

# The Investigation and Development of a Personal Finance Tool to Improve Financial Capability

**Kiran Sanganee**

Department of Computer Science

University of Warwick

Supervised by Sara Kalvala

Year of Study: 3<sup>rd</sup>

14 April 2023

## **Abstract**

Your abstract goes here. This should be about 2-3 paragraphs summarising the motivation for your project and the main outcomes (software, results, etc.) of your project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Current services . . . . .	6
2.1.1	Manual Importing Applications . . . . .	7
2.1.2	Mobile Banking Applications . . . . .	8
2.1.3	Paid Applications . . . . .	9
2.2	Open Banking . . . . .	10
2.2.1	Plaid . . . . .	10
2.3	Further Motivation . . . . .	13
<b>3</b>	<b>Design</b>	<b>14</b>
3.1	Technologies . . . . .	14
3.1.1	Next.js . . . . .	14
3.1.2	TailwindCSS . . . . .	15
3.1.3	PocketBase . . . . .	16
3.1.4	Python and TensorFlow . . . . .	16
3.2	UI . . . . .	17
3.2.1	Authentication Pages . . . . .	18
3.2.2	Dashboard . . . . .	19

*The Investigation and Development of a Personal Finance Tool to Improve Financial Capability* 2

3.3	Strategies . . . . .	20
3.3.1	Transactions . . . . .	20
3.3.2	Categories . . . . .	21
3.3.3	Budgets . . . . .	22
3.3.4	Investments . . . . .	23
3.4	Budget Prediction . . . . .	25
3.4.1	Manual Pattern Identification . . . . .	25
3.4.2	Linear Regression . . . . .	25
3.4.3	Neural Network . . . . .	26
3.4.4	Conclusion . . . . .	27
3.5	Software Architecture . . . . .	28
3.6	Endpoints . . . . .	29
3.7	Legal, Ethical and Professional Considerations . . . . .	29
<b>4</b>	<b>Implementation</b>	<b>30</b>
4.1	Project Management . . . . .	30
4.1.1	Research and Design . . . . .	31
4.1.2	Proof of Concepts . . . . .	32
4.1.3	Repeated Strategy Implementation . . . . .	37
4.1.4	Budget Prediction . . . . .	40
4.2	Strategies . . . . .	41
4.2.1	Active Bank Accounts . . . . .	41
4.2.2	Transactions . . . . .	42
4.2.3	Categories . . . . .	45
4.2.4	Budgets . . . . .	47
4.2.5	Investments . . . . .	51
4.3	Endpoints . . . . .	54

<i>The Investigation and Development of a Personal Finance Tool to Improve Financial Capability</i>	3
<b>5 Evaluation</b>	<b>56</b>
<b>6 Conclusions</b>	<b>57</b>
6.1 Future work . . . . .	57

# Chapter 1

## Introduction

The term personal finance tool encompasses a wide range of software, including budgeting tools, investment management software and credit score calculators. This project aims to investigate strategies which help build a user's financial capability and confidence; followed by the implementation of a web application to support these strategies. In particular, the research focused on strategies to help individuals to manage their, often several, bank accounts and expenses in one place.

Being financially capable is defined by the Financial Educators Council as having the "skills and knowledge of financial matters to confidently take effective action that best fulfills an individual's personal, family and global community goals" [14]. The website therefore aims to give user's the tools to view their financial circumstances, as part of the knowledge, and with it comes the confidence to make informed beneficial decisions. For the rest of this document, financial capability and financial literacy will be used interchangeably.

Part of the motivation for this kind of project comes from the recent open banking technology movement [15]. This is where thousands of major banks have opened a set of endpoints to allow third party applications to securely access their customers' financial data with authorisation. It allows developers to build personalised financial applications and services that are tailored to the needs of the user, based on this data; and is most appropriate for this project. Further detail and other motivations are discussed in the background chapter

below (2).

## Chapter 2

# Background

The purpose of this section is to provide background information into the problem area, as well as introduce and explain concepts that will be used throughout the rest of the document.

### 2.1 Current services

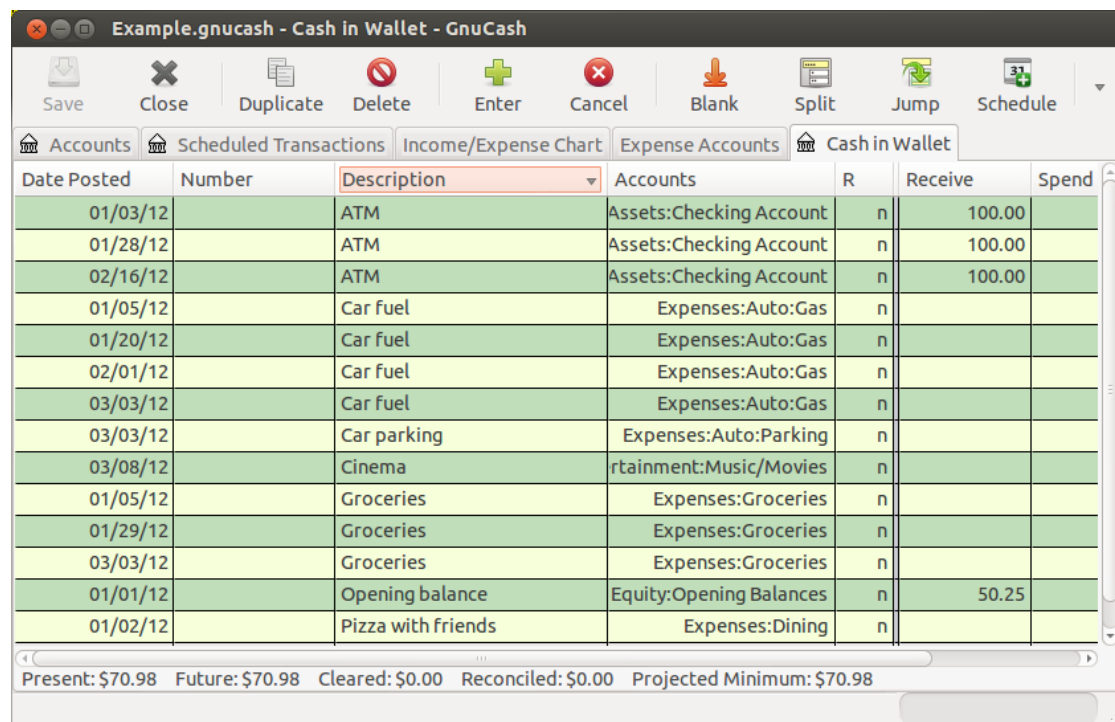
Understanding the current available applications and their limitations is important. It helps to narrow down the problem area, and identifies the requirements for the new system.

It is fair to say that there are a wide range of personal finance applications available, however, they are not all suitable for the same use case. For the purpose of this project, the focus is on those that help provide a user with an overview of their finances, as these are most applicable in improving a user's financial literacy. These existing tools can be divided into three main categories. Firstly, there are the applications that do not utilise open banking, and as such require manually importing the data. Secondly, there are the mobile banking applications that do use open banking, but instead often focus on single accounts, so not a big overview. Finally, there are applications which you often have to pay for, or are filled with advertisements and furthermore, lack analytics.



### **2.1.1 Manual Importing Applications**

The first category of applications are those that do not use open banking. Often, these tools are older and lack updates which is why they haven't been improved to utilise the end points offered in open banking. The most prominent example of this GNU Cash [4]. This is a free and open source application that allows users to track their finances. It is a desktop application and very powerful tool, however, it is not very user friendly and requires a lot of experience using it before it is effectively used. Switch to Linux mentions that it is a "great FOSS tool [...], but it can be complicated to setup" [24], as part of their tutorial on how to use it. The fact that there are several tutorials and little documentation demonstrates that the UI is difficult to use, which is made worse from it looking very outdated and complex (see figure 2.1). The main weakness that software like this has is that it requires the user to manually import their data. Most users do not have all their accounts and transactions readily available in a structured format, and this even worse for when the user would want it to update live with their recent transactions. Not many aspects of these applications are useful for improving financial literacy, so few will be used in the final web application.



The screenshot shows the GNU Cash application window titled 'Example.gnucash - Cash in Wallet - GnuCash'. The window has a menu bar with 'Accounts', 'Scheduled Transactions', 'Income/Expense Chart', 'Expense Accounts', and 'Cash in Wallet'. Below the menu bar is a toolbar with icons for Save, Close, Duplicate, Delete, Enter, Cancel, Blank, Split, Jump, and Schedule. The main area displays a table of transactions with columns: Date Posted, Number, Description, Accounts, R, Receive, and Spend. The table contains 15 rows of data, including ATM withdrawals, car fuel, car parking, cinema, groceries, and an opening balance. At the bottom, a status bar shows: Present: \$70.98, Future: \$70.98, Cleared: \$0.00, Reconciled: \$0.00, Projected Minimum: \$70.98.

