specific month.
38.	see a comparison with the previous month in the monthly reports	I can easily see if i made or spend more that the previous month.

Table 8. User Stories of the Insights feature

4.2. Use Cases

UC1. createAccount

ACTOR

User

MAIN FLOW

1. The use case starts when the user opens the application.
2. The system redirects the user to the “Create account” screen.
3. The user enters the required user account information values and requests that the system saves the entered values.
4. The user clicks the “Create account” button.
5. The system validates the entered user account information.

6. The values for the user account information are saved in the user account.
7. The system notifies the user that the account has been successfully created.

ALTERNATIVE FLOW 1: The User already has an account.

UC2: sign-in

ALTERNATIVE FLOW 2: The User enters invalid User Account information

1. If the e-mail is already in use, the system informs the user of that.
2. If the e-mail is invalid, the valid naming criteria are displayed.
3. If the password is invalid, the valid naming criteria are displayed.

POST-CONDITIONS

The user account is created.

UC2: sign-in

ACTOR

User

PRECONDITIONS

The user has created a user account.

MAIN FLOW

1. The use case starts when the user opens the application.
2. The system redirects the user to the “Create Account” screen.
3. The user clicks the “Sign in” button at the bottom of the screen.
4. The system redirects the user to the “Sign in” screen.
5. The user enters the required user account information values and requests that the system saves the entered values.
6. The user clicks the “Sign in” button.
7. The system validates the entered user account information.

POST-CONDITIONS

The user is signed-in.

ALTERNATIVE FLOW 1: The User enters invalid User Account information

1. If the user account information values entered by the user are invalid, the system displays a relevant message.

UC3: addWallet

ACTOR

User

PRECONDITIONS

The user has created a user account.

MAIN FLOW

1. The use case starts when the user clicks on the wallet icon on the bottom tab navigator.
2. The system redirects the user to the “Transactions” screen and displays all existing transactions in a list.
3. The user swipes left, or clicks on “Wallets” at the top tab navigator.
4. The system redirects the user to the “Wallets” screen and displays all existing wallets with their current balance in a list.
5. The user clicks the “Add Wallet” on the top of the screen.
6. The system redirects the user to the “Add/Edit Wallet” screen.
7. The user enters the wallet name in the “Name” text box.
8. The user enters the available amount in the “Amount” text box.
9. The user clicks on the “Save” button.

POST-CONDITIONS

The user has added a wallet.

UC4: editWallet

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one wallet.

MAIN FLOW

1. The use case starts when the user clicks on the wallet icon on the bottom tab navigator.
2. The system redirects the user to the “Transactions” screen and displays all existing transactions in a list.
3. The user swipes left, or clicks on “Wallets” at the top tab navigator.
4. The system redirects the user to the “Wallets” screen and displays all existing wallets with their current balance in a list.
5. The user clicks on the edit icon of the wallet he/she wants to edit.
6. The system redirects the user to the “Add/Edit Wallet” screen featuring a “Name” and an “Amount” text box, filled with the current name and amount of the wallet.
7. The user changes the name and/or the amount of the wallet.
8. The user clicks on the “Save” button.
9. The system saves the changes.

POST-CONDITIONS

The user has successfully edited a wallet name or amount.

UC5: deleteWallet

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one wallet.

MAIN FLOW

1. The use case starts when the user clicks on the wallet icon on the bottom tab navigator.
2. The system redirects the user to the “Transactions” screen and displays all existing transactions in a list.
3. The user swipes left, or clicks on “Wallets” at the top tab navigator.

4. The system redirects the user to the “Wallets” screen and displays all existing wallets with their current balance in a list.
5. The user clicks on the delete icon of the wallet he/she wants to delete.
6. The system displays a pop-up question to the user “ Are you sure you want to delete this wallet?”
7. The user clicks on the “Yes” button.
8. The system deletes the wallet.

POST-CONDITIONS

The user has successfully deleted a wallet.

UC5: ViewWalletTransactions

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one wallet.

MAIN FLOW

9. The use case starts when the user clicks on the wallet icon on the bottom tab navigator.
10. The system redirects the user to the “Transactions” screen and displays all existing transactions in a list.
11. The user swipes left, or clicks on “Wallets” at the top tab navigator.
12. The system redirects the user to the “Wallets” screen and displays all existing wallets with their current balance in a list.
13. The user clicks on the wallet which transactions he/she wishes to see.
14. The system redirects the user to the “Wallet Details” screen of the wallet he/she clicked on and displays the list of the wallet’s transactions.

UC6: addCategory (Income/ Spending)

ACTOR

User

PRECONDITIONS

The user is signed-in.

MAIN FLOW

1. The use case starts when the user clicks on the settings icon on the top right of the screen and then “Configure Categories”.
2. The system redirects the user to the “Categories” screen.
3. The system displays spending categories in a list, under “Spending” in the top tab navigator.
4. The user can swipe left, or click on “Income” in the top tab navigator, to see the list of income categories.
5. The user clicks the “Add category” button while on the screen (Income/ Spending) of the category they want to add.
6. The system redirects the user to the “Add/Edit Category” screen featuring a “Name” text box.
7. The user enters the name of the category.
8. The user can choose an icon for the category from the icon list.
9. The user clicks the “Save” button.

POST-CONDITIONS

The user has successfully added a new category.

UC7: editCategory (Income/ Spending)

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one spending or income category.

MAIN FLOW

1. The use case starts when the user clicks on the settings icon on the top right of the screen and then “Configure Categories”.
2. The system redirects the user to the “Categories” screen.

3. The system displays spending categories in a list, under “Spending” in the top tab navigator.
4. The user can swipe left, or click on “Income” in the top tab navigator, to see the list of income categories.
5. The user clicks on the edit icon of the category he/she wants to edit.
6. The system redirects the user to the “Add/Edit Category” screen featuring a “Name” text box, filled with the current name of the category, and a list of icons.
7. The user changes the name and/or the icon of the category.
8. The user clicks on the “Save” button.

POST-CONDITIONS

The user has successfully changed the name and/or icon of a category.

UC8: deleteCategory (Income/ Spending)

ACTOR

User

PRECONDITIONS

The user is signed-in.

MAIN FLOW

1. The use case starts when the user clicks on the settings icon on the top right of the screen and then “Configure Categories”.
2. The system redirects the user to the “Categories” screen.
3. The system displays spending categories in a list, under “Spending” in the top tab navigator.
4. The user can swipe left, or click on “Income” in the top tab navigator, to see the list of income categories.
5. The user clicks on the delete icon of the category he/she wants to delete.
    5.1. If the category is not used in any transactions or budgets the system displays a pop-up question to the user “Are you sure you want to delete this category?”.

5.2. If the category is used in transactions or budgets the system displays a pop-up question “This category is already used in transactions or budgets, are you sure you want to delete it?”.

6. The user clicks on the “Yes” button.
7. The system deletes the category.

POST-CONDITIONS

The user has successfully deleted a category if it wasn't used in transactions or budgets, or hides it indefinitely if it was used.

UC9: createBudget

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one spending category, and one wallet.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The system displays all of the user’s budgets.
4. The user clicks the “New budget” on the top of the screen to add a budget.
5. The system redirects the user to the “Add/Edit Budget” screen.
6. The user sets the name of the new budget, in the “Budget name” text box.
7. The user chooses one or more of the available spending categories in the “Choose category/ies” drop-down menu.
8. The user chooses the last day of the budgeting period from the calendar.
9. The user enters the budget amount in the “Amount” text box.
10. The user can add a note to the “Note” text box.
11. The user clicks the “Save Budget” button.

POST-CONDITIONS

The user has created a new budget.

UC10: editBudget

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one spending category, and one wallet.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The system displays all of the user’s budgets.
4. The user clicks on the budget he/she wants to edit.
5. The system displays the “Budget Details Screen” for this budget.
6. The user clicks on the edit icon at the top right of the screen.
7. The system displays the “Edit Budget” screen.
8. The user can edit one or more of the following: budget name, the budget amount, end date, and note.
9. The user clicks on the “Save” button.
10. The system makes the changes and redirects the user to the “Budget Details” screen of this budget.

POST-CONDITIONS

The user has changed the name, date, amount, or note of the budget.

UC11: deleteBudget

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one spending category, and one wallet.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The system displays all of the user’s budgets.
4. The user clicks on the budget he/she wants to delete.
5. The system displays the “Budget Details” screen for this budget.
6. The user clicks on the delete icon at the top right of the screen.
7. The system displays a pop-up question to the user “ Are you sure you want to delete this budget?”.
8. The user clicks on the “Yes” button.
9. The system deletes the budget and redirects the user to the “Budgets” screen.

POST-CONDITIONS

The user deleted a budget.

UC12 : seeBudgets

ACTOR

User

PRECONDITIONS

The user is signed-in and has created at least one budget.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The system displays all of the user’s budgets.

UC13 : seeBudgetDetails

ACTOR

User

PRECONDITIONS

The user is signed-in and has created at least one budget.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The system displays all of the user’s budgets.
4. The user clicks on the budget.
5. The system redirects the user to the “Budget Details” screen and displays the following:
    5.1. A header with the budget name, the budget amount, the amount spent on this budget, and the balance amount.
    5.2. A card with the budget’s categories and a bar chart representation of the balance.
    5.3. A card with a line chart, featuring the budgeting period on the x-axis, and the money spent on the y-axis.
    5.4. A card with the budget’s transactions.

UC14: addTransaction (expense/income)

ACTOR

User

PRECONDITIONS

The User is signed in and has at least one category and one wallet.

MAIN FLOW

1. The use case starts when the user clicks the “+” button.
2. The system redirects the user to the “Add Transaction” screen and displays a top tab navigator featuring two tabs: “Expense” and “Income”.

2.1. If the user wants to add an expense transaction the system displays the add transaction form by default.

2.2. If the user wants to add an income transaction he/she swipes left or clicks on the “Income” tab of the top tab navigator and the system displays the add income form.

3. The user chooses the wallet of the transaction, from the “Choose wallet” drop-down menu.
4. The user enters the amount in the “Amount” text box.
5. The user can change the date which is auto-filled by the system.
6. The user can write a note in the “Note” text box.
7. If the user wants this transaction to be recurring:
   7.1. The user clicks on the “Recurring transaction” checkbox.
   7.2. The system displays a dropdown menu with the options: days, weeks, months, and a numeric textbox field, where the user can select the interval for the transaction.
8. The user clicks the “Save” button.

POST-CONDITIONS

The user has successfully added a new transaction, and the transaction's wallet balance has been updated. If it was an expense transaction, and the category belonged to a budget, then that budget’s balance is updated.

UC15: editTransaction

ACTOR

User

PRECONDITIONS

The User is signed in and has added some transactions.

MAIN FLOW

1. The use case starts when the user clicks the wallet icon at the bottom tab navigator.
2. The system displays the “Transactions” screen.
3. The user clicks on the transaction he/she wants to edit.
4. The system redirects the user to the “Edit Transaction” screen.
5. The system displays the transaction details.
6. The user can change the transaction amount through the “Amount” text box, and the transaction date through the date picker.
7. The user clicks on the “Save” button.

POST-CONDITIONS

The user has successfully edited a transaction. The respective wallet and budget, if there is one, are updated.

UC16: deleteTransaction

ACTOR

User

PRECONDITIONS

The User is signed in and has added some transactions.

MAIN FLOW

1. The use case starts when the user clicks the wallet icon at the bottom tab navigator.
2. The system displays the “Transactions” screen.
3. The user clicks on the transaction he/she wants to edit.
4. The system redirects the user to the “Edit Transaction” screen.
5. The system displays the transaction details.
6. The user clicks on the delete icon at the top right of the screen.
7. The system displays a pop-up question to the user “ Are you sure you want to delete this transaction?”.
8. The user clicks on the “Yes” button.
9. The system deletes the transaction and redirects the user to the “Transactions” screen.

POST-CONDITIONS

The user has successfully deleted a transaction. The respective wallet and budget, if there is one, are updated.

UC17: seeTransactionHistory

ACTOR

User

PRECONDITIONS

The User is signed in and has added some transactions.

MAIN FLOW

1. The use case starts when the user clicks the wallet icon at the bottom tab navigator.
2. The system displays the current month's transactions grouped by date from newest to oldest.

ALTERNATIVE FLOW 1: The user wants to see transaction history for a certain time period.

1. Steps 1-2 of the Main Flow.
2. The user clicks on the calendar icon at the top of the "Transactions" screen.
3. The system displays a modal featuring the following options: "This week", "This month", "This year", and "Custom Period", with "This month" being the default.
4. If the user clicks on "Custom Period", the system displays a form with two date pickers.
5. The user selects one of the options and clicks the ok button.
6. The system displays the transactions that correspond to the selected time period.

ALTERNATIVE FLOW 2: The user wants to see the upcoming transactions.

1. Steps 1-2 of the Main Flow.
2. The user clicks on the repeat icon at the top of the "Transactions" screen.
3. The system redirects the user to the "Upcoming Transactions" screen.
4. The system displays the upcoming transactions in chronological order.
5. If today is the due date of a transaction, a "mark as paid" button is displayed in the transaction's view box.
    5.1. If the user clicks this button, the system displays a modal with a question, asking the user to complete the transaction.
    5.2. If the user clicks "yes" the transaction is created, and added to the "Transactions" screen.
    5.3. The system computes the next transaction's date and adds it to the "Upcoming Transactions" screen.
6. The user can click on the "x" to dismiss the next upcoming transaction.

UC18: createGoal

ACTOR

User

PRECONDITIONS

The user is signed-in.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The user swipes left, or clicks on the “Goals” at the top tab navigator.
4. The system redirects the user to the “Goals” screen.
5. The user clicks on the “Create goal” button.
6. The system redirects the user to the “Add Goal” screen.
7. The user sets the name of the new goal, in the “Goal Name” text box.
8. If the user wants to set a target amount, he/she sets the amount in the “Target amount” text box.
9. If the user wants to set a due date for the goal, he/she chooses the desired day from the calendar.
10. The user can add a note to the “Note” text box.
11. The user can select an icon from the icon list for the goal.
12. The user clicks the “Save” button.

POST-CONDITIONS

The user has created a goal.

UC19: editGoal

ACTOR

User

PRECONDITIONS

The user is signed-in.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The user swipes left, or clicks on the “Goals” at the top tab navigator.
4. The system redirects the user to the “Goals” screen.
5. The user clicks on the goal he/she wishes to edit.
6. The system redirects the user to the “Goal Details” screen.
7. The user clicks on the edit icon at the top of the screen.
8. The user can change the name of the goal, at the “Goal Name” text box.
9. The user can change the amount in the “Target amount” text box.
10. The user can change the desired day from the calendar.
11. The user can add a note to the “Note” text box.
12. The user clicks the “Save” button.

POST-CONDITIONS

The user has made changes to a goal.

UC20 : seeGoalDetails

ACTOR

User

PRECONDITIONS

The user is signed-in and has created at least one goal.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The user swipes left, or clicks on the “Goals” at the top tab navigator.
4. The system displays all of the user’s goals.
5. The user clicks on the goal.
6. The system redirects the user to the “Goal Details” screen and displays the following:

6.1. A header with the goal name, the target date (if there is one), the amount added to this goal this month, and the other field dependent on the type of the goal:

6.1.1. If the goal has a target date and a target amount, the field is the minimum monthly amount that needs to be added in order to reach the goal in time.

6.1.2. If the goal has a target date but not a target amount, the field is the estimated savings at the target date taking into consideration the amount added this month.

6.1.3. If the goal has a target amount but not a target date, the field is the estimated time to reach the goal taking into consideration the amount added this month.

6.1.4. If the goal has neither a target date nor a target amount, the field is the estimated savings at the end of the year taking into consideration the amount added this month.

6.2. A progress circle for a visual representation of the goal's progress with a percentage in the middle and the actual saved and target amount (if it exists).

6.3. A "Add saved amount" button for updating the goal.

6.4. A "Set goal as reached" button.

UC21 : addSavedAmountToGoal

ACTOR

User

PRECONDITIONS

The user is signed-in and has created at least one goal.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the "Budgets" screen.
3. The user swipes left, or clicks on the "Goals" at the top tab navigator.
4. The system displays all of the user's goals.

5. The user clicks on the goal.
6. The system redirects the user to the “Goal Details” screen.
7. The user clicks on the “Add saved amount” button.
8. The system displays a pop-up with a dropdown menu featuring a “+” and a “-” and a text box.
9. The user sets the amount and chooses “+” for adding or “-” for subtracting.
10. The user clicks on the “check” icon.
11. The system redirects the user to the “Goals” screen featuring the updated goal.

POST-CONDITIONS

The user has changed the saved amount of a goal.

UC22: deleteGoal

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one goal.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The user swipes left, or clicks on the “Goals” at the top tab navigator.
4. The system displays all of the user’s goals.
5. The user clicks on the goal he/she wants to delete.
6. The system displays the “Goal Details” screen for this goal.
7. The user clicks on the delete icon at the top right of the screen
8. The system displays a pop-up question to the user “ Are you sure you want to delete this goal?”.
9. The user clicks on the “Yes” button.
10. The system deletes the budget and redirects the user to the “Goals” screen.

POST-CONDITIONS

The user deleted a goal.

UC23: setGoalAsReached

ACTOR

User

PRECONDITIONS

The user is signed-in and has at least one goal.

MAIN FLOW

1. The use case starts when the user clicks the envelope icon at the bottom tab navigator.
2. The system redirects the user to the “Budgets” screen.
3. The user swipes left, or clicks on the “Goals” at the top tab navigator.
4. The system displays all of the user’s goals.
5. The user clicks on the goal he/she wants to set as reached.
6. The system displays the “Goal Details” screen for this goal.
7. The user clicks on the “set goal as reached” button at the end of the screen.
8. The system displays a pop-up question to the user “Are you sure you want to set this goal as reached?”.
9. The user clicks on the “Yes” button.

POST-CONDITIONS

The goal can now be seen by the user on the ReachedGoals screen.

UC24: seeMoneyAllocationInPieCharts

ACTOR

User

PRECONDITIONS

The user is signed-in and has added some income and some expense transactions.

MAIN FLOW

1. The use case starts when the user clicks the pie chart icon at the bottom tab navigator.
2. The system redirects the user to the “Charts” screen.

3. The system displays an overview of the month's income, expenses, and balance in the header and two cards featuring the spending and income money allocation in pie charts.
4. The user can click on the calendar icon and select a date range for which he/she wants to see the overview and pie charts.

POST-CONDITIONS

The user views his/her income and spending allocation in pie charts for a date range he/she selected.

UC24: seeReportsOfThePreviousMonths

ACTOR

User

PRECONDITIONS

The user is signed-in and has added some income and some expense transactions during the previous month.

MAIN FLOW

1. The use case starts when the user clicks the pie chart icon at the bottom tab navigator.
2. The system redirects the user to the "Charts" screen.
3. The user swipes left or clicks on the "Reports" tab on the top tab navigator.
4. The system displays a matrix with the income and expense amounts of each of the previous months.
5. The user clicks on the month for which he/she wants to see details.
6. The system redirects the user to the selected month's "Monthly Report" screen and displays a matrix with the user's categories, their total amount for the month, and a percentage compared to the amounts of the previous month.

POST-CONDITIONS

The user views his/her income and spending amounts per category for each of the previous months.