Date Posted	Number	Description	Accounts	R	Receive	Spend
01/03/12		ATM	Assets:Checking Account	n	100.00	
01/28/12		ATM	Assets:Checking Account	n	100.00	
02/16/12		ATM	Assets:Checking Account	n	100.00	
01/05/12		Car fuel	Expenses:Auto:Gas	n		
01/20/12		Car fuel	Expenses:Auto:Gas	n		
02/01/12		Car fuel	Expenses:Auto:Gas	n		
03/03/12		Car fuel	Expenses:Auto:Gas	n		
03/03/12		Car parking	Expenses:Auto:Parking	n		
03/08/12		Cinema	rtainment:Music/Movies	n		
01/05/12		Groceries	Expenses:Groceries	n		
01/29/12		Groceries	Expenses:Groceries	n		
03/03/12		Groceries	Expenses:Groceries	n		
01/01/12		Opening balance	Equity:Opening Balances	n	50.25	
01/02/12		Pizza with friends	Expenses:Dining	n		

Present: \$70.98 Future: \$70.98 Cleared: \$0.00 Reconciled: \$0.00 Projected Minimum: \$70.98

Figure 2.1: Example GNU Cash UI [8]

## 2.1.2 Mobile Banking Applications

These applications are the default applications that come with each bank. Almost always, if the bank offers a website for online banking, they also offer a mobile application that perform the same functions. These include the major banks such as HSBC, NatWest, Lloyds etc., but also includes the online-only banks such as Monzo and Revolut. Sometimes, these applications do not use open banking as instead just display information about the accounts with that specific company, so do not need to use the services of other banks. There are some that do connect with other banks, but they often don't incorporate the information into the analytics, and instead just display the balances.

Taking Revolut [22] as a case study, we can find its strengths to incorporate, as well as its weaknesses to avoid in the web application. Firstly to even use the Revolut's app functionality, a user must open a bank account with the service and prove their identity. Although this does provide the user with

confidence that their information is secure and personalised, it also means it is more difficult to use, slower to gain access to the information and overall just has a catch. It also means that the primary purpose of the app is not to provide a user with an overview of their finances, but instead to provide a banking service, so will not be aimed at improving financial capability. There are some features of Revolut which are worth acknowledging, such as the UI and ability to track expenditure. In having a slick and intuitive user interface, it enables users to understand all aspects of their finances and helps build confidence. The expenditure tracking is particularly useful, it allows users to customise a budget for a set time period and shows what and when they have spent money in during this period; overall aiming to help them stay within the budget. Revolut is a service that does utilise open banking as allows to you connect the app to other bank accounts, however it doesn't incorporate these into the budgets and only really includes them in the net worth section. This is a good example of a service that does use open banking, but doesn't utilise it to its full potential.

### **2.1.3 Paid Applications**

The final category of applications is really just the 'other' section, however the majority of these do incur costs to use. These applications are often more powerful than the free alternatives, yet are often filled with advertisements and are not as user friendly. A good example of this is Quicken which won the awards for the best budgeting app in 2020 and 2021 [11], but costs up to £10/month. It utilises open banking effectively to work with many different bank accounts, yet it does not enable quick-toggling of bank accounts to incorporate/ignore during the analysis and overview. This is a feature which would be particularly useful in giving analysis and a better overview as a user would be able to segment e.g. all the savings accounts. Despite having some effective budgeting features, it lacks the ability to perform budgeting prediction from patterns in expenditure. This feature also would be useful in improving financial capability because it will help the user plan for future expenses and identify areas where they can save money. Overall, these paid applications have some aspects which would be useful in the web application, but they

also are missing some basic ones which would be more focused on improving financial literacy.

## **2.2 Open Banking**

Open banking is defined as "APIs [that] enables third-party developers to build applications and services around the financial institution", in this paper [20]. The open banking movement, is therefore the recent pressure on banks to open up their data to third-party developers. They do this by creating a set of endpoints which developers can query, and will respond with accurate and live data. For example, a user would first login to their online bank via a popup, this would then return an authentication token which the website can use to query e.g. to get their recent transactions; the information is then returned in a secure response. Each bank has their own set of endpoints such that the authentication process and available information differs across them all. This is where a lot of problems arise as reading the documentation is not straightforward and time consuming, however it is necessary to enable the web application to support all the major banks. This lack of standardisation is where Plaid [17] come in.

### **2.2.1 Plaid**

Plaid is a platform that, effectively, wraps all the endpoints provided by each individual bank and provides a single standardised set of REST API endpoints as URLs. This means that the web application only needs to query Plaid, and in turn will be able to support all the banks that Plaid provide access to - over twelve-thousand institutions [18].

To use Plaid's endpoints, a strict authentication flow must be followed as it is handling very private data. Plaid has three different types of tokens as part of this flow. The first is the link token; Link is the name for the widget that is used to authenticate the user with their bank but to do so requires a link token. They can simply be requested from Plaid's API and are valid for 4 hours, but are not tied to any user and are not treated as passwords. Secondly, there are

public tokens; these are what the link widget returns to the web application after the user has authenticated with their bank and so are unique to that user. They also are not treated as passwords as are valid for only 30 minutes and cannot be used to access a user's private information directly. They must be exchanged for an access token, which is the third type of token. This exchange is done via a Plaid endpoint but must be done via the web application's server, rather than client because the response access token must be treated like a password and be kept extremely securely. If it was done on the client, it risks being exposed and anyone with this token can access that user's information. Embedded within the access tokens can be a set of bank accounts, and a user may have several access tokens tied to them.

The flow is therefore: the client requests a link token on behalf of the user, the response link token is then passed to the link widget where the user signs in with their bank; the link widget returns a public token to the client which is passed to the application's backend; the backend exchanges the public token for an access token and then stores this in the database in such a way that only that user can access it. Whenever the client wants to gain access to that user's transactions, it queries the server acting as a proxy. The server identifies which user is asking for the information and attaches their access token to a new request to Plaid. The Plaid response is then forwarded back to the client and the client can display the information to the user. This flow is shown in Figure 2.2.

Further aspects of Plaid relevant to this project include the different modes of operation. When the web application makes queries to Plaid's endpoints, one of three modes can be specified and each produce different results. The first is sandbox mode, this is the default that everyone has access to. All the endpoints and request formats remain the same, but the responses are fake data generated by Plaid themselves. This is useful for testing the web application without having to connect to a bank so do not have to worry about leaking any private information, as well as getting fast and reliable responses. The second is development mode where the responses are real data. The full authentication flow must be followed and a user will login to their bank. To gain access to this mode, it must be requested by Plaid and they review the application to check

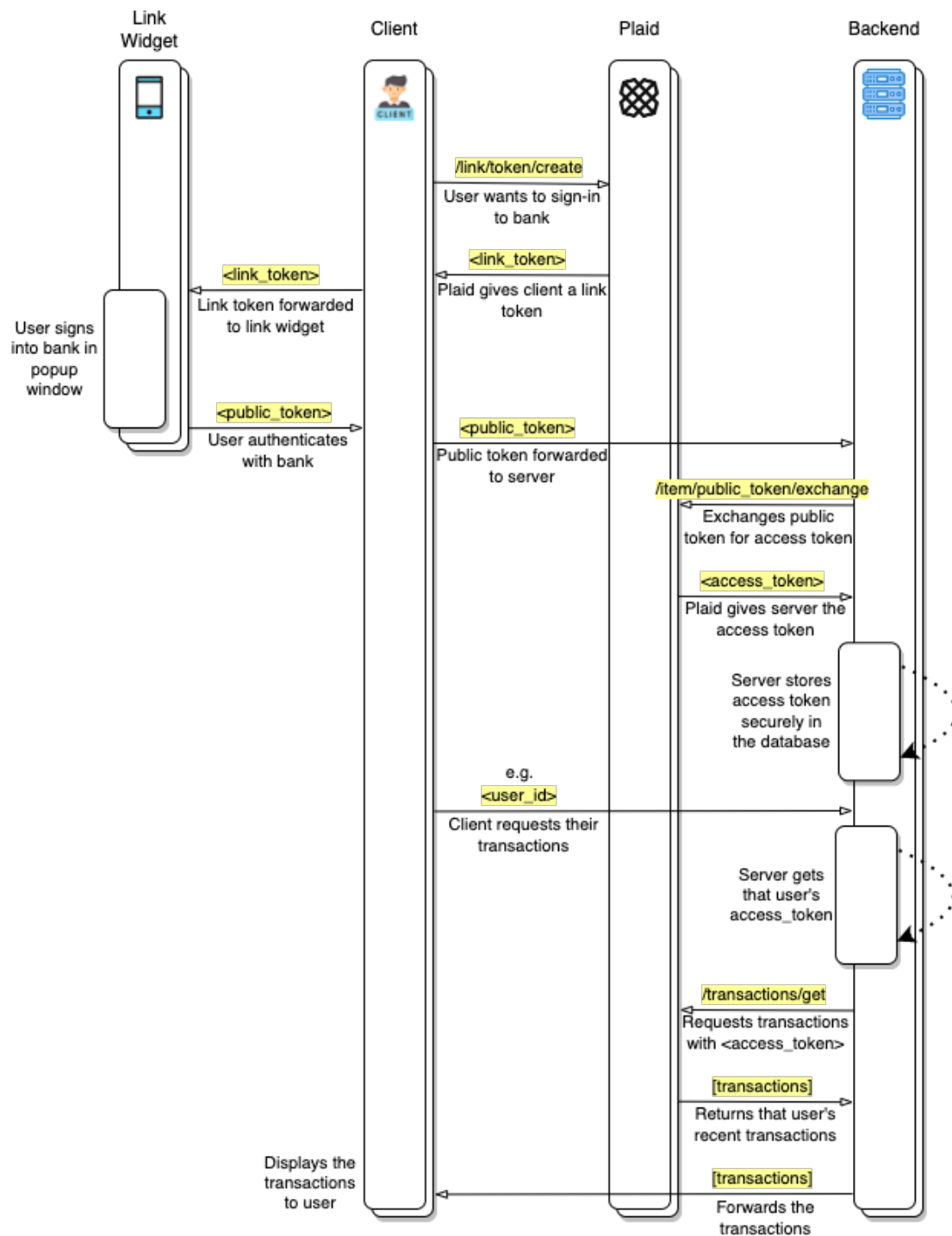


Figure 2.2: Plaid Authentication Flow

that it is secure and will not leak any information. Once given access, there is a limit of one-hundred user accounts connected to actual banks, so this had to be managed carefully. The final mode is production mode, this is the same as development mode but with no limits. To gain access to this, further vetting is required and the queries start to cost. For this project, only sandbox mode was initially used, then following the completion of the prototype, development mode was requested and given for further testing and demonstrations.

## **2.3 Further Motivation**

As previously mentioned, the main motivation for this project came from the recent open banking movement, so using modern technology to help push the field. In addition to this, the limited current services available was also a factor as it was identified that there was a need for an application like this. Finally, the timing for this project is very appropriate as the UK government announced that there is a financial crisis currently going on, which they named the 'Cost of Living Crisis' [9]. Furthermore, in the Money and Pensions Service's recent report, they advised that people get a budget planner to help cope with the current times. This then helped concrete an objective for the web application to include budgeting features, as would also help improve financial capability.

## Chapter 3

# Design

This chapter will aim to describe the design of the web application and how it supports the overall goal of the project.

### 3.1 Technologies

As mentioned in the background section 2, the web application will be utilising Plaid for the open banking aspect. This requires a frontend and a backend to be developed to follow the authentication flow for the ensured security. The majority of the personal finance tool will be written in JavaScript and will be using the Next.js framework.

#### 3.1.1 Next.js

Next.js is a full-stack React framework by vercel [25]. It incorporates the React library for the frontend, as well as supporting API routes for the backend, meaning it is an ideal framework for this project. Other benefits of using Next.js include the ability to perform server side rendering, which will improve the initial loading time as well as making the component development much less complicated. In addition, Next.js has in built automatic optimisations to improve building and serving times. The queries to get, for example,



a user's transactions go through 3 stages. The first is the initial query from client to Next.js API route; then this accesses the database and then queries Plaid's endpoints; finally Plaid makes the query to the bank and response is propagated back. This means that any optimisations to improve the speed and help avoid making the loading times of data feel slow are valuable and what Next.js provides by default. Next.js is a popular web application framework with over one-hundred-thousand stars on GitHub [26], therefore has a large community and is well documented such that it is easy to find solutions to any common problems that may arise. Finally, Next.js is a framework that focuses on the ease of development to maximise productivity, and so means the whole experience will be more enjoyable and good software engineering practices can be followed.

### **3.1.2 TailwindCSS**

As part of the Next.js environment, the website serves jsx components that can be styled as normal HTML elements with CSS. TailwindCSS is a framework that contains a large amount of utility CSS classes. The application incorporates these classes to completely customise the way the website looks. Another consideration for helping to build the UI was to use a component library. This may have cut down on development time, but often doesn't look as appealing and often look quite generic, as have limited ability to customise. TailwindCSS is more flexible and enabled the UI to be built to the exact specification of the design and the developer's vision. Similar to Next.js, TailwindCSS is also quite popular, so has a large amount of resources and cheat sheets to aid development. Often Next.js and TailwindCSS are used in combination, such as in the t3 web stack, as they synergise well together [23].

The UI is a big focus for this project, as is one of the limitations in the current systems that perform similar tasks; the background research found that their interfaces are often ugly and unintuitive. A lot of time was spent prototyping and designing the components before the functionality was actually implemented into them. By having an appealing UI, the user will have more confidence in the application and therefore their finances, which is a key part

of financial capability. Furthermore, provided that the user's data is displayed in an informative and easy to understand way, they are more likely to absorb the information and therefore make better financial decisions.

### **3.1.3 PocketBase**

Part of Plaid's authentication flow requires the access tokens are associated with only a single particular user, however that user may have several access tokens. To support this not only is a database needed for basic user authentication to signup and login, but also to store all the access tokens. PocketBase was determined to be most appropriate for these use cases due to its minimalism and fast setup.

PocketBase is an "open source backend consisting of embedded database (SQLite) with realtime subscriptions, built-in auth management, convenient dashboard UI and simple REST-ish API" [19]. It is run as a single executable file and the Next.js API routes can connect to it locally. In addition, the built-in authentication management is for the user to be able to login and persist their session across refreshes, this makes the user experience of the application much more seamless. Furthermore, means less time is spent on development for this basic feature. The dashboard UI is also useful for the developer to manage users and aid development as can help visualise the database structure as well as manage the tables inside. Finally, PocketBase has a JavaScript SDK that can be used to connect to the database from the API routes. This means less time is spent on ensuring the REST requests are all correct as instead is a simple function call.

### **3.1.4 Python and TensorFlow**

Following the research into strategies which would help build financial capability, the incorporation of budget prediction was proposed. The thought process and explanation as to why the project uses a recurrent neural network is explained later in this section, but it was decided that implementing it would be done with Python, and in particular, using the TensorFlow library. TensorFlow is a machine learning platform that is used to help build and optimise

many machine learning models.

By using these technologies for the machine learning aspect of the project, it allows faster development. Python is a language that is often used for machine learning for a variety of reasons. Nazar Kvartalnyi [12] comments that some of these reasons include the fact that it is simple, consistent and intuitive; that there are a variety of libraries and frameworks such as TensorFlow that support the process; and that there is a great community for giving support during development. Furthermore, the developer of the application already has experience with Python and TensorFlow so there would be little learning toll to implement the budget prediction.

Using Python does have some drawbacks such as the extra complexity and need for the application to communicate with the neural network. To overcome this, once the neural network had been trained, it was hosted with a python flask server such that it could be treated like an API. It would have a single route that would take the input data, and then response with the output of the neural network. This simple solution meant more time could be focused on ensuring the neural network is accurate and precise.

## **3.2 UI**

As mentioned earlier, there was a big focus on making the user interface appealing and intuitive. By working with TailWindCSS, prototype designs could be made in software such as Figma, and then the developer could match the design exactly with appropriate CSS classes and JavaScript functionality. The design process that was opted for in the end, however was to first give the components their basic functionality; then perform adequate component-based styling; then go back to the functionality and modify if appropriate; and then finally finish of the styling with the whole page in mind and make sure it fits in well. This process was more suitable for the project as it meant that the websites functionality was not compromised by the styling, but also the on-the-fly styling is a added benefit from TailWindCSS classes as allows for quick viewing of designs and changes.

### 3.2.1 Authentication Pages

When a user first visits the site, they must first create an account. From research into the limitations of other services, it was noted that having a quick and seamless experience at this stage was important for the user. This is in contrast to applications like Monzo and Revolut, where the signup process is long as they are not signed up for the analytics, but also making a bank account. The focus here was simplicity so the design for these pages were simple input boxes and a functionality button to login or create an account.

Personal Finance Tool

(query)

email

password

Don't have an account?

Login

Figure 3.1: Login Page

Above is the basic design for these pages. Most of it is self explanatory, however where it says "(query)" is where any message or response can be displayed to the user. These include "you have successfully logged out", or "that user does not exist", or "incorrect password", and will be given the relevant colours. The "don't have an account?" and "login" text will be modified appropriately for each page that it is on - for example if instead the user is creating an account, it will say "already have an account?" and "signup" respectively.

This was the first and only page that was styled from a draw.io specification as the process of doing this was found quite slow and tedious. It didn't maximise the value of using TailwindCSS to quickly add the CSS classes and perform

dynamic styling. For the rest of the pages, the more efficient process outlined in the introduction to this section was followed.

### 3.2.2 Dashboard

Once the minimum strategies for the web application had been determined - transactions, categories, budgets and investments. It was decided that this information was to be presented on a single central page, and the ability to switch between the strategies is done via a header bar. The was so quick access to all the information is possible, but also so that strategies not being viewed can be preloaded in the background to avoid long waits for the information to load, ultimately improving the user experience.



Figure 3.2: Dashboard Navigation Bar

The header bar shown above (3.2) was initial design for this dashboard (and minimally changed in the final product). By default, the transactions strategy is chosen. When hovering on the other strategies, a light grey bar underneath the strategy name appears; then being selected turns this bar orange and removes the bar from the old strategy. Underneath this bar is the information provided for that strategy, for example in the transactions strategy, below will be all the transactions.