5. Design and Implementation

5.1. Database

Relational database is a type of database that stores and organizes data points with defined relationships for fast access. A relational database organizes data into tables containing information about each entity and representing pre-defined categories through rows and columns. Structuring data this way makes it efficient and flexible to access [13]. Since this application has entities like transactions, budgets, categories, wallets, and goals that are dependent on each other a relational database is the best way to approach data management. In particular, this thesis uses MySQL as the database engine. This section explains the database structure. The database schema is shown in Figure 1.

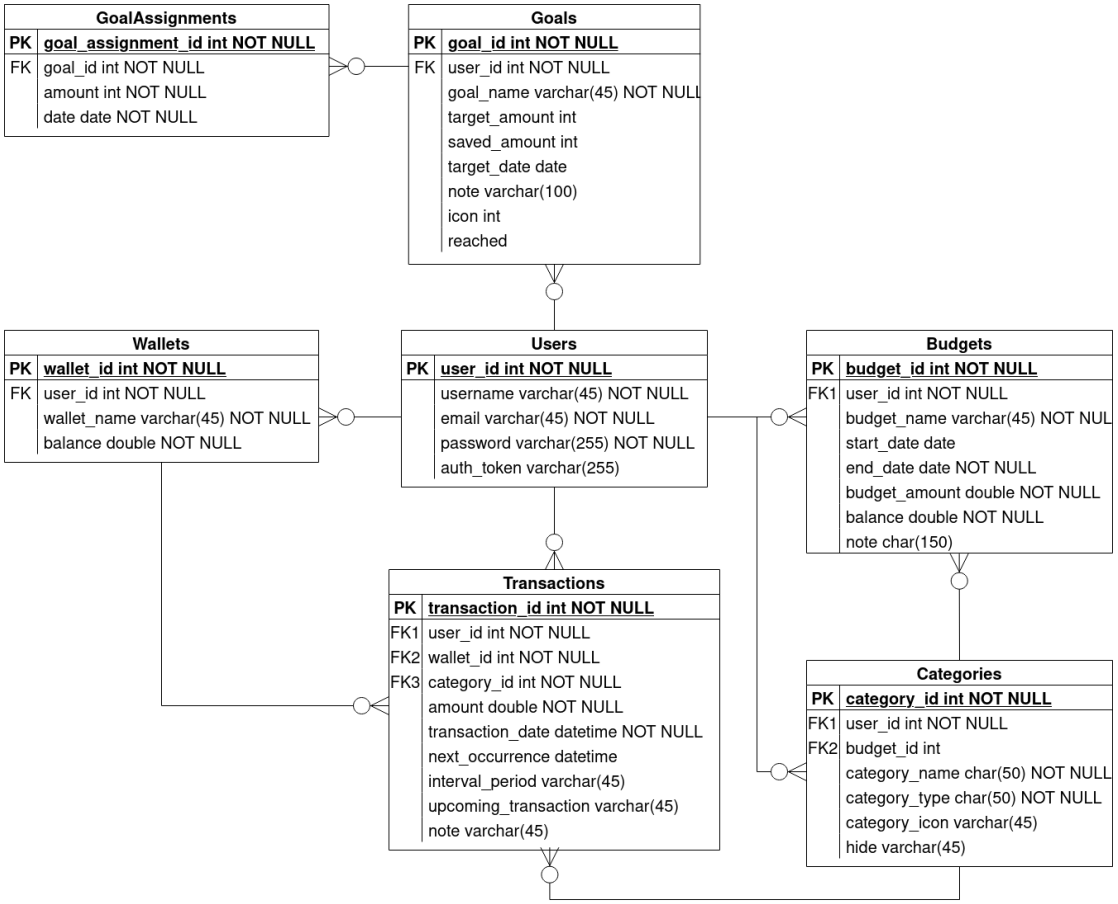

Figure 1. MySQL Database Schema

5.1.1. Database Tables

The primary key of each table is an auto-increment integer named after the table. All entities, other than User, and GoalsAssignment can only exist in relation to a User entity. For that reason, every table has a *user_id* field which is a foreign key referencing the Users' *user_id* column.

Users

This table saves the user information. Additionally to the user information, the table has an *auth_token* field. For the authentication token, JWT (JSON Web Token)[14] is used. Typically, an authentication token is saved locally, and not on the database. Because currently, the application does not have some kind of local storage, centralized storage is used.

Categories

This table saves the spending and income categories. There are some default categories in every user account, and each user can add more or edit them to their preference. This table also has a *budget_id* field which is a foreign key referencing the Budgets' *budget_id* column. If a category does not belong to any budget, the field is null.

Wallets

This table saves the user's wallets and their balance.

Transactions

This table saves the information about transactions. The fields *wallet_id* and *category_id* are foreign keys to the Wallets' *wallet_id* column and the Categories' *category_id* column. Since the application supports recurring transactions, there are some fields in this table about them. The *interval_period* field is of varchar datatype and has the following format: "XY", where X is an integer and Y is one of the following: D (days), W (weeks), or M (months). From the *transaction_date* and *interval_period* fields, the *next_occurrence* field is calculated. When the *next_occurence* date arrives, and the user makes the transaction, a new row is written to the table and its *next_occurrence* is calculated. In order to differentiate the transactions that have already happened is to set their *upcoming_transaction* field to false.

Budgets

This table saves information about budgets. A budget has a name (*budget_name*), a

budget amount (*budget_amount*), a start date (*start_date*), an end date (*end_date*), and one or more categories. Having the *budget_id* in the Categories table instead of having the *category_id* in the Budgets table ensures that a budget can have more than one category while avoiding data repetition in the Budgets table. When the budget is created the *balance* is equal to the *budget_amount*, and it gets updated every time the user makes a transaction during the budget's period with the budget's category/categories.

Goals

This table saves the information about saving goals. A goal's target amount (*target_amount*) and target date (*target_date*) are optional fields, since not all saving goals are amount and time specific. The amount that the user has currently saved is in the *saved_amount* field.

GoalAssignments

This table saves each goal assignment as a different record with *goal_id* as a foreign key to the Goals' *goali_id* column, an *amount,* and a *date* field.

5.2. Server

The Express server offers access to a set of resources. An HTTP request is made by the client, the mobile application, in which the requested resource is specified along with the parameters. The server responds to the application with an HTTP response, whose aim is to provide the client with the resource it requested or inform the client that the action it requested has been carried out, or inform the client that an error occurred in processing its request. [15]

HTTP Requests

The HTTP requests describe the action that needs to be taken on a specific resource. The most commonly used HTTP actions are the following:

- GET: used to retrieve data
- POST: used to send new data
- PUT: used to modify data
- PATCH: used to partially modify data
- DELETE: deletes specified data

We use Axios library for sending and receiving HTTP requests. The request consists of an

action method (post, get, etc.), a URL, and sometimes a request body, and/or a request header. In our server, a POST request is used in some cases to retrieve data when there is a request body since a GET request cannot have a request body.

API

API stands for Application Programming Interface. It is a software intermediary that allows the server and the client to communicate with each other and exchange information. For example, if a user requests a specific transaction from the server, the server will need an API in order to handle the request. The server needs to determine how to respond to various requests for specific resources. In this case, return the transaction information.

Structure

The server uses the Model/Router/Controller structure, which is described in this section.

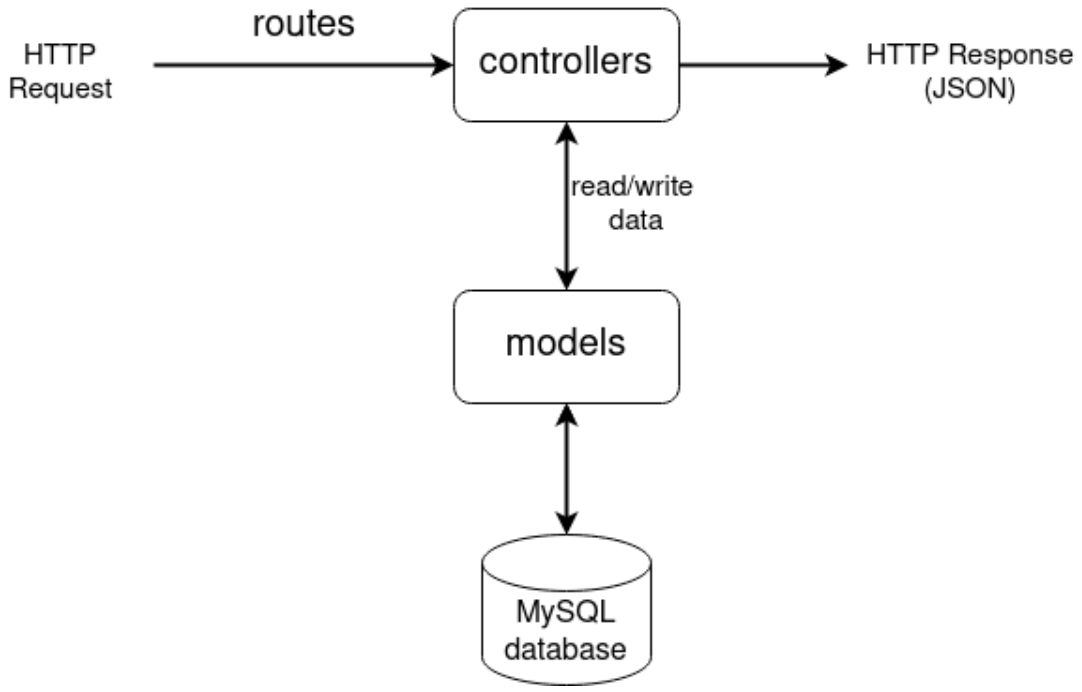

Figure 2. Server - MySQL Database

5.2.1. Models

The model is a layer between the server and the database [16]. In the db.js file, the

connection with the database is established. The models correspond to the application's entities and to the database's tables and are the following: user, wallet, category, transaction, budget, and goal. The model functions communicate with the database by making SQL queries and receiving the response from SQL.

5.2.2. Routes

A route is a section of Express code that associates an HTTP verb (GET, POST, PUT, DELETE, etc.), a URL path, and a controller function that is called to handle that pattern. [17]. Each file of the routes folder contains the paths for the resources of each model (user, wallet, category, transaction, budget, goal).

User routes

Table 9 shows the user routes and their controller functions.

HTTP verb	URL path	Controller function
post	/user	createUser
post	/signin	userSignin
post	/signout	userSignout
delete	/delete-user	deleteUser

Table 9. User routes

Wallet routes

Table 10 shows the wallet routes and their controller functions.

HTTP verb	URL path	Controller function
post	/wallet	createWallet
get	/wallet	getAllWallets
get	/wallet/:wallet_id	getWalletById
put	/wallet/:wallet_id	updateWallet
patch	/wallet/:wallet_id	updateBalance

delete	/wallet/:wallet_id	deleteWallet

Table 10. Wallet routes

Category routes

Table 11 shows the category routes and their controller functions.

HTTP verb	URL path	Controller function
post	/category	createCategory
get	/category/income	getIncomeCategories
get	/category/spending	getSpendingCategories
get	/category/from/:category_name	getCategoryFromName
get	/category/:category_id	getCategoryById
put	/category/:category	updateCategory
patch	/category/budget	updateBudget
patch	/category/hide/:category_id	hideCategory
delete	/category/:category_id	deleteCategory

Table 11. Category routes

Transaction routes

Table 12 shows the transaction routes and their controller functions.

HTTP verb	URL path	Controller function
post	/transaction	createTransaction
post	/transaction-dailysum	getDailyTotalSpending
post	/transaction/get-from-list-of-categories	getTransactionsByCategories

post	/transaction-sum	getTransactionsSum
post	/transaction-by-date-range	getTransactionsByDateRange
get	/transaction	getAllTransactions
get	/transaction-recurring	getAllRecurringTransactions
get	/transaction/:transaction_id	getTransactionById
get	/transaction-wallet/:wallet_id	getTransactionsByWalletId
patch	/transaction/next-occurrence/:transaction_id	updateNextOccurence
patch	/transaction/:transaction_id	updateTransaction
patch	/transaction/upcoming/:transaction_id	updateUpcoming
delete	/transaction/:transaction_id	deleteTransaction

Table 12. Transaction routes

Budget routes

Table 13 shows the budget routes and their controller functions.

HTTP verb	**URL path**	**Controller function**
post	/budget	createBudget
get	/budget	getAllBudgets
get	/budget/:budget_id	getBudgetById
put	/budget/:budget_id	updateBudget
patch	/budget/:budget_id	updateBalance
delete	/budget/:budget_id	deleteBudget

Table 13. Budget routes

Goal routes

Table 14 shows the goal routes and their controller functions.

HTTP verb	URL path	Controller function
post	/goal	createGoal
get	/goal	getAllGoals
get	/goal/:goal_id	getGoalById
put	/goal/:goal_id	updateGoal
patch	/goal/:goal_id	updateSavedAmount
delete	/goal/:goal_id	deleteGoal
post	/goal-assignment/:goal_id	assignToGoal
post	/goal-assignment/monthly-sum/:goal_id	getMonthlySum

Table 14. Budget routes

5.2.3. Controllers

The controller is the intermediate between the route and the model. There are six controllers one for each model (user, category, wallet, transaction, budget, goal). Each controller is responsible for ensuring that the application's request data are valid and if they are the controller forwards the request to the respective model. After receiving the response from the model the controller responds to the application with a JSON document including the requested resource and a status code of 200. When the request data are not valid the controller responds with a JSON document including a relevant message and a status code of 400.

1. User controller functions

createUser	gets from the request body the following data: *email*, *username*, *password*, and *confirmPassword*. Firstly it ensures that this *email* is not already used by calling the findByEmail method of the user model. If the *email* is found in the database the controller responds to the application with a JSON document including the message "This email is already in use, try sign-in". If the *email* is not found in the database, the user's *password* is hashed using the bcrypt library. Finally, the createUser calls the create method of the user model, sending the *email*, the *hashedPassword*, and the *username* as parameters. The controller responds with a JSON document including a *success: true* object, if the creation was successful, or a *message* describing the error.
userSignin	gets from the request body the user's *email*, and *password*. First, it checks if the *email* exists in the database, and if it is not, responds with a JSON document including the message "User not found with this email". After finding the email in the database, it compares the hashed password from the database with the *password* from the request body using bcrypt's compare method. If the passwords are the same, an authentication token is generated using JSON Web Token's sign method. The authentication token is then saved in the database with the saveToken method of the user model. Finally, the authentication token along with the user's id, email, and username is returned to the application inside a JSON document.
deleteUser	is a method made for facilitating the testing process. The method finds the user in the database and deletes the user by calling the deleteUser method of the user model.

Table 15. User controller functions

2. Wallet controller functions

createWallet	gets the *walletName* and *balance* parameters from the request

	body, and the authentication token from the request header. Using the JSON Web Token's verify method, it extracts the user id and forwards it along with *walletName* and *balance* to the create method of the wallet model. An OkPacket is returned to the application to signal the successful completion of the query.
getWalletById	gets the *walletId* from the request parameters, converts it to an integer, and forwards it to the getById method of the wallet model. If the *walletId* is found the controller responds with a JSON document including the wallet record.
getAllWallets	gets the user authentication token from the request header, extracts the user id, and forwards it to the getAll method of the wallet model. The controller then responds to the application with a JSON document including an object with all the user's wallet records.
updateWallet	gets the *walletName* and *balance* from the request body and the *walletId* from the request parameters. If the mandatory fields (*walletName, balance*) are not null, the method forwards them along with the *walletId* to the update method of the wallet model. An OkPacket is returned to the application to signal the successful completion of the query.
updateBalance	gets the *balance* from the request body and the *walletId* from the request parameters. If the *balance* is not null, the method forwards the fields to the updateBalance method of the wallet model. An OkPacket is returned to the application to signal the successful completion of the query.
deleteWallet	gets the user authentication token from the request header, extracts the user id, and forwards it to the delete method of the wallet model. An OkPacket is returned to the application to signal the successful completion of the query.

Table 16. Wallet controller functions

3. Category controller functions

createCategory	gets the *categoryName*, *categoryType,* and *icon* parameters from the request body, and the authentication token from the request header. Using the JSON Web Token's verify method, it extracts the user id. If the *categoryName* and *categoryType* are not null the method forwards them along with the *icon* to the create method of the category model. An OkPacket is returned to the application to signal the successful completion of the query.
getCategoryById	gets the *categoryId* from the request parameters, converts it to an integer, and forwards it to the getById method of the category model. If the categoryId is found the controller responds with a JSON document including the category record.
getSpendingCategories/ getIncomeCategories	gets the user authentication token from the request header, extracts the user id, and forwards it to the getAllSpending/ getAllIncome method of the category model. The controller then responds to the application with a JSON document including an object with all the user's category records.
getCategoryFromName	gets the *categoryName* from the request parameters, and the *userId* from the request header and forwards them to the getCategoryFromName method of the category model. If the category is found the controller responds with a JSON document including the category record
updateCategory	gets the *categoryName* and *icon* from the request body and the *categoryId* from the request parameters. If *walletName* is not null, the method forwards them along with the *walletId* and *icon* to the update method of the category model.
updateBudget	is called after the user creates a budget in order to update a category with a budget, and is called as many times as the number of categories selected by the user for a specific budget. The method gets the *categoryId* and *budgetId* from the request body. If the

	fields are not null, the method forwards them to the updateBudget method of the category model.
hideCategory	is called when the user wants to delete a spending or income category that is already used in a transaction or budget in order to change the *hide* field to true, and not show that category again to the user to use. If a category like this were to be deleted, it would break the budget and transaction because those cannot exist without a category. The method gets the *categoryId* from the request parameters and forwards it to the hideCategory method of the category model.
deleteCategory	gets the user authentication token from the request header, extracts the user id, and forwards it to the delete method of the category model. An OkPacket is returned to the application to signal the successful completion of the query.