A limitation in the current systems that was outlined earlier is the ability to include several accounts. Furthermore, of the few that do have this feature, none of them have a way to easily enable/disable the incorporation of these accounts in the analytics. Therefore, this aspect was key when thinking about the design of the header bar. On the right in a slightly different bar, there is an accounts dropdown. When clicking on this, all the bank accounts that the user as linked will be displayed hear, along with a toggle button to enable/disable

the account. Only enabled accounts will be included in the analytics, and when switching between strategies, only the selected accounts will persist. An ideal use case for this feature is when a user has several accounts, but only wants to see their transactions on their savings accounts. They can disable all the other accounts to do so. Following this, they can quickly re-enable the other accounts when they want to see them all again by just clicking the toggle button.

### **3.3 Strategies**

The only pages in the whole application are the two described above (authentication and dashboard). Each of the strategies will be implemented as separate components but will appear on the dashboard page. This means that each component ought to be roughly styled in the same way as to match the general theme.

#### **3.3.1 Transactions**

This is the default strategy that is shown on the dashboard because it contains the information that would be the most useful in helping the user can an overview of their finances. By viewing their recent transactions, they can see where money is being spent and where it is coming from, and then make adjustments accordingly. A lot of traditional banks to not allow quick access to this information, and viewing transactions for all the accounts that have been toggled on can be extremely useful in identifying expenses that are not necessary.

From Plaid, the endpoint returns a list of transactions in reverse chronological order, per account. Each transaction has `account_id`, `amount`, `date`, `time`, `location`, `currency code`, `merchant name`, `category` and more. The web application will group the transactions from all toggled accounts by their date, then display each transaction in reverse chronological order. The transaction will be displayed with the merchant name, category, amount and the bank logo of the account it is from. For each date there will be a separate label above all the transactions for that date. Finally a distinction between income and expendit-

ure will be made by colouring the whole transaction background light-green for income and light-red for expenditure.

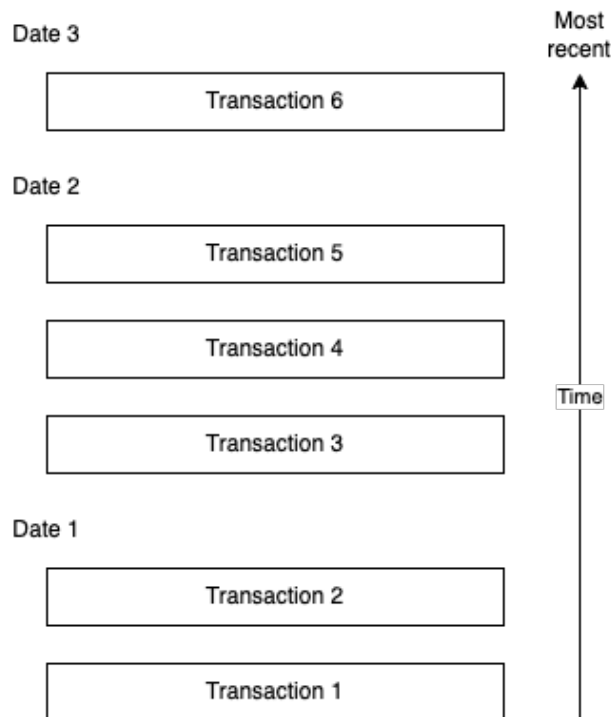


Figure 3.3: Reverse Chronological and Date Grouped of Transactions

### 3.3.2 Categories

The categories strategy would also be helpful at identifying unnecessary expenses. In this case, the application only takes the previous thirty days, and also only takes expenditure. It will list all the transactions in reverse chronological order again, but this time not grouped by anything. A pie chart will also be displayed with the values being the total amounts for each category. The user can then see which category they are spending the most in, and will be able to see the total amounts. In addition, they will be able to group each transaction by it's category and show only the transactions for that category by selecting it. All the transaction data will again be only for the toggled accounts.

The difference between this strategy and the transactions strategy is that in the transaction strategy, the emphasis will be on if the transactions were income or expenditure and to which bank account; whereas in the categories strategy, the emphasis will be on the category of the transaction and the total amount spent in that category. The input data will be the same, but it is a different way of presenting the data such that the user can get all the perspectives on their finances, and ultimately make better decisions.

### **3.3.3 Budgets**

Two different methods of performing some analytics of budgets were considered for this strategy. The first method was giving the user a budget breakdown based on their income. Following some research, it was found that for the average person, utilising a 50:30:20 budget is extremely effective at helping people spend money on the necessities, still having some money for treats and also saving some money. The article [13] by N26, a well respected European bank, explains the strategy in more detail. In essence, it says to spend 50% of your income on needs, 30% of your income on wants, and 20% of your income into savings. Several other articles back up the effectiveness of this strategy and help give it credibility; [21] even talks about how this method carries over into the cost of living crisis.

The method does have some disadvantages however. The major one being that it is extremely difficult to narrow down which category each expense falls under. Often people will disagree what is considered essential, so is already subjective. A common example found from some of the articles that talk about it is a gym membership. In addition, altering your spending habit is quite difficult and often if your income is relatively low, it may not even be possible to save 20% of your income.

The design for this strategy would be to have an input element where the user can enter their income, however it would also help determine the income from previous transactions. The website would then breakdown their income into the 3 sections and give them values for what they can spend on each. Finally, for their previous thirty days of transactions (the usual time period between



each pay day), the user will be able to drag and drop each transaction into the three categories to help them see how much they are currently spending. If they are spending too much on any, they can make adjustments accordingly.

The other method that was considered for a budgeting strategy was to perform budget prediction. This would involve viewing how much they have spent so far for the current month, and performing some pattern recognition to make predictions for the remaining days. They could then compare this to a budgeted amount they planned to spend that month and identify if they are on track. This would be useful for people who are trying to save money for a specific goal, such as a holiday or a new car. The user would be able to see if they are on track to reach their goal, and if not, they can make adjustments to their spending habits to reach it.

The method of performing the prediction can vary from manual pattern identification to machine learning, so at this stage it was more relevant to identify which strategy would be most effective as performing the original aim of the project. Overall, the budget prediction method was more appropriate because every user could utilise the information it provided, furthermore, it also provided analytics which would be extremely difficult for the user to manually, unlike the 50:30:20 method. Also in predicting future expenses, the user could make adjustments to what they spend money on sooner, before it becomes a problem so is more effective.

### **3.3.4 Investments**

According to Finder.com, a reputable source for financial information, in Danny's article on investment statistics, he says that "Almost 1 in 5 Brits have owned stocks or shares" [3]. This is from a survey completed in the year 2023, so is recent, and is fact checked by other journalists. With over 13 million people in Britain alone, a strategy to adhere to this demographic would be useful and popular.

A particularly useful tool, which continues the idea of getting an overview of the user's finances, would be a central portfolio manager where logged in users can view and track their investments. This would be the only strategy that

doesn't utilise Plaid to access the data, so instead the user would have to add it themselves; however, this is not a problem as unlike the other strategies, once the asset had been purchased, they would only need to add it once. Following this, every time the user would come to check their investments, they could see the updated live price and view the amount of profit or loss they have made.

In order to perform this strategy, once the user has added their investments, they need to be stored. One option would be to store them in local storage within the browser. This would be the simplest to implement and as stored offline, would be somewhat secure, however it would have some disadvantages. To begin with, it would only be accessible from the same device and browser, so if the user wanted to access the tool from another device, they would have to add them again. In addition, if the user cleared their browser data, they would lose the investment entry so also have to manually add it again.

The other option would be to persist them using a database. As a PocketBase data store is already being used for the application, this wouldn't require too much extra work and complexity; furthermore, there are some major advantages. Unlike local storage, when using the database, the information is accessible and up to date from any device or browser, so the investment would only have to be added once. Also by linking the investments to only be accessed by the user who created them, they would be extremely secure and could be treated like the Plaid access\_tokens.

In the end, the database option was chosen because synergises well with the current design of the system and has the same advantages as local storage. As well persisting the investments, the application would also have to get the current price of the investment to determine if is profitable. To do this, the application would have to make a request to a service that provides the current price of the investment. Financial modelling prep is an easy to use service that provides quick and live stock prices so is ideal for this task. Given the price the investment was bought for, and the current price, the application would be able to calculate the profit and for example could be displayed as green for positive, and red for negative. This means the user can identify which investments are profitable and should continue, as well as which ones are not

and action should be taken.

## **3.4 Budget Prediction**

One decided that budget prediction was the method for the budgeting strategy, the actual prediction method needed to be designed. For this there was 3 different considerations: a basic manual pattern identification method; utilising linear regression to find the relationship; and using a neural network to find the trends. For each of the method, the only accessible real training data was the author's past 3 years of transactions, so this factor has to also be considered when deciding which method to use.

### **3.4.1 Manual Pattern Identification**

This method would be the simplest to implement and would involve manually identifying the patterns. Given the training data, set time periods would be suggested and then determining which one and by how much is the biggest of these. For example, given the three years transactions, split the data into the different seasons and identify in which season is the most spent. Using this knowledge, if a query is wanting a prediction in the most expensive season, then increase the normal prediction. This can be repeated on many different features, each of different time periods such as weeks, weekends, months, national holidays. Once many different factors have been analysed, the prediction can be made by combining all the factors together. This method would be very fast at making a prediction, but extremely complicated and inaccurate to initially implement. Furthermore, the predictions may not be very accurate as some features may not be factors, and others may not have been considered. Ultimately, this method would sacrifice accuracy for speed and complexity.

### **3.4.2 Linear Regression**

Linear regression is a method for "modelling the relationship between a scalar response and one or more explanatory variables" [6]. To apply it to budget

prediction, the output expenditure would be the scalar response, and the explanatory variables would include the budget set out by the user as well as the current date. The model would find the relationship based on the past 3 years of data, and then when a query comes in, it would quickly be able to use the learnt relationship to output an expected expenditure. The main advantage of this method is that it would not involve any manual pattern identification and the relationship would be computed. Despite this, it still wouldn't be that accurate as the relationship between expenditure and explanatory variables almost definitely will not be linear, which is the major downfall. One option to counter this would be to project the input data into a higher dimension. This would allow the model to find linear relationship in this projected space, but then a better non-linear relationship in the original space. This would be a good option, but would require a lot of computation to do so, may be considered over engineering for this problem, and would not even guarantee a better result.

### **3.4.3 Neural Network**

The final option that was considered was to use a neural network. This would take the input data, manually identify the features and trends and then, with lots of training data, learn the patterns. Within choosing to do a neural network, there is still so much variation as each model can be very different. The neurons themselves can have different activation functions, but also the connections between them. It can be feed-forward with no loops or be a recurrent neural network with some feedback. Each layer can have various number of neurons and the number of layers can also vary. Acknowledging this, if the network was to be designed effectively, it could out perform the other two methods in terms of accuracy so would be most ideal for this problem. The drawbacks include the fact that it would involve a lot of testing and tuning to get the best results, and the fact that the input data is limited doesn't help.

### **3.4.4 Conclusion**

In the end, out of the three options described above, the most applicable model for the problem of budget prediction was a neural network. This was because it didn't rely on manually finding the features to use, so would identify even the hidden and complex patterns, as well as, after being trained, generate quick predictions with high accuracy. The implementation of the budget prediction is described in much more detail in the implementation section 4, but a quick overall explanation is given below.

Deciding the architecture of the neural network would involve lots of trial and error to identify what is most appropriate. There is little research on the topic of predicting expenditure given past expenditure so not much even to help start. Through generalising the problem, the key idea which helped was to think of the problem as a time series problem. By modifying the input data from being a list of chronological transactions, to instead being the cumulative amount spent each day and resetting every month, the input data could be treated as a time series. This has much more research as a problem because time series prediction is a very important area for many industries, such as trading companies trying to predict the stock market. Applying lots of these ideas that have been applied to time series data found that using a recurrent neural network was the most effective.

### 3.5 Software Architecture

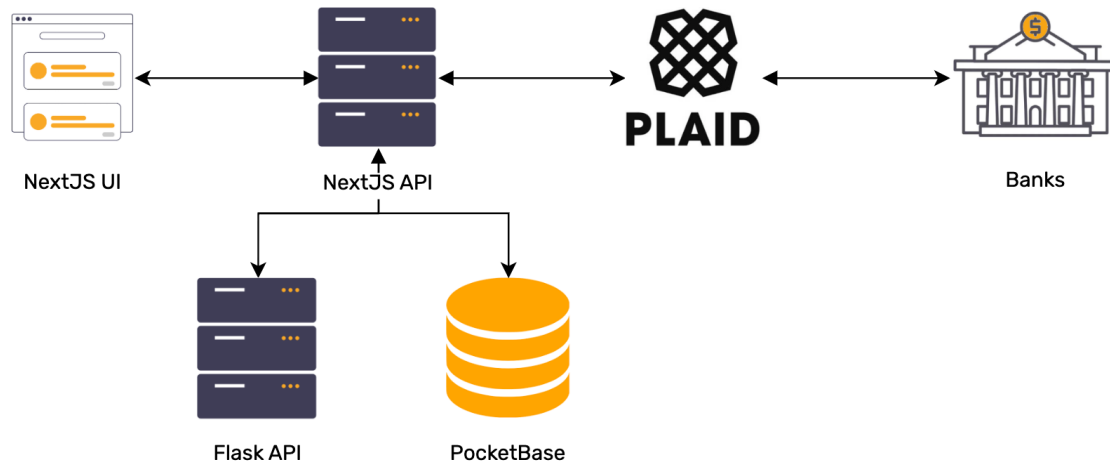


Figure 3.4: Software Architecture

The figure above (3.4) shows the software architecture of the project and the interactions between each service. The four on the left have been developed as part of this project, with the two on the right pre-existing entities. The user interacts with the front end, which is the Next.js UI. Having this separation from the backend was required for the Plaid authentication. The Next.js API routes are how the UI communicates with Plaid and the database. An example API route would be "get\_transactions", where the backend then returns the transactions received from Plaid, which itself returns the transactions from the bank itself. The access\_tokens and other long-term information such as user's login details are stored in the PocketBase database. This can only be accessed by the backend, and for example the access\_tokens can only be accessed by the user's who created in them in the first place for extra security. Finally, there is the Python Flask API which contains the budget prediction neural network. This API only has one route, which takes in the input data, propagates it through the neural network, and returns the output. It must be queried by the Next.js API routes as the input data requires the user's path month's transactions, therefore they must be requested from Plaid. This means when the Budget

prediction page is accessed, the frontend queries Next.js, which itself queries the Python Flask API, after retrieving the transactions from Plaid. The Python Flask API then returns the prediction to the Next.js API, which then returns it to the frontend to be displayed.

### **3.6 Endpoints**

Given the design for the strategies that the web application will exhibit, various Next.js API routes will be required. All of these routes will be prefixed with "/api" to be accessed by the client, and will be accessed using the HTTP POST method. This is because the requests often need to contain some contextual information for the server to decide what to do, for example, contain the user identifier to determine which user's transactions are being requested. The information passed between the client and server will be in JSON format as helps represent the data in a structured format as well as being easy to parse by both the client and server being written in JavaScript. In addition, python has a built in JSON library to work when communicating with the Python Flask API. Also each route will be written such that it can only perform its intended function or return an error. This is such that all the error checking can be done on the client side to improve security, as well as reduce the amount of code to be written and make the user experience better for graceful error handling.

### **3.7 Legal, Ethical and Professional Considerations**

## Chapter 4

# Implementation

### 4.1 Project Management

The table below is the initial project timetable set out in the project specification document. It was used as a rough guide to keep the project on track by helping know what tasks needed to be done and for how long.

Dates	Event
T1 W1-2	Project specification
T1 W1-2	Research into development tools
T1 W3-9	Development of website foundation
T1 W4	Design basic UI
T1 W5-7	User authentication
T1 W8-9	Integration with PlaidAPI
T2 W1	Testing of website foundation
T2 W2-10	Repeated strategy development
Easter	Finishing touches
Throughout	Document research
T3 W1	Dissertation due

Following the submission of the progress report, it was made clearer that the project had four key phases. The first phase was research into the development methodologies and design. The second phase was the development of



two proof-of-concepts and merging them into a single application, following the waterfall methodology. The third phase was repeated strategy implementation and testing, following an agile methodology. The final phase adding the budget prediction as a separate major feature. These divisions were a simple progression from the original timetable and were more natural with the whole project schedule by that point.

#### **4.1.1 Research and Design**

Most of the findings in this section have been outlined in the previous chapters (2, 3), but it is still worth talking about how the research was conducted and the structure of this process. This phase was performed prior to the progress report and change in timetable, so aligned with the original project specification.

Initially, only the aim of the project and the plan to use open banking was set in stone. In early term 1, research into how open banking is performed and how a developer could utilise the service was the main priority, this is when the discovery of Plaid was made, and the decision to use it happened. With familiarisation and proficiency in Next.js, it was a clear choice to use as the framework, and lots of the other design decision were made from understanding how to use Plaid.

Plaid is a relatively new technology, but it is still like any framework so to understand how to use it, a common research methodology was used. Viničius' article [1] talks about the best method for learning a new language or framework and has almost ten-thousand claps (the measure of popularity of Medium), so many people agree. The article is a good summary of the process and the main points which were followed when learning Plaid were: start with the basics, read lots of documentation, find examples on GitHub and Stack Overflow, and finally build something. In addition to this, due to Plaid having an YouTube channel [16], watching videos was also added and provided an effective way to learn.

Plaid had a getting started video, which helped explain the basics of the service as well as the terminology used in the documentation. Following this, lots of the documentation was read to not only help when coding, but also

to understand what features were available and how they could be used to build financial capability. Due to the recency of the revised way Plaid works, there are few example of how to use it on GitHub, and almost no questions on Stack Overflow; however, as part of the Plaid getting started video, they provide a code repository with a basic implementation of Plaid which helped to understand the service. Learning how to use Plaid helped inform lots of other design decisions, for example the Plaid authentication flow dictated the need for separate frontend and backend services to be developed, as well as the need for a database to store the user's credentials.

The other major development technologies that were researched here included Next.js features that had not been used before, such as server side rendering, as well as the combination with TailWindCSS and the use of PocketBase. The research into these technologies was done in a similar way to Plaid as it have proven to be effective. The main difference is that there is much more documentation and resources available for these so was slightly easier to learn. As well as technologies, concepts had to be learnt such as how authentication should be performed to ensure that user information is kept secure and how to follow modern web development best practices to build a successful web application.