Table 17. Category controller functions

4. Transaction controller functions

createTransaction	gets the *walletId*, *categoryId, transactionAmount, transactionDate, nextOccurrence, interval, transactionNote,* and *upcoming* parameters from the request body, and the authentication token from the request header. Using the JSON Web Token's verify method, it extracts the user id. If the *walletId*, *categoryId, transactionAmount* are not null the method forwards them along with the other fields and to the create method of the transaction model. An OkPacket is returned to the application to signal the successful completion of the query.
getDailyTotalSpending	is used for the spending amount chart in each budget. The method gets the budget's date range and the list of category id's of the budget, extracts the user id from the user authentication token from the request header and if the fields are not null, forwards them to the getDailyTotalSpending method of the transaction

	model. The method responds with a JSON document including the total spending amount of each date in the date range only for the spending categories of the request.
getTransactionsByCategories	is used for receiving the transactions of a budget. The method gets the budget's date range and the list of category id's of the budget, from the request body, and if the fields are not null, forwards them to the getTransactionsByCategories method of the transaction model. The method responds with a JSON document including the budget's transactions.
getTransactionsSum	gets a date range selected by the user from the request body, and if the fields are not null, forwards them to the getTrasnactionsSum method of the transaction model. The method responds with a JSON document including the total amount spent or received for each of the user's categories in a specific date range. That object is used to make spending categories and income categories pie charts.
getTransactionsByDateRange	gets a date range selected by the user from the request body and extracts the user id from the user authentication token from the request header. If the date range is not null, the method forwards it along with the user id to the getTransactionsByDateRange method of the transaction model. The method responds with a JSON document including an object with all the user's transaction records in that date range and is used to filter the user's transactions by date range.
getAllTransactions	gets the user authentication token from the request header, extracts the user id, and forwards it to the getAll method of the transaction model. The controller then responds to the application with a JSON document including an object with all the user's transaction records.
getAllRecurringTransactions	gets the authentication token from the request header and extracts the user id. Then, the method forwards the user id to the getAllRecurring method of the transaction model. The controller

	then responds to the application with a JSON document including an object with all the user's current upcoming transaction records.
getTransactionById	gets the transactionId from the request parameters, converts it to an integer, and forwards it to the getById method of the transaction model. If the transactionId is found the controller responds with a JSON document including the transaction record.
getTransactionsByWalletId	gets the *walletId* from the request parameters and the user's authentication token from the request header and extracts the user id. The method forwards the fields to the getTransactionsByWallet method of the transaction model. The controller then responds to the application with a JSON document including an object with all the user's transaction records from a specific wallet.
updateNextOccurence	is called when the user dismisses an upcoming transaction in order to change the next occurrence. The method gets the *nextOccurence* from the request body and the *transactionId* from the request parameters. If *nextOccurence* is not null, the method forwards them along with the *transactionId* to the updateNextOccurrence method of the transaction model.
updateTransaction	is called when the user edits a transaction. The method gets the *transactionAmount* and/or *transactionDate* and/or *transactionNote* and/or *upcoming* from the request body, and the *transactionI*d from the request parameters and forwards them to the update method of the transaction model. An OkPacket is returned to the application to signal the successful completion of the query.
updateUpcoming	After the user confirms paying or receiving an upcoming transaction (old transaction), a new transaction record with its *upcoming* field set to true is created. This method's responsibility is to change the *upcoming* field of the old transaction to false so that this old upcoming transaction will no longer appear in the upcoming transactions. The method gets the *nextOccurence* from

	the request body and the *transactionId* from the request parameters. If *nextOccurence* is not null, the method forwards them along with the *transactionId* to the updateNextOccurrence method of the transaction model. An OkPacket is returned to the application to signal the successful completion of the query.
deleteTransaction	gets the user authentication token from the request header, extracts the user id, and forwards it to the delete method of the transaction model. An OkPacket is returned to the application to signal the successful completion of the query.

Table 18. Transaction controller functions

5. Budget controller functions

createBudget	gets the *budgetName*, *budgetAmount, budgetBalance, startDate, endDate,* and *budgetNote* parameters from the request body, and the authentication token from the request header. Using the JSON Web Token's verify method, it extracts the user id. If the *budgetName*, *budgetAmount, and endDate* are not null the method forwards them along with the others and to the create method of the budget model. An OkPacket is returned to the application to signal the successful completion of the query.
getAllBudgets	gets the user authentication token from the request header, extracts the user id, and forwards it to the getAll method of the budget model. The controller then responds to the application with a JSON document including an object with all the user's budget records.
getBudgetById	gets the *budgetId* from the request parameters, converts it to an integer, and forwards it to the getById method of the budget model. If the *budgetId* is found the controller responds with a JSON document including the budget record.
updateBudget	gets the budget*Name, budgetAmount, budgetBalance, endDate,* and

	budgetNote from the request body, and the *budgetId* from the request parameters. If the *budgetName, budgetAmount, budgetBalance, and endDate* are not null, the method forwards them along with the other fields to the update method of the budget model. An OkPacket is returned to the application to signal the successful completion of the query.
updateBalance	is called after the user adds, updates, or deletes a transaction in order to update the budget's balance. The method gets the *balance* from the request body and the *budgetId* from the request parameters. If *balance* is not null, the method forwards them along with the *budgetId* to the updateBalance method of the budget model.
deleteBudget	gets the user authentication token from the request header, extracts the user id, and forwards it to the delete method of the budget model. An OkPacket is returned to the application to signal the successful completion of the query.

Table 19. Budget controller functions

6. Goal controller functions

createGoal	gets the *goalName*, *targetAmount, savedAmount, targetDate,* and *icon* parameters from the request body, and the authentication token from the request header. Using the JSON Web Token's verify method, it extracts the user id. If the *goalName* is not null the method forwards them along with the others and to the create method of the budget model. An OkPacket is returned to the application to signal the successful completion of the query.
getAllGoals	gets the user authentication token from the request header, extracts the user id, and forwards it to the getAll method of the goal model. The controller then responds to the application with a JSON document including an object with all the user's goal records.

getGoalById	gets the *goalId* from the request parameters, converts it to an integer, and forwards it to the getById method of the goal model. If the goal is found the controller responds with a JSON document including the goal record.
updateGoal	is called when the user edits a goal and gets the *goalName, targetAmount, savedAmount, targetDate, note,* and *icon* from the request body, and the *goalId* from the request parameters. If the *goalName* is not null, the method forwards it along with the other fields to the update method of the goal model. An OkPacket is returned to the application to signal the successful completion of the query.
deleteGoal	gets the user authentication token from the request header, extracts the user id, and forwards it to the delete method of the goal model. An OkPacket is returned to the application to signal the successful completion of the query.
assignToGoal	is called when the user adds or abstracts an amount from a goal. Different assignment records are saved to the GoalAssignments table instead of just updating the *savedAmount* of the Goals table, for the purpose of keeping track of monthly assignments. We keep track of them in order to show the user some insights about his/her goals. The method gets the *amount* from the request body and the *goalId* from the request parameters and forwards it to the createAssignment method of the goal model. An OkPacket is returned to the application to signal the successful completion of the query.
getMonthlySum	calculates and returns the total amount assigned to a goal for a specific month. The method gets the *start* and *end* dates from the request body and the *goalId* from the request parameters and forwards them to the getMonthlySum method of the goal model. The method responds with a JSON document including the total amount.

Table 20. Goal controller functions

5.3. Application

In this chapter, we present some of the React Native basic concepts, and then we analyze the application's different parts.

5.3.1. React Native Basic Concepts

Native Components

Components are considered the core building blocks of a React application. With React Native, we can render views with JavaScript using React components. Components facilitate the task of building UIs. They return a piece of JSX code that tells what should be rendered on the screen. [18]

At runtime, React Native creates the corresponding Android and iOS views for those components. We call these platform-backed components Native Components. React Native has many Core Components. In this application we are mostly working with the following Core Components: [19]

RN UI Component	**Description**
<View>	The basic building block of UI. A container that supports layout with flexbox, style, some touch handling, and accessibility controls
<Text>	Displays, styles, and nests strings of text and even handles touch events
<Image>	Displays different types of images
<ScrollView>	A generic scrolling container that can contain multiple components and views
<TextInput>	Allows the user to enter text
<TouchableOpacity>	A wrapper for making views respond properly to touches. On press down, the opacity of the wrapped view is decreased,

	dimming it.
<FlatList>	A performant interface for rendering basic, flat lists.

Table 21. React Native UI Components

Additionally, we use many Community Components like icons from *expo/vector-icons*, chart components from *react-native-svg-charts, react-native-svg* and *react-native-progress*, and a date-time picker from *react-native-community.*

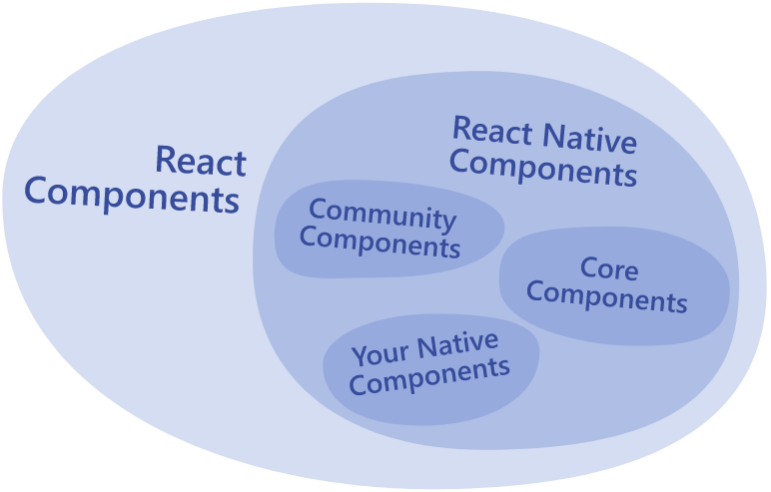

Figure 3. React Components (from React Native Docs)

Our Native Components

Our Custom Components are the backbone of the application and are divided into two folders. The *screens* folder contains the screen Custom Components, and the *components* folder contains the reusable Custom Components of our application. The *screen* Components use our reusable *components*.

Props

Props is short for Properties. The components can be customized at the time of creation using different parameters, and those parameters are known as props. Props are passed from one component to another as a means of passing data between them. We can think of props as arguments we use to configure how our components render. [18]

State

To build an interactive application with React Native we need state. State is useful for

handling data that changes over time or that comes from user interaction. We can think of state as a component's personal data storage. We add state to a component by calling React's useState Hook. A Hook is a kind of function that lets us "hook into" React features. For example, useState is a Hook that lets us add state to function components. [18]

Redux

Redux [23] is a state management library for JavaScript applications. Redux helps manage state but the problem it solves is data flow. React has a one-way data flow as data is passed down the component tree via props and to pull the data up in the tree needs a callback function, which means a callback function must be passed down to any component that wants to pass data up the component tree. As application size increases it becomes unmanageable to handle the data, here comes Redux to solve the problem. Redux revolves around three core concepts:

1. There is a single source of truth. Application data is represented by a single JavaScript object, known as state of the application.
2. State (data) is immutable. State cannot be manipulated directly. It can only be changed by dispatching an action. *Action* is an object with information about the type of an event, and it may also contain data that needs to be updated.
3. Changes are made through pure functions known as *Reducers*. Reducer is basically a big switch statement that has a case for every action type and returns a new state without impacting the original state.

5.3.2. Folder Structure

The source code of the application consists of six parts: redux, navigators, services, utils, components, and screens. In this section, each of the source code parts is explained.

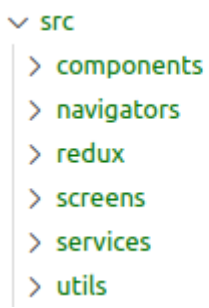

Figure 3. Application's source code folder structure

5.3.2.1. Screens

Our screens folder includes the components that correspond with the pages the user sees in our application, and every screen component is rendered by a navigator component. In table 22 we see the application's screen components grouped by entities.

Folder	**Screen components**
auth	SignupScreen, SigninScreen
category	CategoriesScreens, AddEditCategoryScreen
wallet	WalletsScreen, AddEditWalletScreen, WalletDetailsScreen
transaction	TransactionsScreen, UpcomingTransactionsScreen, AddTransactionScreen, EditTransactionScreen
budget	BudgetsScreen, AddEditBudgetScreen, BudgetDetailsScreen
goal	GoalssScreen, AddEdiGoalScreen, GoalDetailsScreen
insights	ChartsScreen, ReportsScreen, MonthlyReportScreen
	HomeScreen
	SettingScreen

Table 22. Screen components

5.3.2.2. Navigators

Routing and navigation refer to organizing an app into distinct screens, mapping screens to URLs, moving between those screens, and displaying the appropriate user interface elements. In this application React Navigation is used for managing the presentation, and transition between multiple screens, and it is the most popular navigation library in the React Native ecosystem. [20]

The application's navigation consists of Stack Navigators, Bottom Tab Navigators, and Material Top Tab Navigators. Stack Navigator renders one screen at a time and provides transitions between screens. When a new screen is opened it is placed on top of the stack. Bottom Tab Navigator renders a tab bar that lets the user switch between several

screens. Finally, the Material Top Tab Navigator renders a tab view which lets the user switch between several screens using a swipe gesture or the tab bar. [21]

Every screen in the application needs to be rendered by a navigator. Each screen component is provided with the navigation prop automatically. The prop contains various convenience functions that dispatch navigation actions. The most frequently used of these functions is the navigate() function which navigates the user to another screen. This folder contains the different navigators used in the application.

Navigator	**Responsibilities**
RootProvider	The RootProvider uses the Provider component and makes the Redux store available to any nested components that need to access the Redux store. The RootProvider is not technically a navigator but it is needed in order to render the Provider at the top level and inside the Provider it renders the RootStackNavigator of the application
RootStackNavigator	RootStackNavigator renders the application's NavigationContainer. The NavigationContainer is responsible for managing the application's state and linking the top-level navigator to the application environment. Using the store's *isLoggedin* state, the RootStackNavigator renders the AuthNavigator if *isLoggedin* is false, or the AppNavigator if *isLoggedin* is true.
AuthNavigator	The AuthNavigator is a Stack Navigator that renders two screens. The SignupScreen and the SigninScreen.
AppNavigator	The AppNavigator is the application's main navigator that renders the BottomTabNavigator, the SettingsNavigator, the AddTransactionNavigator, and all the screens that are not rendered by the previous three navigators: AddEditBudgetScreen, AddEditGoalScreen, AddEditWalletScreen, EditTransactionScreen, BudgetDetailsScreen, WalletDetailsScreen, GoalsDetailsScreen,

	RecurringTransactionsScreen, and MonthlyReportScreen
BottomTabNavigator	The BottomTabNavigator renders the HomeScreen, the EnvelopeNavigator, the WalletNavigator, and the InsightsNavigator.
SettingsNavigator	The SettingsNavigator is a Stack Navigator that renders the SettingsScreen, the CategoriesNavigator and the AddEditCategoryScreen.
CategoriesNavigator	The CategoriesNavigator is a Material Top Navigator that renders the CategoriesScreen, once for the spending categories on the left and once for the income categories on the right.
EnvelopeNavigator	The EnvelopeNavigator is a Material Top Navigator that renders the BudgetScreen on the left and the GoalsScreen on the right.
WalletsNavigator	The WalletNavigator is a Material Top Navigator that renders the TransactionsScreen on the left and the WalletsScreen on the right.
InsightsNavigator	The InsightsNavigator is a Material Top Navigator that renders the ChartsScreen on the left and the ReportsScreen on the right.

Table 23. Navigators and their responsibilities

5.3.2.3. Components

The components folder contains all the reusable components of the application. The folder structure (figure 5) is inspired by the Atomic Design, but not followed strictly. Atomic design is a methodology for designing and developing user interfaces in a modular manner by putting the focus on building components rather than applications. It helps to build consistent, solid, and reusable design systems. In his search for inspiration and parallels, Brad Frost the writer of Atomic Design kept coming back to chemistry. The thought is that all matter is composed of atoms. Those atomic units bond

together to form molecules, which in turn combine into more complex organisms to create all matter in our universe ultimately. Similarly, interfaces are made up of smaller components. This means we can break entire interfaces into fundamental building blocks and work from there. That's the basic gist of atomic design. [22]

The organization of this application's components consists of atoms, molecules, and organisms that do not strictly follow the definitions of Atomic Design. Initially, we define the atoms, molecules, and organisms in this context, and finally, we analyze each component of the application.

Atoms

"Atoms are the basic building blocks of matter. Applied to web interfaces, atoms are our tags, such as a form label, an input, or a button. Atoms can also include more abstract elements like color palettes." In this structure Atoms are the least complex custom Components, meaning that they only consist of Core and Community components, and not other custom Components. In our application, the atoms are divided into buttons, texts, charts, inputs, and other uncategorized atoms and files.

The **buttons** folder contains all the button components used in the application. For the buttons, we use the TouchableOpacity core component of RN. For the text buttons, we use the Text core component. For the icon buttons, we use icons from the expo vector-icon library. The buttons include the following components: *AddEntityButton*, *AddTransactionButton*, *FormButton*, *CalendarModalDateButton*, *BudgetOverspendButton*, *CalendalButton*, *CategoryIconButton*, *EditButton*, *DeleteButton*, NotificationButton, *RecurringTransactionsButton*, *GoBackButton*, *SettingsButton*, *MarkAsPaidButton*, *DismissButton*, *AuthFormFooter*.

The **texts** folder contains little pieces of text like titles, footers, and box containers that are frequently used in the application. Usually, they consist of View and Text core components and some contain Image, TouchableOpacity, and vector-icon components. The texts components include the following components: *HeaderTitle*, *EntityTitle*, *TransactionText*, *TransactionAmount*, *DotAndCategoryName*, *BoxItem*, *UpcomingTransactionFooter*.

The **charts** atoms contain the charts of the application. For the charts we used components from react-native-progress, react-native-svg, react-native-svg-charts and d3-shape. Our charts include the following components: *CustomProgressCircle*,

BudgetProgressBar, *GoalProgressBar*, *CustomLineChart*, *CustomPieChart*.

The **inputs** atoms are components used to form inputs and are the following: *CustomTextInput*, *CustomDropdown* using react-native-element-dropdown components, *CustomDatePicker* using react-native-community/datetimepicker component.

Other atoms include the following: *CategoryIconForDisplay*, *colors,* and *categoryIcons*.

There is a more detailed description of the atom components in the Appendix.

Molecules

"Molecules are groups of atoms bonded together and are the smallest fundamental units of a compound. While molecules can be complex, as a rule of thumb they are relatively simple combinations of atoms built for reuse." In this structure, Molecules consist of Core and Community components, Atoms, and other Molecules.

The **listItem** molecules render a View of an entity item, and they include one or more of the following methods: *onPress*, *onPressEdit*, and *onPressDelete*. The *onPress* methods are called when the "item" is pressed. Those methods dispatch the entity's actions and sometimes make other APIRequest calls before they navigate the user to the entity details screen. The *onPressEdit* methods are called when the edit icon, if there is one, is pressed, and their responsibility is to dispatch the entity's actions that set their fields and navigate to the edit screens. Finally, the *onPressDelete* methods are called when the delete icon, if there is one, is pressed, and their responsibility is to call the respective APIRequestDelete-entity- method. The listItem components include the following components: *CategoryListItem*, *WalletListItem*, *TransactionListItem*, *UpcomingTransactionListItem*, *BudgetListItem*, *GoalListItem*, *WalletListItemForCard*, *TransactionListItemForCard*, *BudgetListItemForCard*, *GoalListItemForCard*.

Other molecules include *BudgetBarChart*, *LabeledInput*, *Transaction*, *CalendarModalCustomDate*, *CalendarModalDateRangeButtons*.

There is a more detailed description of the molecule components in the Appendix.

Organisms

"Molecules give us some building blocks to work with, and we can now combine them together to form organisms. Organisms are groups of molecules joined together to form a relatively complex, distinct section of an interface." In this structure, Organisms consist of Core and Community components, Atoms, and Molecules.

The **chartCards** organisms include the *PieChart* component, featuring a card-like view with a category pie chart.

The **headers** organisms include all the header components of the application and are the following components: *Header*, *HeaderTabNavigator*, *HeaderWithDateRange*, *BudgetHeader*, *GoalHeader*, and *WalletHeader.*

The **homeScreenCards** organisms include all the card-like views of the HomeScreen. They are the following: *SummaryCardForHome*, *WalletsCardForHome*, *TransactionsCardForHome*, *BudgetsCardForHome*, *GoalsCardForHome.*

The **lists** organisms render a Flatlist component. A Flatlist component receives from its props a list and a renderItem method that renders each item of the list. Our lists organisms include *CategoryList*, *WalletList*, *TransactionList*, *BudgetList*, and *GoalList.*

The **modals** organisms use the Modal, Portal, and Provider components of the react-native-paper and are the *CalendarModal*, and the *AddSavedAmountModal.*

There is a more detailed description of the organism components in the Appendix.

5.3.2.4. Redux

The whole global state of the app is stored in an object tree inside a single *store*. In this section, the actions and reducers concepts are analyzed, and a reference is made to the actions and reducers of our application. Figure 7 contains the Redux folder structure.

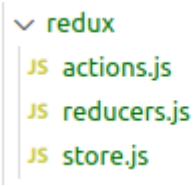

Figure 4. Redux folder structure

Actions

Actions are plain JavaScript objects that have a *type* field. We can think of an action as an event that describes something that happened in the application. Actions are triggered by the react native dispatch() method that is called by the components. They are the only source of information for the store. This means that if any change in the state is necessary, then the change will only be dispatched through the actions. Each action corresponds to a state. Our Redux actions are encapsulated in methods, the action creators. It is through these methods that actions get dispatched inside a component. Our actions, grouped by entities, are shown in table 24. Most of their names are self-explanatory.

Entity	EntityActions
user	setUsername, setEmail, setPassword, setConfirmPassword, setId, setToken, setIsLoggedin
category	setCategoryId, setCategoryType, setCategoryName, setSpendingList, setIncomeList, setCategoryIcon, setSpendingSum, setIncomeSum, setCategories2Budgets (categories2Budgets state includes tuples of budgets with their corresponding categories)
wallet	setWalletId, setWalletBalance, setWalletName, setWalletsList, setWalletsForDropdown (walletsForDropdown state includes the user's wallets and their balances, ready to be used in dropdown menus)
transaction	setTransactionId, setTransactionAmount, setTransactionDate, setTransactionNote, setInterval, setTransactionsList, setTransactionsListForBudget, setTransactionsListForChart, setRecurringTransactionsList, setTransactionDateRangeTitle, (transactionDateRangeTitle state can be "This week", the current month, or "This year"), setTransactionCustomDateRangeTitle, (customDateRangeTitle is a date range in the following form: dd/mm/yy - dd/mm/yy), setTransactionsSumsByCategory, setTransactionsByWallet, setTransactionDueToday, setTransactionOvedue
budget	setBudgetId, setBudgetName, setBudgetAmount, setBudgetBalance, setStartDate, setEndDate, setBudgetNote, setBudgetsList
goal	setGoalName, setTargetAmount, setTargetDate, setSavedAmount, setGoalNote, setGoalsList, setGoalIcon, setGoalMonthlySum
chart	setChartDateRangeTitle (the same as setTransactionsDateRangeTitle but for the chart's transactions), setChartCustomDateRangeTitle (the same as setTransactionsCustomDateRangeTitle but for the chart's transactions)

Table 24. Redux actions grouped by entities

Reducers

Reducers are pure functions that take the current state and an action as arguments and return a new state result. In other words, (state, action) => newState. Reducers must always follow some special rules in order to be predictable:

- To avoid random behavior they should only calculate the new state value based on the state and action arguments, and not based on variables outside themselves.
- They are not allowed to modify the existing state. Instead, they must make immutable updates, by copying the existing state and making changes to the copied values.
- They must not do any asynchronous logic or other "side effects".

We have divided our reducers into seven functions, one for each entity: userReducer, categoryReducer, transactionReducer, budgetReducer, goalReducer, chartReducer

Store

The store, with the help of reducers, stores the entire state of the application as a plain JavaScript object. It is important that there is only a single store in a Redux application. For splitting our data handling logic, we use reducer composition and create multiple reducers, as explained in the reducers, that can be combined together instead of creating separate stores.

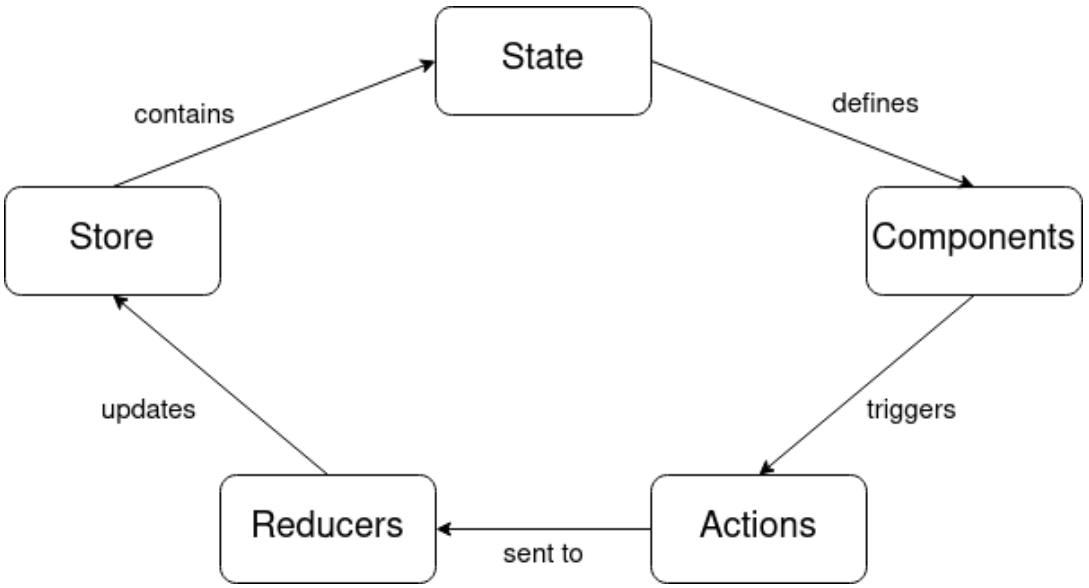

Figure 5. Redux workflow

5.3.2.5. Services

The services folder contains the methods that are responsible for communicating with the server and handling API requests. They also dispatch redux actions to update the store's states with the data received from the server. The services methods are called by multiple screens and components when the application needs to make a request to the server. Because of that, services are the link between the application and the server. The client.js file includes the Axios instance through which all the methods send their requests. Services are grouped by feature as shown in Figure 7.

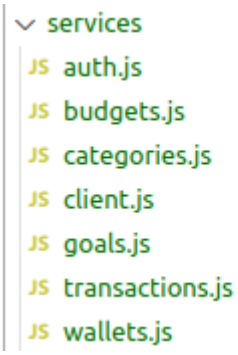

Figure 6. Services folder structure

Method Parameters

The services functions have several parameters. Most of them are states from the Redux store, some are component states and others are simple variables. The parameters' names that are not self-explanatory are explained below.

header: contains the user's authentication token.

dispatch: is the dispatch function from the redux store. We get access to dispatch from the useDispatch hook. These methods cannot have their own dispatch functions since they are not components and only components can use hooks. That is why they get the dispatch from their arguments.

action: is a string that is either "Income" or "Expense".

request: is a string that is either "post" or "delete".

Auth

Table 25 shows the auth services methods with their parameters and their descriptions.

Method name	Parameters	Description

APIRequestSignup	username, email, password, confirmPassword, dispatch	makes a post request to "/user", with the props attributes, and if the user creation is successful, calls the *APIRequestSignin*.
APIRequestSignin	email, password, dispatch	makes a post request to "/signin", and if it is successful, dispatches the actions *setUsername*, *setToken*, and *setId* with the data received from the server.

Table 25. Auth services methods

Categories

Table 26 shows the category services methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**
APIRequestCreateCategory	newCategory, header, dispatch, navigation	makes a post request to "/category", and after that, calls the *APIRequestGetIncomeCategories* and *APIRequestGetSpendingCategories* methods.
APIRequestGetIncomeCategories	header, dispatch	makes a get request to "/category/income", and with the data received by the server, dispatches the *setIncomeList* action.
APIRequestGetSpendingCategories	header, dispatch	makes a get request to "/category/spenidng", and with the data received by the server, dispatches the *setSpendingList* action.
APIRequestUpdateCategory	categoryId, categoryName, icon	makes a put request to "/category/:category_id", with *categoryName* and *icon* in the request body.

APIRequestDeleteCategory	categoryId, navigation	makes a get request to the "/category/:category_id", to check if the category belongs to a budget, and a get request to "/transaction-category/:category_id" to check if there are transactions with this category. If there is no budget or transaction with this category, the method makes a delete request to "/category/:category_id". If there is at least one budget or one transaction, it makes a put request to the "/category/hide/:category_id".
APIRequestDeleteCategoriesFromBudget	categoriesNames, header, dispatch	for each of the categoryNames, makes a get request to "/category/from/:category_name", to get the category's id, and then calls the *APIRequestUpadetCategoriesWithBudget* method of the services budget file.

Table 26. Category services methods

Wallets

Table 27 shows the wallet services methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**
APIRequestCreateWallet	newWallet, header, dispatch, navigation	makes a post request to "/wallet", and after that, calls the *APIRequestGetWallets* method, and then navigates the user back to the WalletsScreen.
APIRequestGetWallets	header, dispatch	makes a get request to "/wallet" in order to dispatch the *setWalletsList*

		action.
APIRequestEditWallet	newWallet, walletId	makes a put request to "/wallet/:wallet_id", with the *newWallet* in request body.
APIRequestGetWalletById	walletId	makes a get request to "/wallet/:wallet_id" in order to dispatch the *setWalletName* action.
APIRequestDeleteWallet	walletId	makes a delete request to "/wallet/:wallet_id", and calls the *APIRequestGetWallets* to update the *walletsList* state.

Table 27. Wallet services methods

Transactions

Table 28 shows the transaction services methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**
APIRequestCreateTransaction	newTransaction, action, header, dispatch,	makes a post request to "/transaction", and if it is successful, makes a post request to "/transaction-by-date-range", with the *transactionDate* in the request body, from which gets the latest transaction. With this calls the *updateWallet* and *updateBudget* methods, which change the balance. Finally, it calls the *APIRequestGetTransactionSums*.
updateWallet	walletId, transactionId, transactionAmount,	gets the old wallet balance by making a get request to "/wallet/:wallet_id". Then, depending on whether the request is

	request, action, header, dispatch	*post* or *delete*, and whether the action is *Income* or *Expense*, the method finds the new wallet balance by calling a corresponding method. Finally, it calls the *updateBalance* method.
updateBalance	id, route, balance, reduxSetter header, dispatch	makes a patch request to the route, dispatches the reduxSetter action to set the state of the new balance, and calls the *APIRequestGetBudgets* method. If it is a *post* request and an *Expense* action, and the new balance is negative, it displays to the user that the transaction was unable to complete due to lack of funds, and makes a delete request for the latest transaction.
updateBudget	categoryId, transactionAmount, transactionDate, request, action, header, dispatch	makes a get request to the "/category/:category_id". From that response gets the budget_id cell record. If the budget cell is not null, means there is a budget for this category, that needs to update its balance. Finally calls the *UpdateBalance* method with the parameters received, the *budgetId*, the "/budget/" route, and the *setBudgetBalance* action.
APIRequestGetTransactionsByDateRange	fromDate, toDate, header, dispatch, transactionOrChart	makes a post request to the "/transaction-by-date-range/", and from the respond, dispatches the *setTransactionsList* action if transactionOrChart is *transaction* or the *setTransactionsListForChart* if transactionOrChart is *chart*, of both if transactionOrChart is *both.*

APIRequestFetchDataAfterNewTransaction	header, dispatch	calls the *APIRequestGetTransactionsByDateRange* with transactionOrChart = "both", the *APIRequestGetBudgets* and the *APIRequestGetRecurringTransactions.*
APIRequestGetRecurringTransactions	header, dispatch	makes a get request to the "/transaction-recurring", and dispatches the *SetReccuringTransactionsList* action.
APIRequestGetTransactionsSums	fromDate, toDate, header, dispatch	makes a post request to the "/transaction-sum", with the *fromDate* and *toDate* to the request body, and dispatches the *SetTransactionsSumsByCategory* action.
APIRequestGetTransactionsByWallet	walletId, header, dispatch	makes a get request to the "/transaction-wallet", and dispatches the *SetTransactionsByWallet* action.
APIRequestUpdateUpcomingTransaction	transactionId	makes a patch request to the "/transaction/upcoming/:transaction_id", with the *upcoming: false* in the request body.
APIRequestUpdateNextOccurrence	transactionId, nextOccurrence, header,dispatch	makes a patch request to the "/transaction/next-occurrecne/:transaction_id", with the *nextOccurence* in the request body.
APIRequestEditTransaction	transactionId, newTransaction, header, dispatch, action	makes a patch request to "/transaction/:transaction_id", and if that is successful, makes a get request to the "/transaction/:transaction_id". Using that transaction record updates the wallet and the budget like *APIRequestCreateTransaction* does.

APIRequestDeleteTransaction	transactionId, budgetId, walletId, transactionDate, header, dispatch, action, amount	makes a delete request to the "/transaction/:transaction_id", and if it is successful, updates the wallet balance with the *updateBalance* method. If the deleted transaction was part of a budget, it updates the budget balance with the *updateBalance* method.

Table 28. Transaction services methods

Budgets

Table 29 shows the budget services methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**
APIRequestCreateBudget	newBudget, header, selectedCategories, dispatch,	makes a post request to "/budget" with the *newBudget* in the request body. Then, the method dispatches the *setBudgetId* action in order to call the *APIRequestUpdateCategoriesWithBudget* .
APIRequestUpdateCategoriesWithBudget	newBudget, header, selectedCategories, dispatch	makes a patch request to "/category/budget" with the *categoryId* and *budgetId* in the request body for each category of the *selectedCategories*. If that is successful, it makes a get request for the spending categories in order to dispatch the *setSpendingList* action and call the *makeCategories2BudgetsList* method (utils, budgetMethods).
APIRequestGetBudgetById	budgetId, header, dispatch	makes a get request to "/budget/:budget_id" in order to

		dispatch the *setBudgetName* action.
APIRequestGetBudgets	header, dispatch	makes a get request to "/budget" in order to dispatch the *setBudgetsList* action.
APIRequestEditBudget	newBudget, header, setDailyTransactions	makes a put request to "/budget/:budget_id", with the *newBudget* in the request body.
APIRequestGetDailySumAmounts	fromDate, toDate, selectedCategories, header, dispatch	makes a post request to "/transaction-dailysum", with the *selectedCategories*, *fromDate*, and *toDate* in the request body. Then, the method sets the *setTransactionsList state* with the response from the server.
APIRequestGetTransactionsOfThisBudget	fromDate, toDate, selectedCategories, header, dispatch	makes a post request to "/transaction-budget", with the *selectedCategories*, *fromDate*, and *toDate* in the request body. Then, the method dispatches the *setTransactionsListForBudget* action with the response from the server.
APIRequestDeleteBudget	budgetId, navigation, header, dispatch	makes a delete request to "/budget/:budget_id", and calls the *APIRequestGetBudgets* to update the *budgetsList* state.

Table 29. Budget services methods

Goals

Table 30 shows the goal services methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**

APIRequestCreateGaol	newGoal, header, dispatch,	makes a post request to "/goal" with the *newGoal* in the request body.
APIRequestAddSavdeAmount	goalId, amount, savedAmount	makes a post request to "/goal-assignment/:goal_id", with the *amount* in the request body, and makes a patch request to the "/goal/:goal_id", with the updated amount in the request body.
APIRequestGetGoalSavedAmount	goalId, header, dispatch	makes a get request to "/goal/:goal_id", and if it is successful, dispatches the *setSavedAmount* action with the response from the server.
APIRequestGetGoals	header, dispatch	makes a get request to "/goal" in order to dispatch the *setGoalList* action.
APIRequestEditHGoal	newGoal, header, dispatch	makes a put request to "/goal/:goal_id", with the *newGoal* in request body.
APIRequestGetMonthlySum	goalId, monthPeriod, header, dispatch	makes a post request to "/goal-assignment/monthly-sum/:goal_id", and dispatches the *setGoalMonthlySum* action with the response from the server.
APIRequestDeleteGoal	goalId, navigation, header, dispatch	makes a delete request to "/goal/:goal_id", and calls the *APIRequestGetGoals* to update the *goalsList* state.

Table 30. Goal services methods

5.3.2.6. Utils

The utils folder contains the methods that handle the logic of the application: validation, and calculation methods that are frequently used from services, screens, and

components.

Authentication methods

Table 31 shows the authentication util methods with their parameters and their descriptions.

Method name	Parameters	Description
isEmailValid	email	checks if the email is valid.
isFormValidSignup	fieldsList	checks if the fields of the sign-up form are all completed and valid. Returns true, if they are valid, or returns false and displays alerts to the user.
isFormValidSignin	fieldsList	checks if the fields of the sign-in form are all completed and valid. Returns true, if they are valid, or returns false and displays alerts to the user.
areFormFieldsFilledOut	fieldsList	takes as parameters a list of mandatory fields, and alerts the user if null or undefined fields are found.
serverApprovesSignup	serverResponse	checks the sign-up response of the server. If it is successful returns true, and if it is not alerts the user with a corresponding message.
serverApprovesSignin	serverResponse	checks the sign-in response of the server. If it is successful returns true, and if it is not alerts the user with a corresponding message.

Table 31. Authentication util methods

Category methods

Table 32 shows the category util methods with their parameters and their descriptions.

Method name	Parameters	Description
setStatesNull	dispatch	makes the entity's states (*categoryIcon*, *categoryName*, *categoryType*) null.
findCategory	id, incomeList, spendingList	searches for the *id* inside the *incomeList* and *spendingList*, and returns the whole category record.
findCategoryType	id, incomeList, spendingList	searches for the *id* inside the *incomeList* and *spendingList*. Returns "Income" if the *id* is found in the *incomeList*, or "Spending" if the *id* is found in the *spendingList*.
findCategoryIcon	id, incomeList, spendingList	searches for the *id* inside the *incomeList* and *spendingList*, and returns the icon of this category.
findSum	incomeList/ spendingList, transactionsSums ByCategory	calculates and returns the total income or spending amount based on the transactions in the transactionsSumsByCategory state that stors monthly tatal spending and income in each category.
makeCategoriesFor Dropdown	incomeList/ spendingList	calculates and returns tuples of category id- category name.
makeListOfCategori esForBudgetForm	spendingList	calculates and returns tuples of category id- category name only for the categories that do not already belong to a budget.

Table 32. Category util methods

Wallet methods

Table 33 shows the wallet util methods with their parameters and their descriptions.

Method name	Parameters	Description
setStatesNull	dispatch	makes the entity's states (*walletName*, *walletBalance*) null.
findWallet	id, walletsList	searches for the *id* inside the *walletsList* and returns the whole wallet record.
makeWalletsForDropdown	walletsList	calculates and returns tuples of wallet id-wallet name and balance.

Table 33. Wallet util methods

Transaction methods

Table 34 shows the transaction util methods with their parameters and their descriptions.