The final point in Vinicius' article was to build something. This is the major reason which influenced the next phase of the project to be building two proof of concepts to test and improve the knowledge of the technologies and concepts learnt, but also provided a good basis for the actual application itself.

#### **4.1.2 Proof of Concepts**

To put the research from the previous phase into practise and work with the new technologies, this phase was to build two separate proof of concepts, one to have basic user authentication and the other to be able to interact with the Plaid API. Once both the prototypes were built, they would be merged into one application and form the basis for the rest of the project.

This phase of implementation is where there was first deviation from the original plan in the specification. In this instance, the development followed the

waterfall methodology. This was because it lent itself nicely in the way this phase was structured. Having the knowledge of what the technologies can do, and having a clear goal for each proof of concept, a clear step-by-step plan could be constructed. In addition, there was only a single developer so the waterfall methodology was a good way to ensure that the project was on track and that the developer was not overwhelmed with too many tasks at once. Waterfall is known for being effective in projects that have concrete objectives set out from the beginning. Both applications were worked on the same time, performing each stage of the waterfall process for both applications. This was done to ensure that the applications could also interact with each other when the time came to merge them. For example the second stage of waterfall is design, so both the designs for the applications were set out at the same time, and both before the implementation (the next stage).

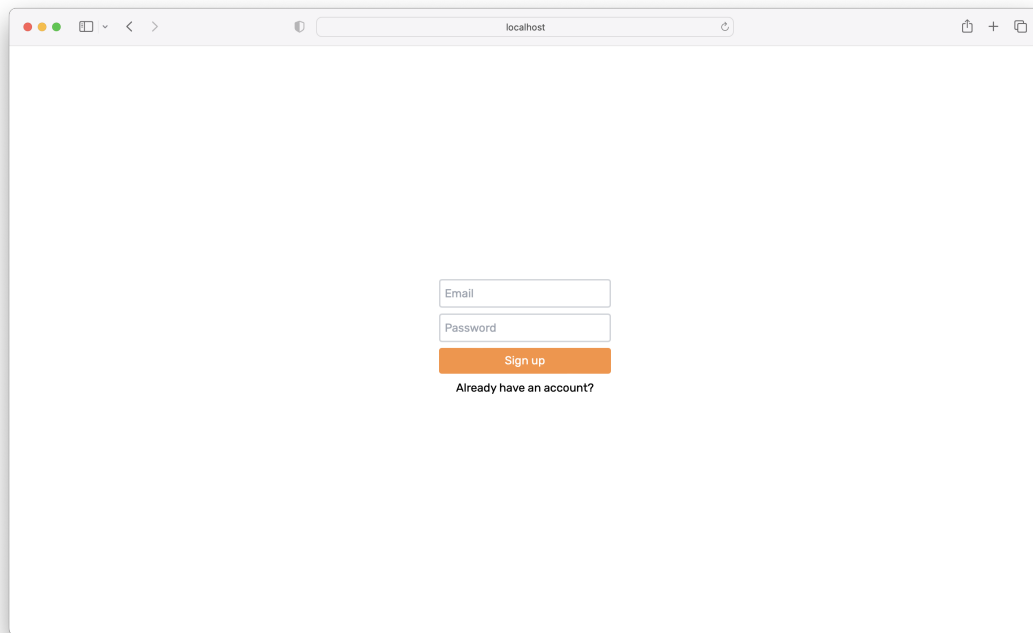
This sequential process is easy to manage but does not perform well for unexpected changes. This meant that the original requirements had to be detailed enough to encompass any possible queries, as well as have some contingency plans in place. For example, from some initial research, Plaid's endpoints could take in an array of accounts to return the transactions for all of them, however online elsewhere, someone had mentioned that this often just failed. To account for this, the plan was to use the Plaid endpoint, however if it didn't work, there also was a plan and design in place to manually query the endpoint for each account, which would functionally be the same.

## **User Authentication**

User authentication was needed to only allow logged in user's to access their dashboard, and only each logged in user can access the transactions for the access\_tokens which they created. The design for this application was to have three pages. The first page would be the signup page, where they can create an account. The second page would be the login page, where they can login to their already created account. The final page would be the dashboard which displayed the user's email if they were logged in, and would redirect to the login page if they were not. The signup page and login page would have links to each other so that the user could navigate between them.

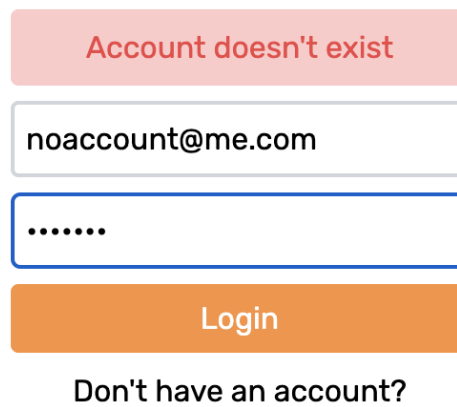
The implementation for the user authentication was simple yet secure. Both the signup page and login page have two inputs for the email and password, along with a submit button for the form. When a user creates an account, firstly the client checks that the email is a valid and unique email, and that a password has been entered. If so, the database is then populated with the new account and the user is sent to their dashboard. If the email is not valid or unique, the user is notified of this via an error message. The creation of user's account, as well as when performing the logging in is done using the PocketBase npm package, which automatically hashes and salts the passwords for security.

The login page works in a similar way, but instead of creating a new account, just checks that the entered email and password match an account in the database. If so, the user is sent to their dashboard, if not, they are notified of the error via a message on the page. On the dashboard page, the user is checked to be logged in via the PocketBase authStore, which uses JSON Web Tokens (JWT) under the hood. This means they can hold the user's ID and allows the database to limit access to the user's data by only allowing requests that come from that user. The ID is stored encrypted in the JWT, which itself is stored encrypted in a cookie. In addition, the JWT is only valid for a short period of time, to avoid cases where the cookie is stolen and the user's account can be accessed by someone else. Security is a major concern for this project as it handles sensitive financial data.



A screenshot of a web browser window displaying a minimal signup page. The browser's address bar shows 'localhost'. The page contains two input fields: 'Email' and 'Password'. Below these fields is an orange 'Sign up' button. Underneath the button is a link that reads 'Already have an account?'.

Figure 4.1: The minimal signup page for the user authentication



A screenshot of a login component. At the top, a red error message box says 'Account doesn't exist'. Below this is an email input field containing 'noaccount@me.com'. Underneath the email field is a password input field with masked characters '.....'. Below the password field is an orange 'Login' button. At the bottom, there is a link that reads 'Don't have an account?'.

Figure 4.2: The login component, with an example error message

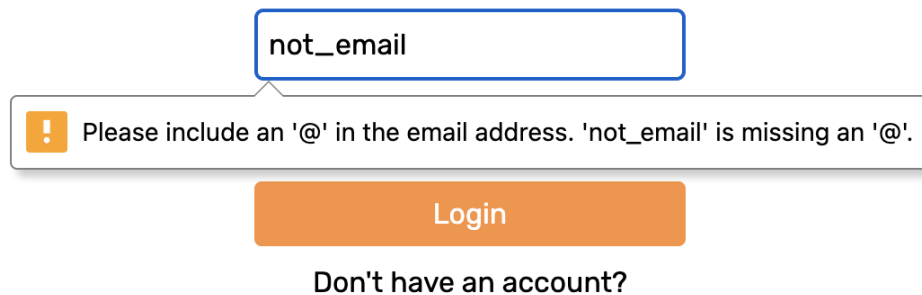


Figure 4.3: The client-side validation for the email input

## Plaid API

The second proof of concept was to be able to interact with the Plaid API. The design was to allow a user to follow the authentication flow of Plaid and link a bank account, this results in getting an `access_token`. Then, using the token, access their recent transactions. To begin the process, a user would click on a button to request the link token and have the link widget popup. The rest of the authentication flow would be as described in the background section (2), and after linking the `access_token` would just be stored client side for now.

This system required minimal design and little styling as the authentication flow is dictated by Plaid themselves, and the aim of this proof of concept was just to have functionally, so no need for fancy looking components. Again as using the waterfall methodology, the implementation was also simple. From the previous research, the whole process was already known and so from the complete plan created in the stage before, there was little deviation.

## Merging

At this stage, there were two independent proof of concepts that served different purposes, however the aim for the project was to have one working application that does what both of them do, and more. Both the proof of concepts had been written using git, but where on separate branches. The basic user authentication was deemed more fragile, so it was decided to merge

the Plaid interaction into that application. The idea was to have the user first perform the basic authentication, then on the dashboard they could link their bank accounts, and then for each access\_token generated, this was to be stored in the database. Furthermore, on the dashboard, if that user already had an access\_token available, they could request their transactions and they would just be displayed in the console.

With the waterfall methodology, and having the knowledge of how to implement these systems already independently, each stage could, and was, meticulously planned so the merge went smoothly and the bare-bones application was ready for the strategies to be added. The only new feature, not in either was the ability to store the access\_token in the database, and retrieve it when doing the transaction query; however, this was not that difficult from performing the user authentication such as getting the user's email/password, and it is a common task so there were lots of resources online to help.