Method name	Parameters	Description
setStatesNull	dispatch	makes the entity's states (*transactionAmount*, *transactionDate*, *transactionNote*) null.
didDateChange	date, itemDate	returns true if *date* equals *itemDate.*
isRecurringTransaction	transaction	returns true if the *interval_period* of the *transaction* is not null.
didIntervalChange	interval, itemInterval	returns true if *interval* equals itemInterval.
findIntervalSTR	interval	from the *interval* abbreviation, returns a sentence (ex. From *interval* = "1W" returns "Every 1 week").
findNextOccurrence	Interval, transactionDate	from the *transactionDate* and the *interval* calculates and returns the *nextOccurence.*
makeTitleForRecurr	nextOccurence	from the *nextOccurence* and the current

ingTransactions		date calculates and returns when the next transaction is due.
makeSmallTitleForTransaction	date	from the given *date*, and the current date, calculates and returns a small string that displays a date.
makeSmallTitleForRecurringTransaction	nextOccurence	from the *nextOccurence*, and the current date, calculates and returns a small string that displays a date.
isCustomPeriod	from, to	Checks if a date range is a week, a month, a year, and if it is it returns false.
recurringTransactionsToday	list, dispatch	for each transaction in the *list*, checks the *nextOccurence* field, and if it finds at least one transaction which *nextOccurence* is today, dispatches the *setTransactionDueToday* action with true.
isPostExpenseOrDeleteIncome	request, action	returns true if the request is *post* and the action is *Expense* or if the request is *delete* and the action is *Income.*
isPostIncomeOrDeleteExpense	request, action	returns true if the request is *post* and the action is *Income* or if the request is *delete* and the action is *Expense.*
handlePostExpenseAndDelteIncome	oldBalance, transactionAmount	returns the subtraction of the *transactionAmount* from the *oldBalance.*
handlePostIncomeAndDeleteExpense	oldBalance, transactionAmount	returns the addition of the *transactionAmount* and the *oldBalance.*

Table 34. Transaction util methods

Budget methods

Table 35 shows the budget util methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**
setStatesNull	dispatch	makes the entity's states (*budgetBalance*, *budgetName*, *budgetNote, endDate, budgetAmount*) null.
findBudgetId	id, incomeList, spendingList	searches for the *id* inside the *spendingList* and returns the category's budget_id.
makeCategories2BudgetsList	spendingList, dispatch	loops through the spendingList, and makes tuples of categoryName and budgetId.
setStartDateIfNull	startDate, dispatch	if *startDate* is null, it dispatches the *setStartDate* action with the current date.
makeListOfBudgetCategories	budget, categories2Budgets	for each budget-category tuple in the *categories2Budgets* list, saves and returns only the category names of the *budget*.
findPerioUntil	date	finds if the date is this week, next week, this month, next month, or this year, and returns that in a string.
getDaysOfBudget	startDate, endDate	from the startDate and the endDate of the budget, calculates and returns the total days of the budget.
findCategoryIdsOfBudget	categoriesNames, spendingList	searches for the categoryNames in the spendingList and returns a list of their ids.
findXAxisData	daysOfBudget	splits the *daysOfBudget* into 3 equal date ranges and returns an object with 4 lists. Those lists iclude the 4 dates, and their position into the *daysOfBudget* date period.

deconstuctAndKeepDates	XAxisData	returns the dates of the *XAxisData* object.
deconstuctAndKeepPositions	XAxisData	returns the positions of the *XAxisData* object.
positionOfToday	daysOfBudget	for the daysOfBudget date range, finds and returnd the position of today's date in that range.
makeDailySumsForChart	daysOfBudget, dailyTransactions	calculates and returns a list with the total daily amount spent for each day of the *daysOfBudget*.
isExpiredBudget	endDate	returns true if the *endDate* of the budget has passed.
onPressQuestionmark		displays an alert to the user that lets him/her know that he/she can choose one or more categories for a budget.

Table 35. Budget util methods

Goal methods

Table 36 shows the goal util methods with their parameters and their descriptions.

Method name	**Parameters**	**Description**
setStatesNull	dispatch	makes the entity's states (*goalIcon*, *goalName, goalNote, targetDate, targetAmount*) null.
findThirdBoxTitle	targetAmount, targetDate	checks whether targetAmount and targetDate are null, or have a value. For each of the four different combinations of those, returns a different string.
findThirdBoxValue	targetAmount,	checks whether targetAmount, and

	targetDate, savedAmount, monthlySum	targetDate are null or have a value. Each of the four different combinations of those calculates and returns a different value.

Table 36. Goal util methods

6. User Interface

In this chapter, the User Interface of the application is presented.

6.1. Auth Screens

In the **Sign-up Screen** (Figure7) a new user can create an application account through the form by completing all the form fields. In the **Sign-in Screen** (Figure7), a user can log in to his/her account with his/her e-mail and password.

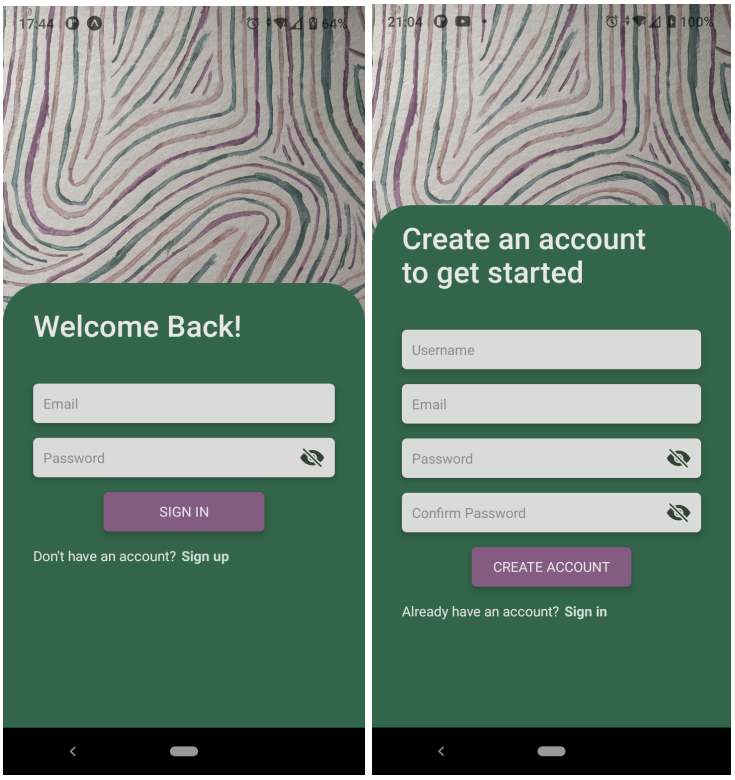

Figure 7. Sign-up and sign-in screens

6.2. Home Screen

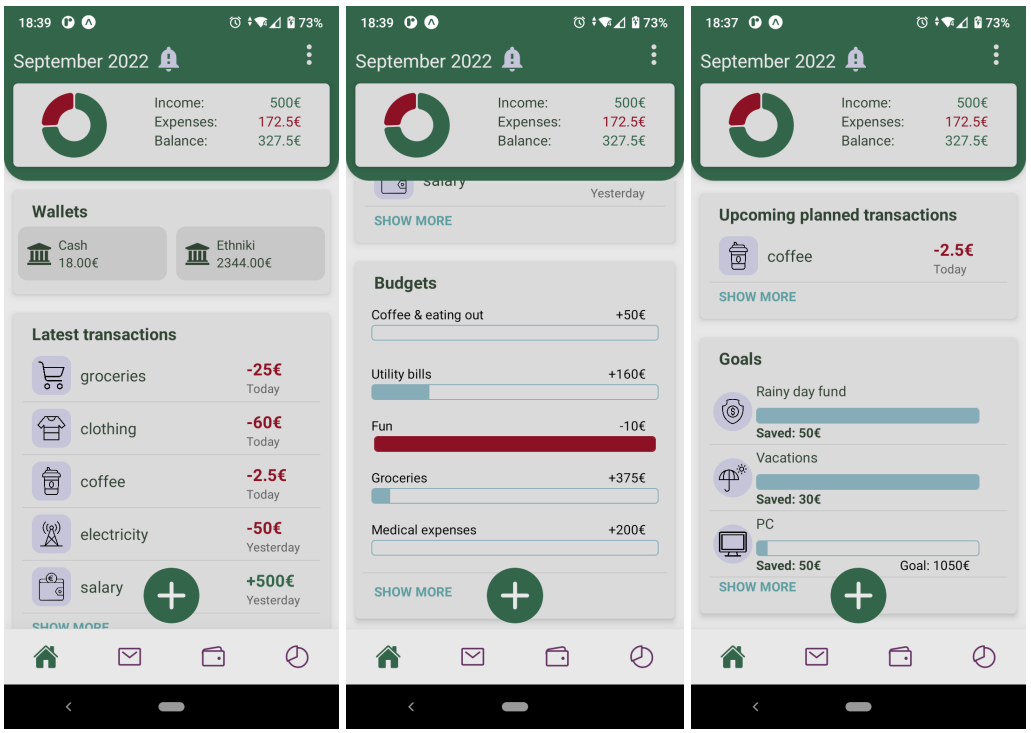

Figure 8. Home Screen

The **Home Screen** (Figure 8) is the landing screen of the application, once the user is logged in. The Home Screen has a header that features some monthly data: the income, the expenses, the balance, and a pie chart of the balance and expenses. From the three dots, the user can navigate to the Settings Screen. When the user clicks on the grey notification icon, a pop-up informs him/her that he/she has planned transactions for today. Through the pop-up, the user can navigate to the screen that shows the upcoming planned transactions. The main part of the page consists of cards: the Wallets card, with the user's wallets and their balance, the Latest transactions card, featuring the user's latest transactions, the Budgets card, where he/she gets a summary of his/her budget's progress, the Upcoming planned transactions card, that shows to the user his/her planned transactions and last, but not least, is the Goals card, with a summary of the user's Goals. If the user clicks on any item of these cards (wallet, budget, transaction, goal), he/she is navigated to the respective screen. The green floating "+" button is for adding a new income or expense transaction.

6.3. Settings Screen

The **Settings Screen** (Figure 9) is accessible from the three dots icon at the HomeScreen header. Through this screen, the user can configure his/her income and spending categories, see his/her reached goals or log out from the application.

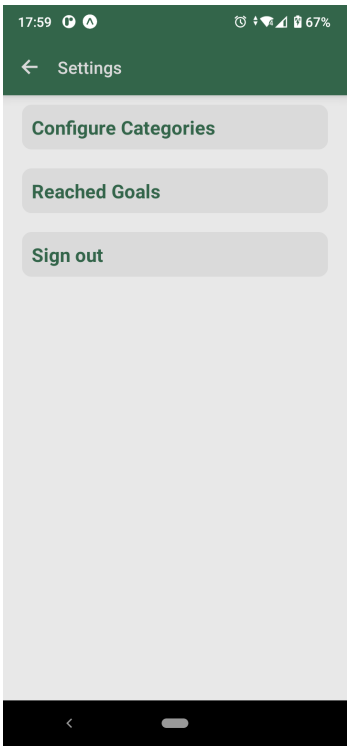

Figure 9. Settings Screen

6.4. Category Screens

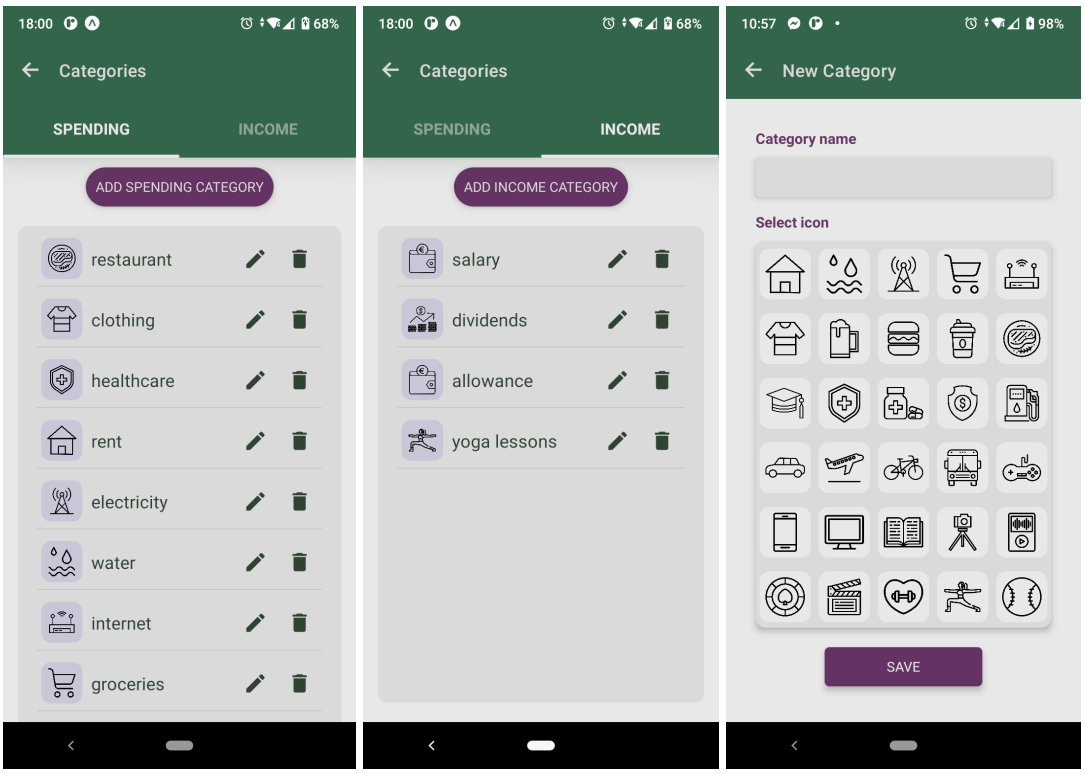

Figure 10. Category Screens

The **Category Screen** (Figure 10) has two parts: one for the spending categories and one for the income categories. The user can add a new category by pressing the respective button, and edit or delete an existing category. The **Add-Edit Category Screen** (Figure 10) has a form from which the user can add a new category or a form from which the user can change the category's name and icon.

6.5. Wallet Screens

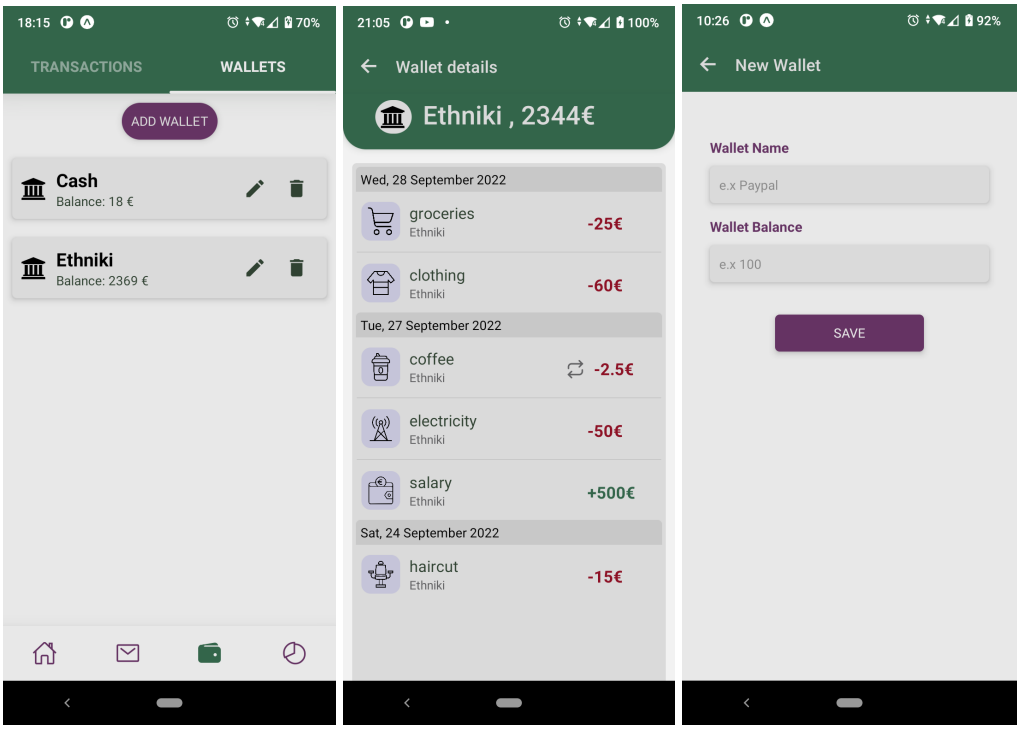

Figure 11. Wallets Screen, Wallet Details Screen, Add-Edit Wallet Screen

To navigate to the **Wallets Screen** (Figure 11) the user needs to press the wallet icon at the bottom navigator, swipe right, or click on the WALLETS tab at the top right of the screen. From the Wallets Screen, the user can see his/her wallets and their balance. The user can add a new wallet by pressing the “add wallet” button. If the user has stopped using the application for a while he/she can change the balance of his/her wallet, by clicking on the edit icon and delete a wallet that no longer serves him/her by clicking on the delete icon. The user can also press on the wallet to see the monthly transactions he/she did through that wallet.

6.6. Transaction Screens

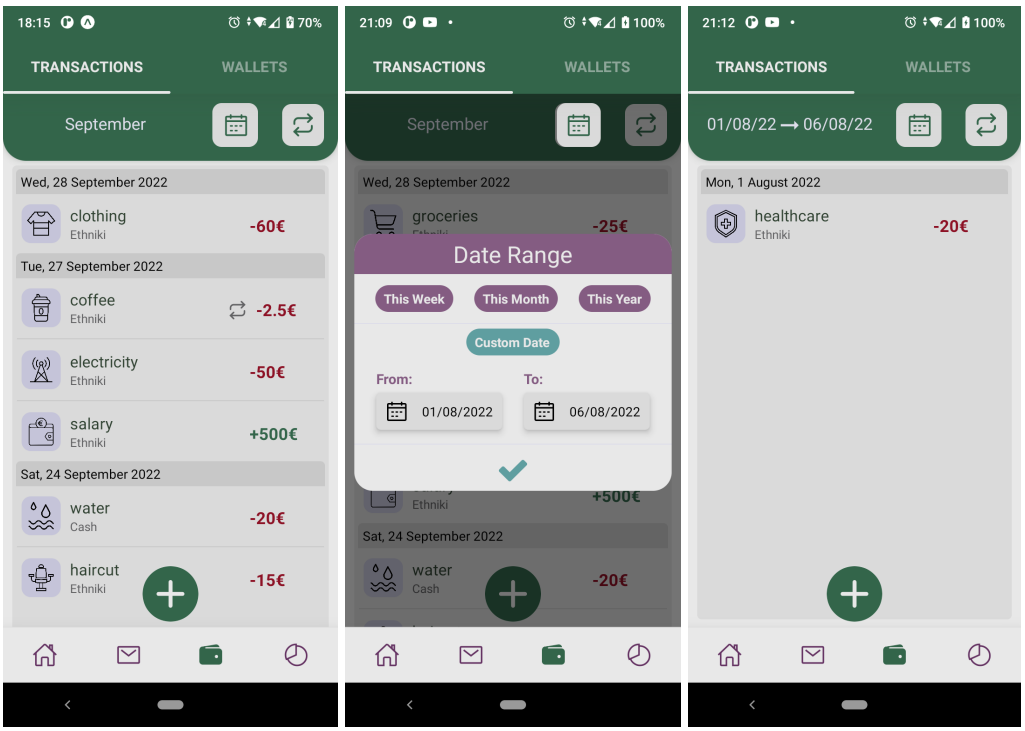

Figure 12. Transactions Screen, change date range pop-up

In the **Transactions Screen** (Figure 12) the user can see his/her transactions. The user can navigate there by clicking on the wallet icon on the bottom tab navigator. By default, the transactions featured are the transactions of the current month, as the user can see on the header. By clicking on the calendar icon, a pop-up appears and allows the user to select from the existing date ranges or choose a custom one. After clicking on the "tick" icon the user can see at the header the date range he/she chose and the transactions of that date range on the screen.

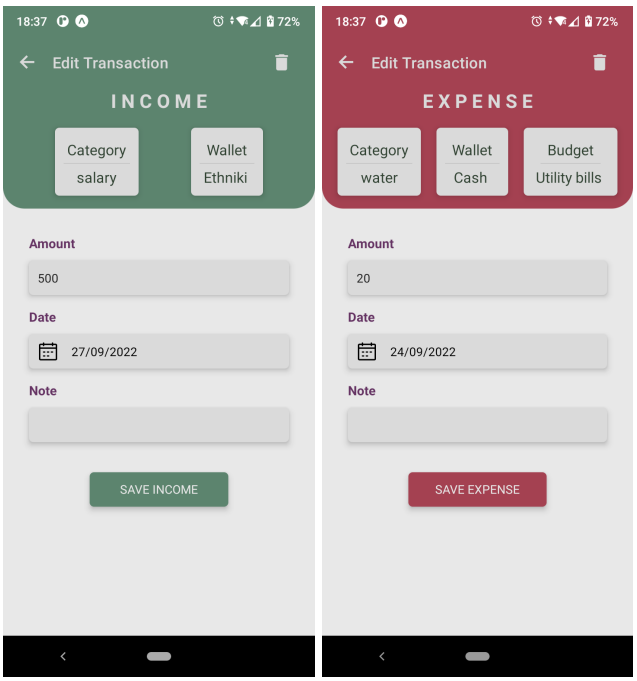

Figure 13. Edit Transaction Screen

If the user clicks on a transaction he/she is navigated to the **Edit Transaction Screen** (Figure 13), where he/she can see some details about the transaction, like the wallet, category, and budget (if it exists) on the header, along with a delete icon, to delete this transaction. Under the header, there is a form with the amount and the date, which the user can change and press the “save” button.

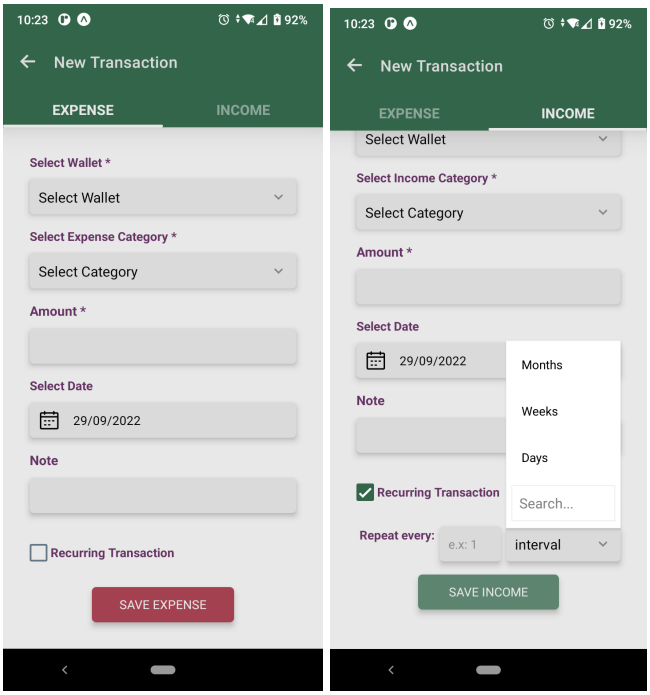

Figure 14. Add Transaction Screen

From the Transactions Screen the user can add a new transaction by clicking on the "+" button. On the **Add Transaction Screen** (Figure 14), there are two tabs, one for adding an expense on the left, and one for adding an income to the right. The user can add a new expense or income by selecting a wallet, a spending or income category, and inputting an amount. The date is set by default to the current day and the user can also change the date and make a past transaction (not a future one). If the user wants to make this transaction recurring, he/she can press the Recurring transaction checkbox. When the checkbox is pressed a new form appears, through which the user can select the interval of the transaction.

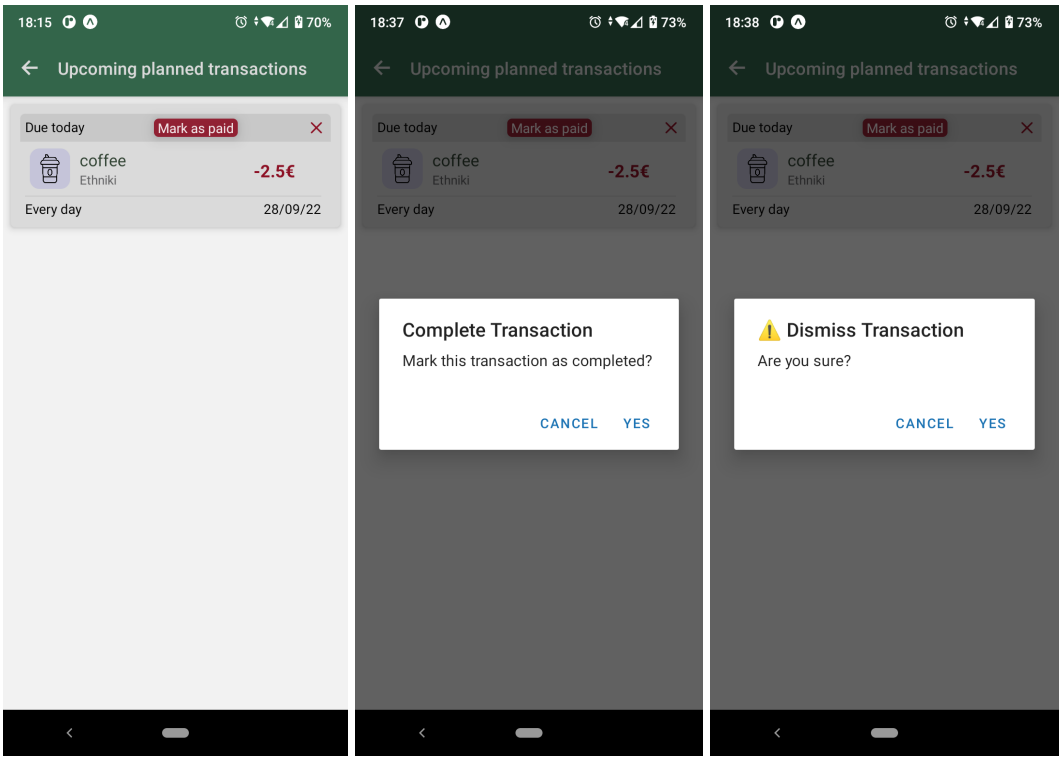

Figure 15. Recurring Transactions Screen

The user can see his/her recurring transactions on the **Recurring Transactions Screen** (Figure 15) by clicking on the repeat icon in the Transactions Screen header. If the user wants to "skip" a transaction he/she can click on the red "x" icon to dismiss the next transaction, and then the due date will be updated. If the transaction is due today or is overdue the user can click on the "mark as paid" button, to add the transaction, without having to go through the Add Transaction Screen.

6.7. Budget Screens

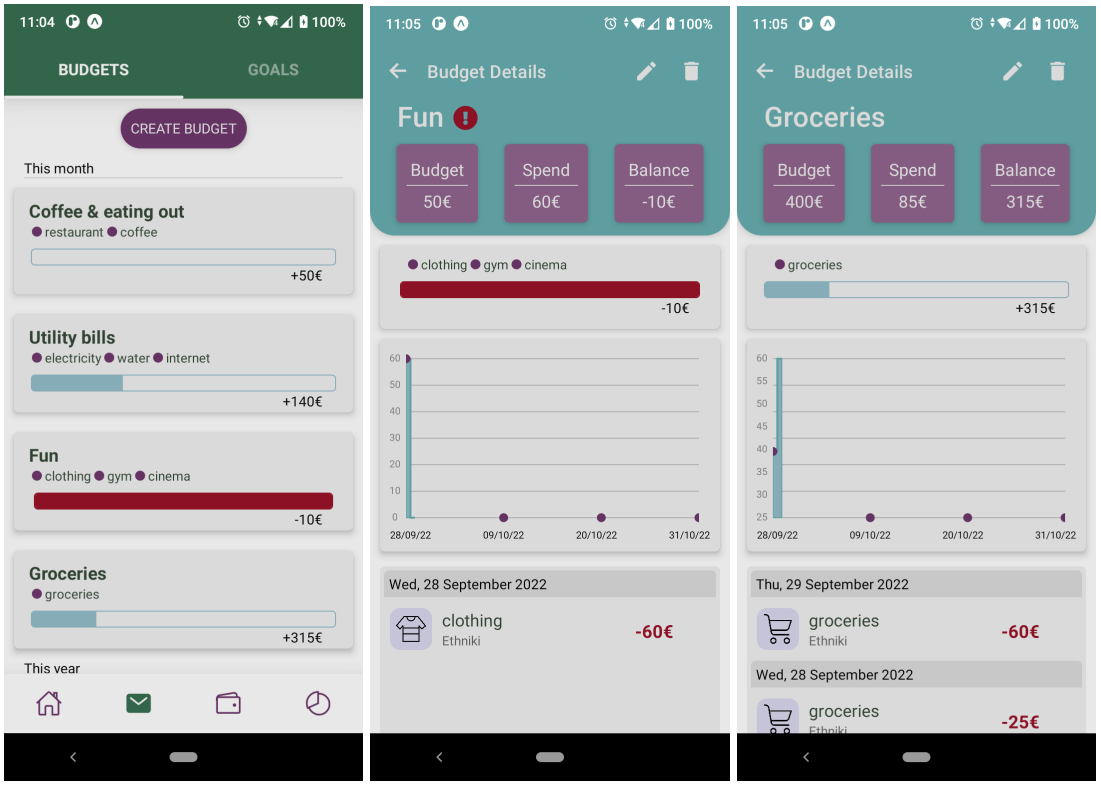

Figure 16. Budgets Screen, Budget Details Screen

The user can navigate to the **Budgets Screen** (Figure 16) by clicking on the envelope on the bottom navigator. On this screen, the user can see his budgets categorized by their end date into this week, this month, etc. The user can see the budget name, the categories he assigned in this budget, and the balance of the budget. The balance when positive, indicates the amount that the user has left to spend, if negative indicates the amount that the user spent more than what his/her plan was. If the user has overspent on a budget the bar becomes red to indicate that. The user can click on a budget to see more details about it or create a new budget by clicking the "create budget" button at the top of the screen.

By clicking on a budget the user is navigated to the **Budget Details Screen** (Figure 16). In the header of this screen, the user can see the budget's initial amount, the amount spent, and the balance. After that, the user sees a line chart that shows the amount of money he/she has spent every day of the budget. Finally, the user sees the transactions of the budget's categories.

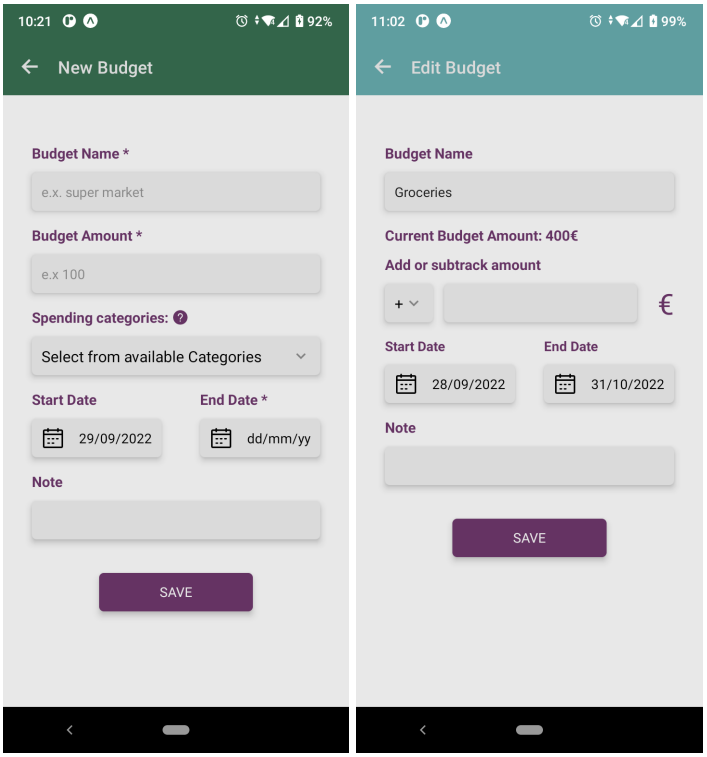

Figure 17. Add-Edit Budget Screen

When the user clicks on the "create budget" button he/she is redirected to the **Add-Edit Budget Screen** (Figure 17)**.** The user inputs a name and a budget amount and selects one or more spending categories from the drop-down menu. The start date is by default to the current day and can be changed to a future date, not a past one. The user must click on the end date calendar and select the end date through a date picker.

6.8. Goal Screens

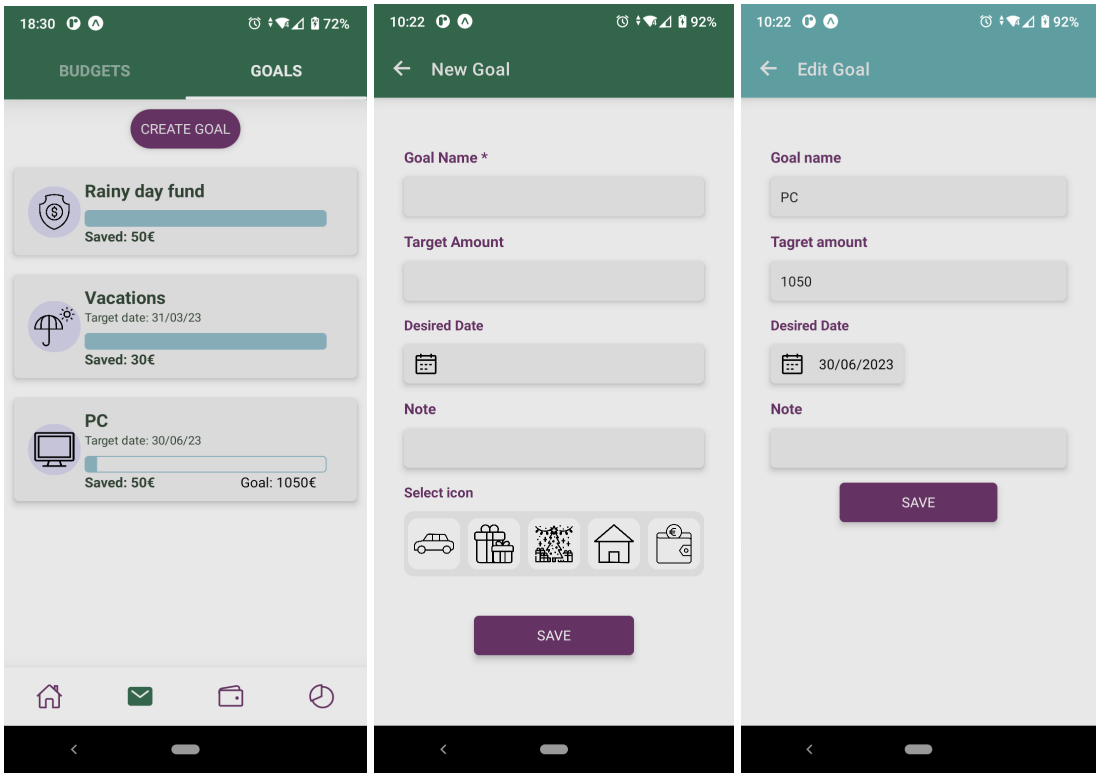

Figure 18. Goals Screen, Add-Edit Goal Screen

The user can navigate to the **Goals Screen** (Figure 18) by clicking on the envelope on the bottom navigator, swiping left, or clicking on the Goals tab of the screen. On this screen, the user can see his goals. The user can see the goal icon and name, the target date (if there is one), the target amount (if there is one), and the saved amount. The bar indicates the percentage of the target amount that has been saved. If there isn't a target amount, the bar is always full. The user can click on a goal to see more details about it or create a new goal by clicking the "create goal" button at the top of the screen. When the user clicks on the "create goal" button he/she is redirected to the **Add-Edit Goal Screen.** The user inputs a name and selects an icon. If he wants he can input a target amount, and a target date by selecting a date from the date picker.

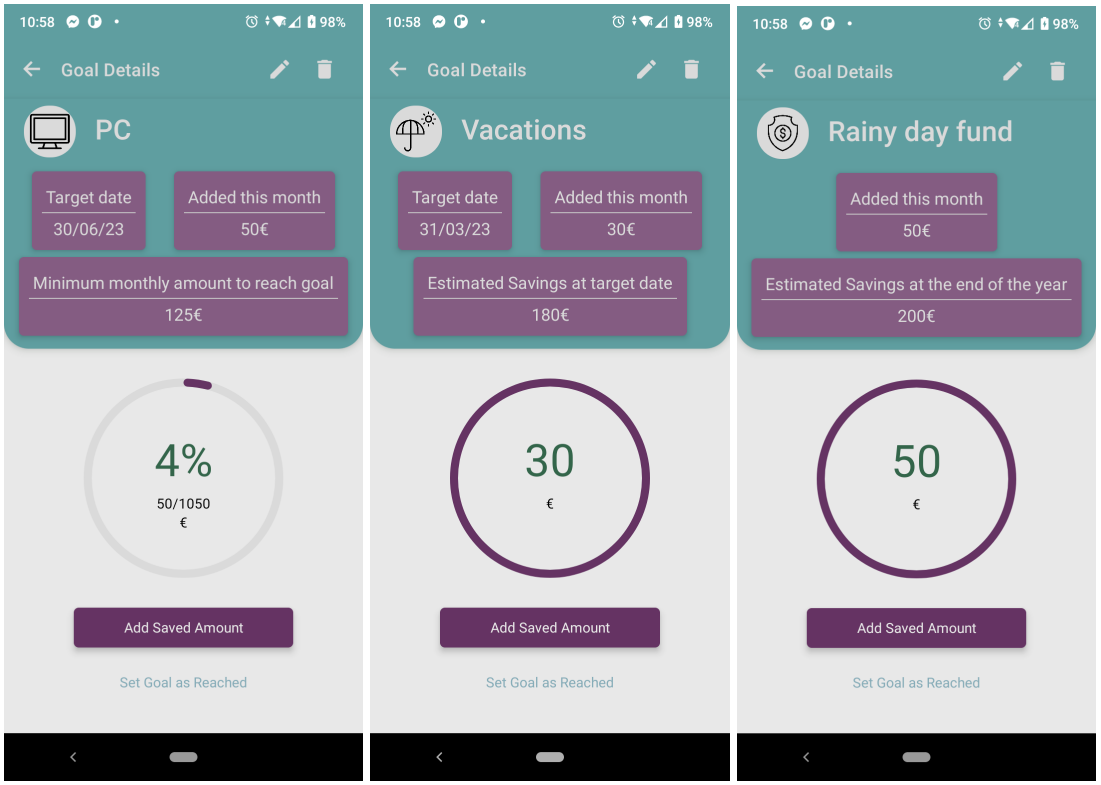

Figure 19. Goal Details Screen

When the user clicks on a goal, he/she is navigated to the **Goal Details Screen** (Figure 19). This screen is a little different for the different kinds of goals, but always has a header that informs the user about the target date (if there is one), the amount saved this month, and gives an insight into the goal's progress. In the main part of the screen, the user sees a progress circle that indicates the goal's progress. By clicking on the "add saved amount" button the user can add an amount to the goal. By clicking on the "set goal as reached button" the user can archive an achieved goal. Last but not least the user can edit the goal by clicking on the edit icon at the top right of the screen and be navigated to the Add-Edit Goal Screen for edit, or delete the goal by clicking on the delete icon.