During the implementation, something that was not considered was when a user has multiple access token and what to do. This oversight was easily mitigated, as the rest of the functionality was implemented in accordance with the original plan. Then as an extra a slight modification was made afterwards that when requesting the transactions, instead of getting the first access\_token, it instead returns all the access\_tokens, and then for each a query is made to Plaid. This is different to a user having multiple bank accounts linked, as each access\_token can contain multiple bank accounts, but the ability to toggle the individual bank accounts was not worried about at this stage.

### **4.1.3 Repeated Strategy Implementation**

At this stage, the application was extremely basic and only had the ability to link bank accounts and retrieve transactions. The actual strategies to help improve financial capability were yet to be added or even designed. This was a separate major phase in the project and instead of using the waterfall methodology, an agile methodology was more appropriate and so used. It wasn't any particular sub-methodology like scrum or lean, but instead just had very agile practises as was worked on in sprints. Each sprint was for a different

strategy and so they could be independently added, but would come together to form the final application. This separation was also shown during development as each strategy was developed on a separate branch, and then merged after passing the tests and being mostly complete.

## **Agile Development**

In general, the waterfall methodology should be prioritised for a project of this scale and especially where there is only one developer. For the repeated strategy development, due to the nature of the requirements being difficult to define prior to each strategy, agile was more appropriate. Within each strategy, and therefore sprint, research into the how best to include the strategy to maximise financial capability. This meant that the actual plan for each strategy was not known until the research was complete, so unable to set clear objectives in waterfall. Similarly, agile allowed for each sprint to consist of this research, but then followed by the design and implementation. If any issues arose during the implementation, they could be easily managed by altering the design, if appropriate, due to the flexible nature of agile.

Initially it was decided that each sprint would be three weeks long, one week each for research, development, and testing. However, it was found that this gave too much time for each section and especially for research so was modified accordingly. Agile's ability to change on the fly was a major benefit, in particular for this example. The spring was reduced down to roughly ten days, where there was so set time for each section, but instead the time was split among the three. This was useful after completing the transaction strategy, and moving on to the categories strategy. Much less time needed to be spent on the research as there was some crossover, meaning more time could be spent on the implementation and testing.

## **Credential Management**

Following the merged proof of concepts, the application was sent to Plaid to request development access (instead of sandbox). Thanks to the previous research, the authentication flow was strictly and securely followed such that



the application was approved. As outlined in the background section (2), development access meant that real live financial information could be accessed and used. This was a major milestone, but also came with further things to manage. With development access, the application could have one-hundred access\_tokens in total. The credentials were therefore treated as a resource and were used infrequently to ensure that enough were available for the final testing stages and the demonstration in the presentation. To account for this, all unit tests were only ever run in sandbox mode and only once these were passed did integration testing begin. Integration testing was firstly done in sandbox mode, but then once happy, the same tests were performed in development mode. More often than not, no issues arose only in development mode, but it was still important to test in both modes to ensure that the application is reliable.

### **GitHub Flow**

As mentioned earlier, each strategy had its own branch and only merged once it was complete. This was in attempt to follow the GitHub Flow branching model [7]. It involves having the main branch always deployable; all changes are made through separate feature branches via pull requests and merging; and rebases are used to avoid conflicts. The advantages include that multiple versions do not need to be managed, as well as that frequent changes can be pushed e.g. after every sprint, and the codebase is much cleaner to work with.

The original plan was to use the GitFlow workflow [5] where there are develop and feature branches, however this workflow is aimed at more collaborative projects, as well as ones that need to have stable releases at set time periods. Furthermore, GitHub Flow offered the same advantages for a single developer, and was much less complex.

### **MVP and Testing**

By the end of the repeated strategy implementation, a minimal viable product (MVP) was created that had the functionality to improve financial capability, but still had room for further features like budget prediction. Each individual

component was thoroughly tested, as well as the integrations and pages as a whole. The individual component testing was often done in code and included the frontend as well as the backend, whilst the integration testing was mainly done manually. There was some integration tests completed in code, and all the automatic tests had to be passed before any branch was merged.

#### **4.1.4 Budget Prediction**

Following the repeated strategy implementation, there was a basic budgeting strategy, but it only displayed the expenditure so far in the current month and allowed the user to set a budget. The original aim was to increase financial capability, and a worthwhile feature that would do this would be to predict the expenditure and allow the user to adjust earlier, ultimately to make better decisions.

This was added as a separate major feature, unlike the previous strategies, because it was much bigger. A lot more research and planning was required for this as it would involve creating machine learning models, as well as, a separate server to host the model. By this time, only minor tweaks were being made to the MVP as hot-fixes, so the GitHub Flow workflow was followed again, by performing this in a separate feature branch.

The process for this feature was somewhat treated like another agile sprint, however it was not formalised like this, as going into the prediction, there was little certainty as to how effective it would be, or even what the requirements were outside of predicting future expenses.

Also as part of managing this section, the machine learning models would need real financial data for training to be the most accurate. This has some privacy concerns as the model would learn from the this data the information would be shown in the predictions. Furthermore, there was a limitation on the amount of financial data that could be accessed, so in the end it was decided to mix the data. The model was trained using only the developer's financial data as well as some of the data from Plaid's sandbox mode. This was in an attempt to still create a useful model, but to not leak any information and it not overfit on a single individual. In addition, some data manipulation techniques,

outlined in the implementation, were used to simulate more data and make the model more robust.

## **4.2 Strategies**

This section aims to outline the strategies that were implemented in the application. Only the final option that was decided in the design section will be explained. They were each implemented as independent sprints, however they all had the same goal of improving financial capability by providing the user with information and tools to make better financial decisions.

### **4.2.1 Active Bank Accounts**

As part of the navigation bar show in figure 3.2, the rightmost button with the label 'accounts' is a dropdown. Clicking it displays a menu containing all the linked bank accounts, an add accounts button, and a logout button. Each linked bank account is a bar containing toggle button, the name and type of the account, and a logo representing which bank chain it is from. The add accounts button is where the user can link more bank accounts by beginning the Plaid authentication flow. The logout button is simply where the user can logout and the application will clear the cookie.

This was implemented as part of the transactions sprint as was not included in the initial proof of concepts, however is necessary for all the other strategies. All the bank accounts are active by default, however the user can toggle them off to not include them in the analytics. Each bank account has an associated id, and so a list of active bank account ids is stored as state in the navigation bar. This array is passed to each of the strategy components as a prop to filter the data and only include the active accounts.

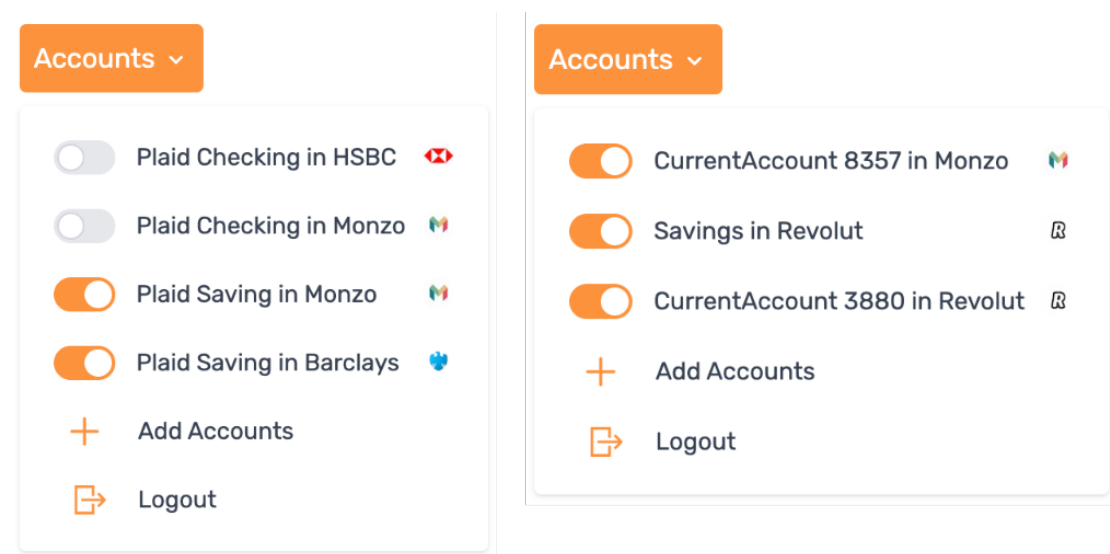


Figure 4.4: Account dropdown for (left) sandbox mode and (right) development mode

In the above figure 4.4, the left image is from the sandbox mode and the right image is from the development mode. In sandbox mode, only the savings accounts have been toggled on, for example if a user wanted to only view the analytics for those accounts. In development all the bank accounts, which in this case is a Monzo and two Revolut accounts, are enabled. When switching between a toggled account, the icon has a slight animation.

### 4.2.2 Transactions

This was the first strategy implemented, so there was a lot of experimentation and learning how best to get the data from Plaid and propagate it to the frontend, then display nicely. Each strategy was implemented as a separate component and the data required, such as the user object and active accounts, were passed down as props in the React model. This meant that the dashboard would simply just render the component of the strategy currently being viewed.