6.9. Charts Screen

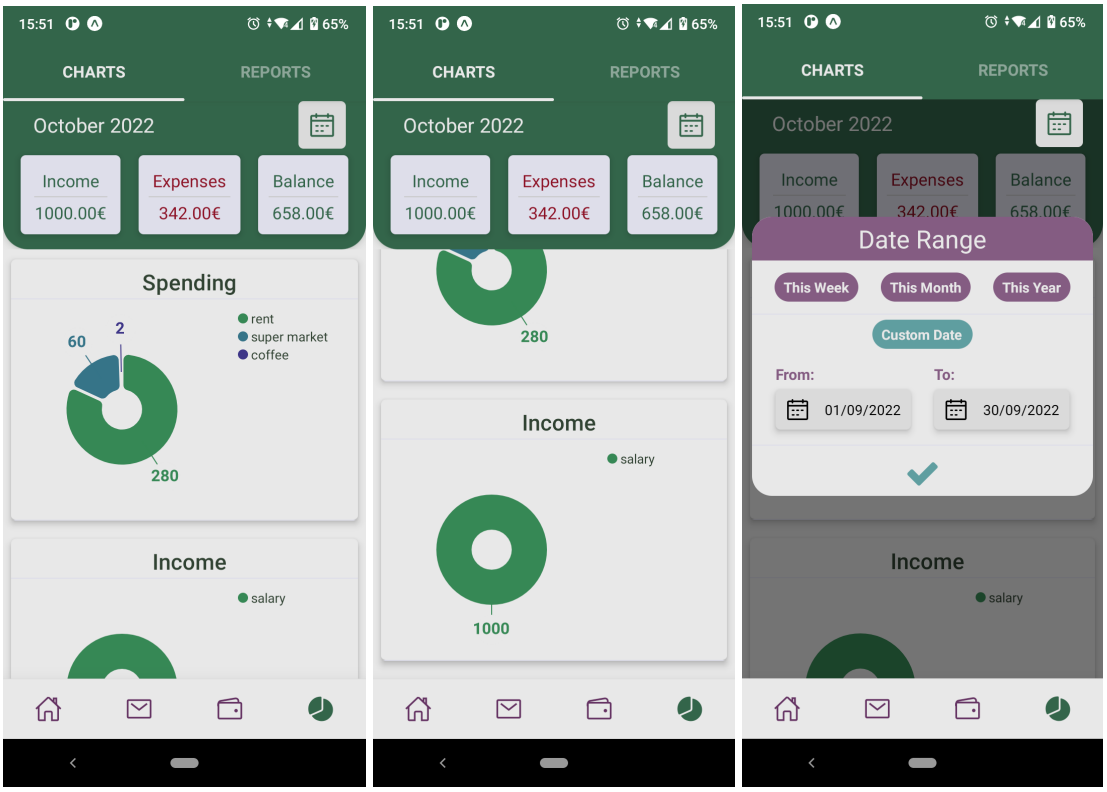

Figure 20. Charts Screen

The user can navigate to the **Charts Screen** (Figure 20) by clicking on the graph icon at the bottom tab navigator. The user can see his spending and income by categories on this screen in pie charts. By default, the user sees data about the current month, but he/she can choose the date range from the calendar pop-up just like in the transactions.

6.10. Reports Screens

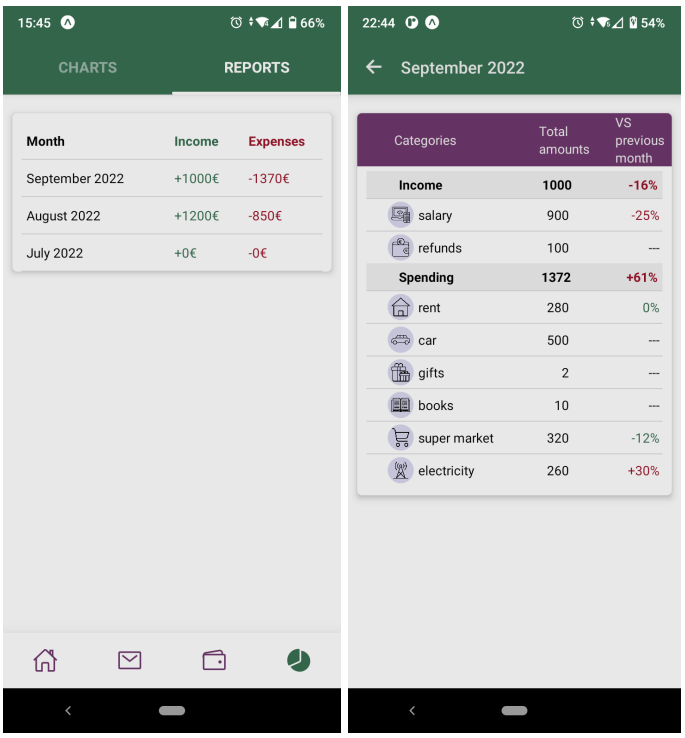

Figure 21. Reports Screen, Monthly Report Screen

The user can navigate to the **Reports Screen** (Figure 21) by clicking on the pie chart icon on the bottom navigator, swiping left, or clicking on the Reports tab of the screen. The user can see a list of the income and spending total amounts for each of the previous months. When the user clicks on a month he/she is navigated to the **Monthly Report Screen** (Figure 21) for the selected month, where he/she views the amounts spent or earned in each category and a comparison with the previous month's amounts.

7. Testing

In this chapter we explain how we tested our application. Below we present the technologies used and the exact way we used them to achieve our goal.

7.1. Technologies

For the testing we are using **Jest** and **Supertest**. Jest is a JavaScript framework for creating, running, and structuring tests, with a focus on simplicity. With the jest framework, we can write unit test functions that test the smallest testable parts of our application. Supertest is a library for testing HTTP servers. It enables us to programmatically send HTTP requests such as GET, POST, PATCH, PUT, and DELETE to HTTP servers and get results.

7.2. Testing Process

Using the technologies mentioned above, we are able to test the endpoints of our server, meaning the routes of our server. There is one test file for every entity on the server. Each file tests the entity's routes. In order to keep the testing separate from the application, we are using a copy of our database. Before running a test file, we must change the database from db.js, by commenting the application's database and uncommenting the test's database.

```
describe("POST/create-user", () => {
  test("should respond with json created successfully", async () => {
    const response = await request(app).post("/user").send({
      email: "doe@gmail.com",
      username: "jane",
      password: "verysecretpassword",
      confirmPassword: "verysecretpassword",
    });
    expect(response.body.message.data.affectedRows).toBe(1);
    expect(response.body.message.message).toStrictEqual(
      "user created successfully"
    );
  });
})
```

Figure 22. Jest & Supertest test example

For writing tests with Jest, we wrap the actual tests in describe() functions. The describe

blocks can contain more than one related tests. The actual test is in the text() function. In the test, we make an API request to our server with certain inputs. The expect() function is used every time we want to test a received from the server value. The expect function is used alongside “matchers” (toBe, toEqual, toContain) that let us validate different things. Figure 22 shows a simple example of how the describe, text, and expect functions work.

7.2.1. User tests

In this test file we test the user routes. In Figure 23, we see the successful execution of these tests.

```
PASS routes/__test__/unit/user.test.js
  POST/create-user
    ✓ if given a valid username, password and confirm password, should respond with json created successfully (210 ms)
    ✓ if given a valid username, password and confirm password, should respond with a 200 status code (99 ms)
    ✓ if data is missing, should respond with a status code of 400 (62 ms)
    ✓ if given invalid email, should respond with json invalid email (11 ms)
    ✓ if given invalid password, should respond with json invalid password (17 ms)
    ✓ if given passwords that do not match, should respond with json passwords dont match (10 ms)
  POST/signin
    ✓ if given a valid email and matching password, response should have email (33 ms)
    ✓ if given a valid email and matching password, should respond with a 200 status code (38 ms)
    ✓ if the username and/or password is missing, should respond with json email/password is required and with a status code of 400
    ✓ if email/password dont match, should respond with json email/password dont match  (35 ms)
  POST/sign-out
    ✓ should respond with json successfully signed out and with a status code of 200 (19 ms)

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   0 total
Time:        1.657 s, estimated 2 s
```

Figure 23. User tests

7.2.2. Category tests

In this test file we test the category routes. In Figure 24, we see the successful execution of these tests.

```
PASS routes/__test__/unit/category.test.js
  POST/ create category
    ✓ if given a name and type, should create a new record and respond with a status code of 200 (203 ms)
    ✓ if not given a name and/or type, should respond with a status code of 400 (26 ms)
  GET/ category by id
    ✓ if given a category id, should return the category's record (16 ms)
    ✓ if given a category id that isn't in the database, should return empty array (11 ms)
  GET/ categories
    ✓ if given an auth token, should return the users spending categories (9 ms)
    ✓ if given an auth token, should return the users income categories (8 ms)
  GET/ category from name
    ✓ if given a category name and a user id, should find category's id and respond with a status code of 200 (9 ms)
  PUT/ category
    ✓ if given a new category name and icon, should respond with a status code of 200 (23 ms)
    ✓ if given a new category name and icon, should update category name and icon (28 ms)
    ✓ if mandatory fields(name) are missing, should respond with json category name is a mandatory field (5 ms)
  PATCH/ update category with budget id
    ✓ if given a category id and a budget id, budget id should be saved for this category (27 ms)
    ✓ if given a category id and a budget id, should respond with a status code of 200 (9 ms)
    ✓ if category id is missing, should respond with json category id cannot be null (5 ms)
  PATCH/ hide category
    ✓ if given a category id, should update hide column with true (19 ms)
  DELETE/ category
    ✓ should delete the category, and respond with a status code of 200 (13 ms)

Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        1.452 s, estimated 2 s
```

Figure 24. Category tests

7.2.3. Wallet tests

In this test file we test the wallet routes. In Figure 25, we see the successful execution of these tests.

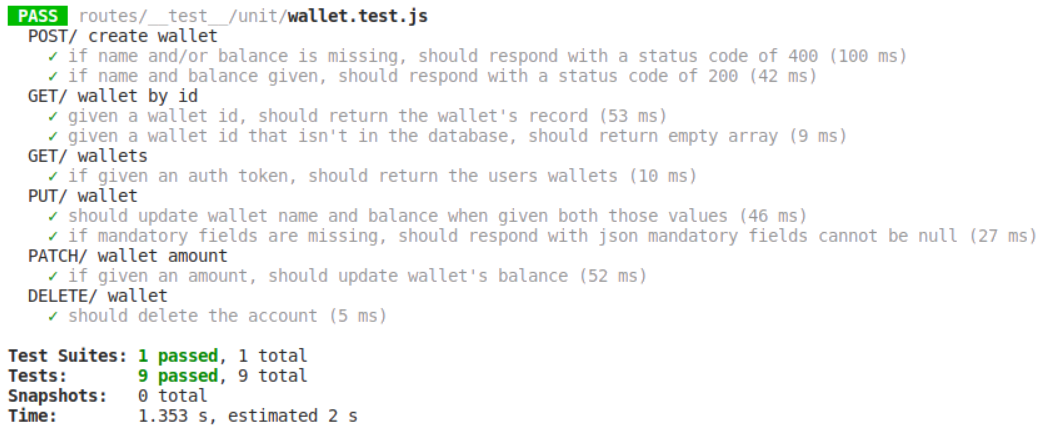

Figure 25. Wallet tests

7.2.4. Transaction tests

In this test file we test the transaction routes. In Figure 26, we see the successful execution of these tests.

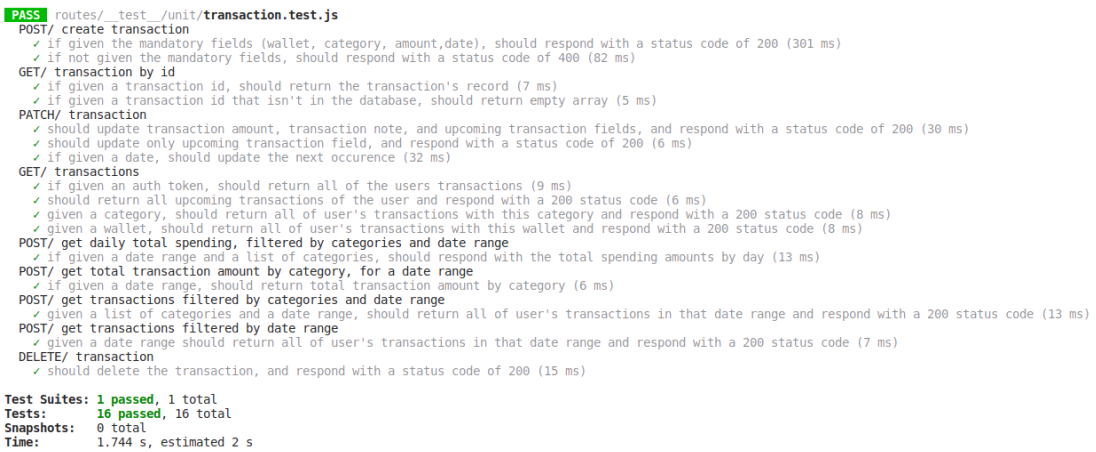

Figure 26. Transaction tests

7.2.5. Budget tests

In this test file we test the budget routes. In Figure 27, we see the successful execution of these tests.

```
PASS routes/__test__/unit/budget.test.js
  POST/ create budget
    ✓ if mandatory fields are missing, should respond with a status code of 400 (118 ms)
    ✓ if mandatory fields given, should respond with a status code of 200 (37 ms)
    ✓ if mandatory fields given, should create a new record (24 ms)
  GET/ budget by id
    ✓ if given a budget id, should return the budget's record (33 ms)
    ✓ if given a budget id that isn't in the database, should return empty array (11 ms)
  GET/ budgets
    ✓ given an auth token, should return the users budgets (16 ms)
  PUT/ budget
    ✓ given all mandatory fields, should update budget (34 ms)
    ✓ if mandatory fields are missing, should respond with a status code of 400 (42 ms)
  PATCH/ budget balance
    ✓ if given a balance, should update budget's balance (33 ms)
    ✓ if not given a balance, should respond with json balance cannot be null (4 ms)
  DELETE/ budget
    ✓ should delete the budget (8 ms)

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   0 total
Time:        1.456 s, estimated 2 s
```

Figure 27. Budget tests

7.2.6. Goal tests

In this test file we test the goal routes. In Figure 28, we see the successful execution of these tests.

```
PASS routes/__test__/unit/goal.test.js
  POST/ create goal
    ✓ if mandatory fields are missing(name), should respond with a status code of 400 (106 ms)
    ✓ if mandatory fields given(name), should create a record and respond with a status code of 200 (128 ms)
  GET/ goal by id
    ✓ if given a goal id, should return the goal record (22 ms)
    ✓ if given a goal id that isn't in the database, should return empty array (7 ms)
  GET/ goals
    ✓ if given an auth token, should return the users goals (10 ms)
  PATCH/ update saved amount
    ✓ if given a goal id and a new saved amount,should update the saved amount (14 ms)
    ✓ if amount field is missing, should respond with a status code of 400 (4 ms)
  PUT/ goal
    ✓ if given at least all mandatory fields, should update goal, and respond with a status code of 200 (6 ms)
    ✓ if mandatory fields(name,icon) are missing, should respond with 'goal name is a mandatory field' (6 ms)
  POST/ create goal assingment
    ✓ if given a goal id and an amount, should create a new record and return a status code of 200 (11 ms)
    ✓ if mandatory fields ( amount) are missing, should return a status code of 400 (7 ms)
  POST/ calculate montly amount assigned to goal
    ✓ if given a start and an end date, should return the total amount (16 ms)
  DELETE/ goal
    ✓ should delete the goal (6 ms)

Test Suites: 1 passed, 1 total
Tests:       13 passed, 13 total
Snapshots:   0 total
Time:        1.49 s, estimated 2 s
```

Figure 28. Goal tests

8. Summary and Future Work

8.1. Summary

The objective of this thesis was the implementation of a Personal Finance Management mobile application for facilitating financial planning and management. With the customizable spending and income categories, the user can make them as particular or as general as he/she wants to meet his/her needs. The user can add his/her different wallets using the wallet feature. With the transaction feature, the user can quickly add his/her spending and income, through a simple and intuitive design, so that he/she can monitor them. A new, updated, or deleted transaction immediately updates the wallet, from which the transaction was made, and the budget, if the transaction's category belongs to one. The recurring transactions feature is very useful, since it shortens, even more, the time that the user spends adding his/her transactions. Flexible budgeting is an essential feature of the application since not many related applications offer it. The user can plan his/her future spending, using the budget feature, which allows him/her to customize its duration. The budget feature's flexibility also includes that one budget can consist of one or more categories, making it easier for the user to group similar spending categories in a budget. As it makes sense, a category can only belong to one budget for the budget's specific duration. We need to emphasize the importance of the goal feature since there are not many non-EBS applications that offer it. Having a feature dedicated to saving goals is very important since people's number one financial struggle is not being able to prioritize savings. The goal feature is flexible, easy-to-use, and offers valuable insights. The charts feature presents income and spending money allocation in pie charts, which the user can understand with a glance. Finally, the reports feature presents monthly spending and income summary, for each of the user's categories, and a comparison with the previous months.

The whole process of the application creation was described - starting from the motivation and the used technologies, continuing with the analysis of the related work, and through the User Stories, the functional requirements of our application emerged. The Database, Server, and client Application were presented in the Design and Implementation. In the Database, we describe the database's model entities and visualize them through a database schema. In the Server, the Express server's routes, models, and controllers functions were described. In the Application, after a short

reference on some of React-Native's fundamental concepts, we described the parts of our application's structure which include: screens, navigators, components, redux, services, and utils. In the User Interface, we present the UI of our application, with short descriptions of the screens. Last, but not least, the endpoints of our application were tested, and their results were presented.

8.2. Future Work

In this section we present some ideas for the application that were not completed due to lack of time.

8.2.1. Financial Literacy (FinLit)

The FinLit feature, although it was in the application proposal, was not completed due to lack of time. With this feature, the user will have access to some kind of library, for financial terms and concepts. The concepts will be divided into chapters, like mini-courses. Before letting the user in a course, he/she first will have to answer some related questions with his/her knowledge thus far. These answers then could be potentially used in a FinLit survey. In the course, the concept will be analyzed, and references will be given to the user, for further research. After the completion of the course, the user will be questioned the same as before, so that he/she can test his/her knowledge and we can get data as to how helpful the course was and how we can improve it.

8.2.2. Insights Improvement

Right now, with the charts feature the user can see his/her spending and income allocation divided into categories for any period he/she wants. After this suggested modification, the user will be able to see his/her money allocation divided by wallets too.

The current reports feature offers monthly reports and comparisons with the last month. After this suggested modification, the user will be able to see three-month, six-month, and yearly reports, as well as comparisons for customized periods. While developing this feature it will be important to remember to keep the reports simple.

8.2.3. Greek Language

Adding the greek language to the application and making it available to the greek speaking population will be a very important feature since there are not many applications correctly translated into greek.

9. Appendix

In this chapter we present our components (atoms, molecules, organisms) in detail.

9.1. Atoms

Buttons

Component name	**Component props**	**Component description**
AddEntityButton	text, onPress	is a text button for adding a new entity. The *onPress* handling method navigates the user to the corresponding *AddEntityScreen*.
AddTransactionButton	onPress, style	is an icon button and it is differentiated from the other entities as the transaction is the most frequent entity a user creates, so it needs to be in a more accessible place on the screen and on more than one screen. The *onPress* handling method navigates the user to the *AddTransactionScreen*.
FormButton	action, onPress, color	is a text button inside the add and edit screens. The *onPress* handling method makes a post server request that creates a new record on the database.
CalendarModalDateButton	text, onPress, style	is a text button used in the *CalendarModal* (explained in the organisms). The *onPress* handling method stores the user's choice in a

		redux variable.
BudgetOverspendButton		is a red exclamation icon button that informs the user, through an Alert, that he/she has spent more than his/her budget plan.
CalendalButton	onPress, style	is a calendar icon button that when it is pressed the *CalendarModal* (explained in organisms) is displayed to the user.
CategoryIconButton	icon, onPress, selected	is an icon button that can be selected from a user for his/her category. The *onPress* handling method selects and deselects a button, and the *selected* state variable dictates the color of the button. If the button is selected it is light purple and when it is unselected it is light grey.
EditButton	onPress, color	is an icon button. The onPress handling method when pressed navigates the user to the corresponding edit screen.
DeleteButton	onPress, color	is an icon button. The onPress handling method, when pressed, asks the user a delete validation through an Alert, sends a delete server request, and finally navigates the user to the corresponding screen.
NotificationButton	navigation, when	is an icon button that when pressed notifies the user through an Alert that an upcoming transaction is due today (grey color) or a transaction is overdue (red color) and prompts the user to take action either by

		confirming the transaction or by dismissing it. The *when* prop indicates whether a transaction is due today (when = today) or any day before (when = past).
RecurringTransactionsButton	onPress, style	is an icon button that when pressed navigates the user to the *RecurringTransactionsScreen*.
GoBackButton	navigation	is an icon button that when pressed navigates the user to the previous screen.
SettingsButton	navigation	is an icon button that when pressed navigates the user to the *SettingsNavigator*.
MarkAsPaidButton	onPressMarkAsPaid	renders a TouchableOpacity, and when pressed calls the *onPressMarkAsPaid* method.
DismissButton	onPressDismiss	renders a TouchableOpacity with an “x” icon, and when pressed calls the *onPressDismiss* method.
AuthFormFooter	question, buttonTitle, navigation, whereTo	renders a Text component on the left with the *question* and a TouchableOpacity on the right with the *buttonTitle*. When the button is pressed it navigates the user to the *whereTo* screen.

Table 37. Button atom components

Texts

Component	**Component**	**Component function**

name	**props**	
HeaderTitle	title	renders a Title component using *title*.
EntityTitle	name, balance, entity, icon	displays the *icon* on the left if it has an *icon* prop or a bank icon if the *entity* is "wallet". It also displays the entity's name, and if there is a *balance* prop that is negative, it renders a *BudgetOverspendButton* component.
TransactionText	category, wallet, note, screen	is the text part of a *TransactionListItem* component.
TransactionAmount	categoryType, amount, date, title	Is the amount part of a *TransactionListItem* component. It displays the income with a "+", and the expenses with a "-". If there is a *date* prop, it also displays a date under the amount.
DotAndCategoryName	dotColor, category	renders the props inside a View component.
BoxItem	label, value, style	is a container component that displays *label* and *value*.
UpcomingTransactionFooter	intervalSTR, nectOccurrence	renders the props inside a View component.

Table 38. Text atom components

Charts

Component	**Component**	**Component function**

name	**props**	
CustomProgressCircle	saved, target	renders a *ProgressCircle* component using its props. Also, it displays a percentage (*saved*/*target*) in the middle of the circle. If there is no *target*, it displays the *saved.*
BudgetProgressBar	leftToSpend, onerspend, balance, budgetAmount, budgetName, footerOnTop	displays a *Progress.Bar* component, in full red, if there is an *overspend* prop or a light blue bar with its progress equal to *leftToSpend* if there is a *leftToSpend* prop. If the *footerOnTop* is true, displays the budget's *name* and *balance* over the progress bar, and if it is false, displays the *balance* under the progress bar.
GoalProgressBar	progress, saved, target	renders a Progress.Bar component, using *progress*. Under the progress bar, it displays Saved = *saved* on the left and Goal = *target* on the right.
CustomLineChart	dates, dailySums, dotPositions, todaysPosition	renders the YAxis and the AreaChart components using the *dailySums*. For the x-axis, it renders the *dates* horizontally. Using the *dotPositions* it displays dots on the chart line at the positions of the *dates*. Finally, using the *todaysPostion* it stops the line of the chart at the current day.
CustomPieChart	data	renders a PieChart component using *data*. The labels in the chart, are rendered from the Labels method which uses the G, Line, Circle, and Text components.

Table 39. Chart atom components

Inputs

Component name	**Component props**	**Component function**
CustomTextInput	placeholder, value, keyboardType, isPassword, hidePassword, setHidePassword, style, onChangeFunction	displays a TextInput react-native component and calls the *onChangeFunction*, received from its props when the user inputs a value. If *isPassword* is true, displays an eye icon that when pressed, hides the password.
CustomDropdown	data, placeholder, selected, setSelected, multi, width	renders a Dropdown or a MultiSelect component depending on the *multi* prop. It uses the *selected, setSelectd* state to store the user's choice.
CustomDatePicker	value, reduxSetter, dispatch, minDate, maxDate, previousEndDate, fromStartOfDay, toEndOfDay	renders a container that looks like text input, that when pressed displays a DateTimePicker component. When the user selects a date the *reduxSetter* action is dispatched, and when the user closes the date picker, the selected date is displayed in the container. The minDate, maxDate, and previousEndDate are used to limit the user's selection in some cases. Finally, the fromStartOfDay, and toEndeOfDay are used to change the time when that is needed.

Table 40. Input atom components

Other atoms

Name	Description
CategoryIconForDisplay	is a component that receives an icon from its props and renders an Image component of this icon.
colors	is a file that exports all the colors that are used in the application
categoryIcons	is a file that exports all the icons that are used in the application

Table 41. Other atom components

9.2. Molecules

Listitems

Component name	Component props	Component description
CategoryListItem	item, navigation, entity	renders an Image with the icon, the category name, an *EditButton,* and a *DeleteButton*.
WalletListItem	item, navigation	renders a bank icon, a TouchableOpacity button with the wallet name and balance, an *EditButton,* and a *DeleteButton.* When the text button is pressed, the *onPressWallet* method dispatches the wallet's field actions and calls the *APIRequestGetTransactionsByWallet* method before navigating to the

		WalletDetailsScreen.
WalletListItemForCard	item, navigation	renders a button with the wallet's information, and when pressed the *onPressWallet* method is called.
TransactionListItem	item, navigation, category, wallet, budgetId, walletId, categoryType, icon, dateChanged	renders a *DateTitle* if it is the first transaction of the day or a separating line, and a *Transaction* component. The *onPress* method navigates the user to the *EditTransactionScreen.*
TransactionListItemForCard	item, navigation, category, wallet, budgetId, walletId, categoryType, icon, nextOccurrence, intervalSTR, upcomingOrNot	renders a *Transaction* component, with different props, depending on its *upcomingOrNot* prop.
UpcomingTransactionListItem	item, navigation, category, wallet, budgetId, walletId, categoryType, icon, nextOccurrence, intervalSTR	renders a *DateTitle* and a *Transaction* component. In the *DateTitle* it passes *onPressMarkAsPaid* and *onPressDismiss* methods as props. The first one creates a new transaction record with the *APIRequestCreateTransaction* and updates the previous transaction with the *APIRequestUpdateUpcomingTransaction*. The second one updates the nextOccurence field of the previous transaction with the *APIRequestUpdateNextOccurrence* method.

BudgetListItem	item, navigation, categoriesNames, period, periodChanged	renders a period title text if this budget is the first of a different time period and a TouchableOpacity with a *BudgetBarChart* component. When the button is pressed, the onPress method, calls the *findCategoriesIdsOfBudget*, *APIRequestGetDailySumAmounts*, *getDaysOfBudget,* and *findXAxisData* methods, before navigating to the *BudgetDetailsScreen*.
BudgetListItemForCard	item, navigation, categoriesNames	like the BudgetListItem renders a TouchableOpacity with a *BudgetBarChart* component with slightly different props and without time period titles.
GoalListItem	item, navigation	renders an Image with the goal icon and a TouchableOpacity button with a *GoalProgressBar* component. The *onPress* method navigates the user to the *GoalDetailsScreen*.
ReportListItem	item, navigation	renders a TouchableOpacity button, featuring a month, its total income and total expenses.

Table 42. ListItem molcule components

Other molecules

Component name	**Component props**	**Component description**
BudgetBarChart	budgetName, balance, budgetAmount, categoriesNames,	displays the *budgetName* (if *dontShowName*, *dontShowCategories* are null), for each of the

	dontShowName, dontShowCategories	*categoriesNames* renders a *DotAndCategory* component (if *dontShowCategories* is null), and alaways renders a *BudgetProgressBar* component with different props depending on whether the *progress* is zero, or the *balance* is positive or negative.
LabeledInput	placeholder, label, value, keyboardType, isPassword, hidePassword, setHidePassword, style, onChangeFunction	Renders a Text label, and a *CustomTextInput* component with its props.
Transaction	item, navigation, icon, wallet, category, budgetId, walletId, categoryType, transactionDate, nextOccurrence, intervalSTR, screen	renders a TouchableOpacity button. The button consists of an Image of the icon, a *TransactionText* component, a *TransactionAmount* component, and a *UpcomingTransactionFooter* component if the *screen* equals “upcoming”.
CalendarModalCustomDate	selectedButton, setSelectedButton, setFromDate, setToDate, setCustomTitle	renders the fourth *CalendarModalButton* for the custom date range. If this button is selected the *selectedButton* state and the *customTitle* redux state are set to “custom”, and a new container is displayed under the button. The container consists of two *CustomDatePicker* components, in

		which the user can select the date range.
CalendarModalDateRangeButtons	selectedButton, setSelectedButton, setFromDate, setToDate, setTitile, setCustomTitle	renders three *CalendarModalButton* components for specific date ranges, and one *CalendarModalCustomDate* component.

Table 43. Other molecule components

9.3. Organisms

Headers

Component name	**Component props**	**Component description**
Header	back, title, settingsButton, navigation, editButton, onPressEdit, deleteButton, onPressDelete, headerColor	is a header that consists of three parts. The first part is the *GoBackButton* component on the left and is not always displayed. The second part is the *HeaderTitle* component. And the third part, at the right of the screen, depends on the Header's props and can be one of the following: a *SettingsButton,* an *EditButton,* or a *DeleteButton* component.
HeaderTabNavigator	leftName, rightName, LeftComponent, RightComponent, params, transactionsList, style	is an interface component for the material top tab navigators of the application. It consists of a left and a right header tab. The names, screen components, and the initial parameters of the tabs come from the component's props.

HeaderWithDateRange	navigation, title, entity	is a header that depending on its prop *entity* renders two different views. It always renders a *DateRangeText* atom component with the *title*. If the *entity* is "transaction" then the component renders two icon buttons. The *CalendarButton* and the *RecurringTransactionsButton* atom components.
BudgetHeader	name, balance, budgetAmount	consists of a budget's name, rendered by the *EntityTitle* atom component, and some details for the budget that receives from its props which are rendered with *BoxItem* atom components.
GoalHeader	name, icon, targetDate, thisMonth, thridBoxTitle, thirdBoxValue	like the BudgetHeader, consists of a goal's name, rendered by the *EntityTitle* atom component, and two or three *BoxItem* atom components. If the goal has a *targetDate*, there is a BoxItem with the date. The second Box item is "Added this month" and it is common in all the goals. The third BoxItem can be measured in months or euros, depending on the goal's type.
WalletHeader	name, balance	consists of a wallet's name, rendered by the *EntityTitle* atom component, and a BoxItem atom component displaying the wallet's balance.

Table 44. Header organism components

ChartCards

Component	**Component**	**Component description**

name	props	
PieChart	sum, dataList, title	is a card-like container with a title, that renders a *CustomPieChart* atom component on the left and a list of *DotAndCategoryName* atom components on the right, using the dataList.

Table 45. chartCard organism component

HomeScreenCards

Component name	Component props	Component description
SummaryCardForHome	props	consists of a pie chart with the income and the spending on the left. It displays the income, expenses, and balance values for the current month on the right. It reads the *spendingSum* and *incomeSum* redux states and makes a list that sends to the *PieChart* atom component through props.
WalletsCardForHome	props	displays the user's wallets and their balance in a card container titled "Wallets".
TransactionsCardForHome	props, upcoming, latest	works for both the latest transactions and the planned ones. It is a card container titled "Latest transactions" or "Upcoming planned transactions", and renders a *TransactionList* component with the *transactionsList* or *reccuringTransactionsList* states that it reads from the redux store. At the end of the list, there is a "SHOW MORE" button that navigates the user to the WalletNavigator (the WalletsScreen) or the ReccuringTransactionsScreen.

BudgetsCardForHome	props	is a card container titled “Budgets” and renders the *BudgetList* component. At the end of the list, there is a “SHOW MORE” button that navigates the user to the EnvelopesNavigator (the BudgetsScreen).
GoalsCardForHome	props	is a card container titled “Goals” and renders the *GoalList* component. At the end of the list, there is a “SHOW MORE” button that navigates the user to the EnvelopesNavigator (the GoalsScreen).

Table 46. HomeScreenCard organism components

Lists

Component name	**Component props**	**Component description**
CategoryList	props, list (incomeList/spendingList), entity	In the *renderItem* method, it renders the *CategoryListItem* component only for the user’s categories that have their *hide* field null.
WalletList	props, width, entity	In the *renderItem* method, it renders the *WalletListItem* component or the *WalletListItemForCard* component, depending on the value of the *entity.* It reads the *walletsList* state from the redux store.
TransactionList	props, width, screen, trasnactionsList, onPress, upcoming, style	In the *renderItem* method, it calls the following utils methods: *findWallet*, *findCategory*, *findCategoryIcon*, *findCategoryType*, *findBudgetId*. Then it renders the *TransactionListItem* or the *TransactionListItemForCard,* or the *UpcomingTransactionListItem* depending on

		the values of the *screen* and the *upcoming*.
BudgetList	props, width, entity	In the *renderItem* method, it renders the *BudgetListItem* component or the *BudgetListItemForCard* component, depending on the value of the *entity*. For both those components, the renderItem finds the budget's categories names by calling the *makeListOfBudgetCategories* util method. For the *BudgetListItem,* it sorts and categorizes the budgets in time periods like "This week", "This month", etc. In the *renderItem* method, it checks if the current item's period is changed from the last's and passes that information to the *BudgetListItem* component. Finally, using the *isBudgetExpired* util method finds any budgets whose endDate has passed, and by calling the *APIRequestDeleteCategoriesFromBudget* method dissociates the budget with the categories, and makes the categories available for the user to use in new budgets.
GoalList	props, width	In the *renderItem* method, it renders the *GoalListItem* component or the *GoalListItemForCard* component, depending on the value of the *entity*. It reads the *goalsList* state from the redux store.
ReportList	props, months	In the *renderItem* method, it renders the *ReportListItem* component.

Table 47. List organism components

Modals

Component name	**Component props**	**Component description**
CalendarModal	entity, navigation, position	displays a container that renders the *CalendarModalDateRangeButtons* component, and a check icon button at the bottom that when pressed, the *onPressChangeDateRange* method is called. That method, uses the states that were set from the user's choices and makes an *APIRequestGetTransactionsSums* call if entity equals "chart", or an *APIRequestGetTRansactionsByDateRange* if entity equals " transaction".
AddSavedAmount Modal	props, goalId, monthlySum, visible, setVisible	displays a container that renders a *CustomDropdown* component with the two signs (+,-), a *CustomTextInput* component for the assignment amount, and a check icon button at the bottom, that when pressed the *onPress* method is called. The method updates everything that is related to the goal assignment by calling the following services methods in this order: *APIRequestGetGoalSavedAmount*, *APIRequestAddSavedAmount*, *APIRequestGetGoalSavedAmount*, *APIRequestGetMonthlySum*, *APIRequestGetGoals*.

Table 48. Modal organism components

References

[1] Jason Fernando, Financial Literacy Definition. https://www.investopedia.com/terms/f/financial-literacy.asp

[2] Leora Klapper, Annamaria Lusardi, Peter van Oudheudsen. Financial Literacy Around the World, INSIGHTS FROM THE STANDARD & POOR'S RATINGS SERVICES GLOBAL FINANCIAL LITERACY SURVEY, https://gflec.org/initiatives/sp-global-finlit-survey/

[3] Ashley McCann. Budgeting Terms: The YNAB Dictionary of Definitions, June 2021, https://www.youneedabudget.com/the-ynab-dictionary-for-budgeting/

[4] JavaScript, https://www.javascript.com/

[5] React-Native, https://reactnative.dev/

[6] JSX, https://reactjs.org/docs/introducing-jsx.html

[7] Expo, https://expo.dev/

[8] Expo Go, https://expo.dev/client

[9] Node.js, https://nodejs.org/en/

[10] Express, https://expressjs.com/

[11] Axios, https://axios-http.com/docs/intro

[12] MySQL, https://www.mysql.com/

[13] Azure, Microsoft. What is a relational database? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-relational-database/#whatis

[14] JSON Web Token (JWT), https://jwt.io/

[15] IBM Documentation. The HTTP protocol, https://www.ibm.com/docs/en/cics-ts/5.3?topic=concepts-http-protocol

[16] Cesare Ferrari, Working with the data model in Express, https://dev.to/cesareferrari/working-with-a-data-model-in-express-hc

[17] Express Tutorial, https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes

[18] React Fundamentals, https://reactnative.dev/docs/intro-react

[19] Core Components and Native Components, React Native Docs, https://reactnative.dev/docs/intro-react-native-components

[20] Expo Docs, https://docs.expo.dev/guides/routing-and-navigation/

[21] React Navigation Docs, https://reactnavigation.org/docs/custom-navigators

[22] Brad Frost, Atomic Design, https://bradfrost.com/blog/post/atomic-web-design/

[23] Redux Docs, https://redux.js.org/introduction/getting-started