The transactions strategy is implemented as a table where each row is a separate transaction to a merchant. The transactions are grouped by date and in reverse chronological order. When querying Plaid, it returns the transactions in order of the accounts, so some extra data preparation was needed.

A single Next.js API route was needed to be built. It would take only the user ID as a parameter, and query the database for all the access tokens associated with that user. It is only able to see the access tokens that it created, so in this case would just get all the access tokens. With these, it makes a request to Plaid to get the transactions for each access token. The transactions are then grouped and sorted in the API route, such that the frontend needs to do minimal work. The transactions are then returned to the frontend as a JSON object to be displayed.

Plaid has multiple endpoints that access transactional data. To begin with, `"/transactions/sync"` was used which gets the transactions. The first call returns all the historical transactions (up to a limit) along with a cursor, subsequent calls update the cursor and only return transactions after the cursor. This reduces the amount of data flow and means the transactions can be stored locally for faster loading. On strategy load however, the UI needed to send a request to Plaid, and a request to the database

This ended up being changed to instead use `"/transactions/get"` which returns the transactions between an input start date and end date as the transactions didn't need to be stored. Plaid encourages to use the sync endpoint as it acts as a subscriber model and also paginates the data, however the get endpoint enabled code reuse for later strategies, withdrew the dependency on the database, and also reduced the number of requests when loading the strategy.

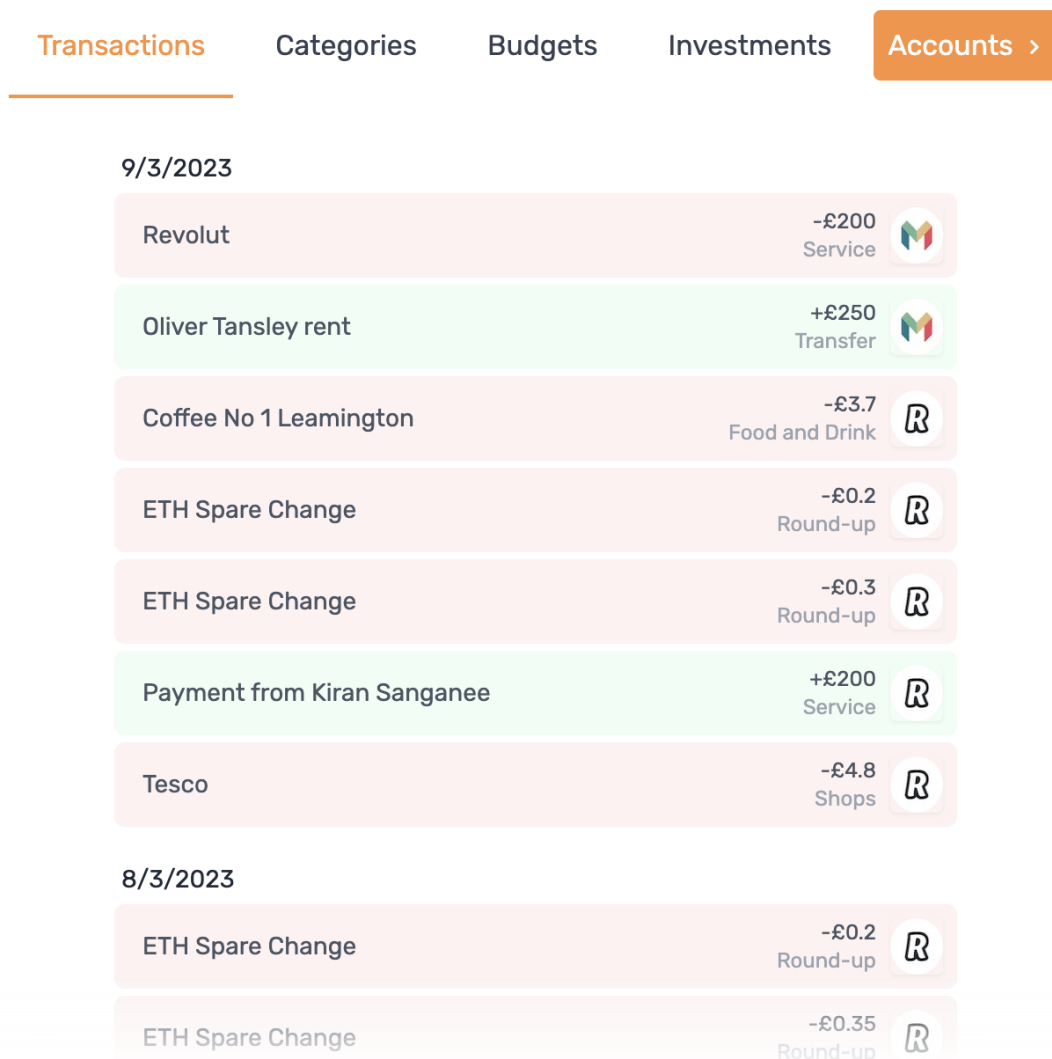


Figure 4.5: Transactions strategy in development mode

The above figure (4.5) is an example of what the transactions look like in development mode with two separate bank accounts enabled. Each transaction has its own bar and displays information that is relevant to help provide an

overview. Income is coloured light green, expenditure is coloured light red; yet when whichever transaction is hovered over is coloured slightly darker for UI feedback. Often enough transactions are shown that scrolling is required, however there was not a consistent way to style the scroll bar as it is dependent on the browser. To account for this, instead the scroll bar was removed entirely, but the bottom-most viewable transactions has an opacity gradient to indicate there are more transactions below.

### **4.2.3 Categories**

This sprint followed on easily from the transaction sprint as much of the backend functionality could be reused. For this strategy, only the expenditure needed to be analysed. One option was to modify the Next.js API route so that it separates the data into income and expenditure (by positive and negative value) automatically, however would have involved some slight changes to the transaction strategy. Instead, a similar but new Next.js API route was created to get the expenditure data only. This also acted as an optimisation as the categories strategy only needed the previous thirty days of transactions so speed up the data processing by creating a different API route.

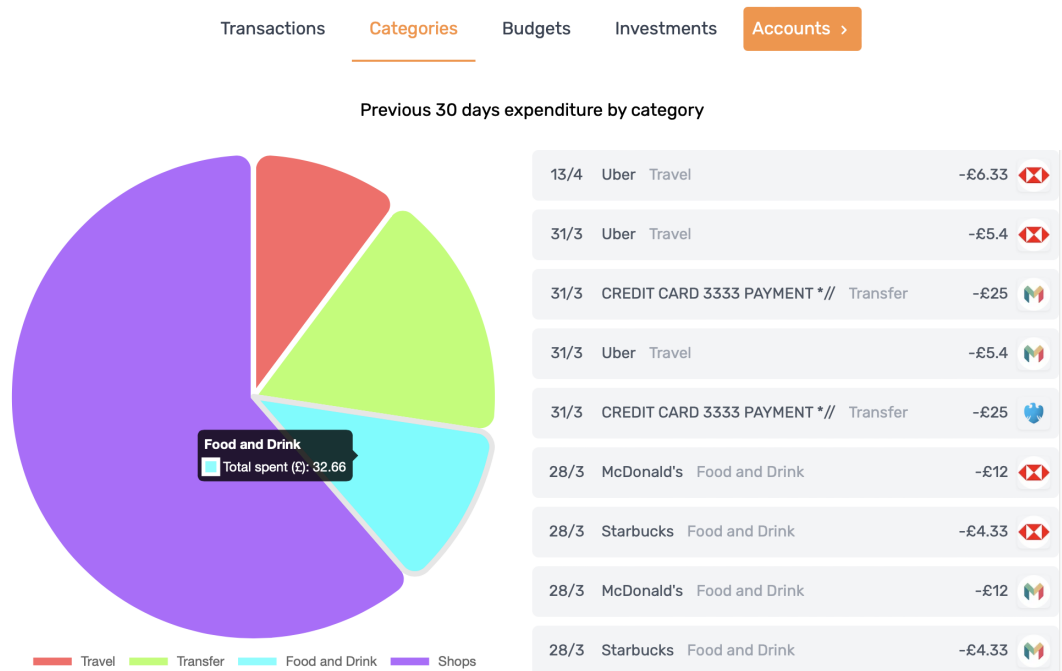


Figure 4.6: Categories strategy in sandbox mode

The above figure (4.6) is an example of what the categories look like in sandbox mode with four accounts enabled. On the left is a dynamic pie chart that shows the relative amounts each category makes up for the past thirty days. The colours are intelligently chosen; in the above example, there are a total of four categories, so four different colours are chosen which are most distinguishable from each other. In the figure, the user is hovering over the light blue section which is 'Food and Drink', so the total for this section is displayed. On the right is the list of all expenditure for the past thirty days but these are not grouped at all, and there is an emphasis on the category; unlike in the transactions strategy.

The below figure (4.7) is an example of what the categories look like in development mode with two accounts enabled. In this case the pie chart is more realistic as is made up of real data. In addition, a filter has been applied to the categories to only show expenses in the 'Food and Drink' category. To apply



this filter, the user has simply clicked on the 'Food and Drink' category in the pie chart; to remove the filter, they can just click the red trash icon above the expenses.

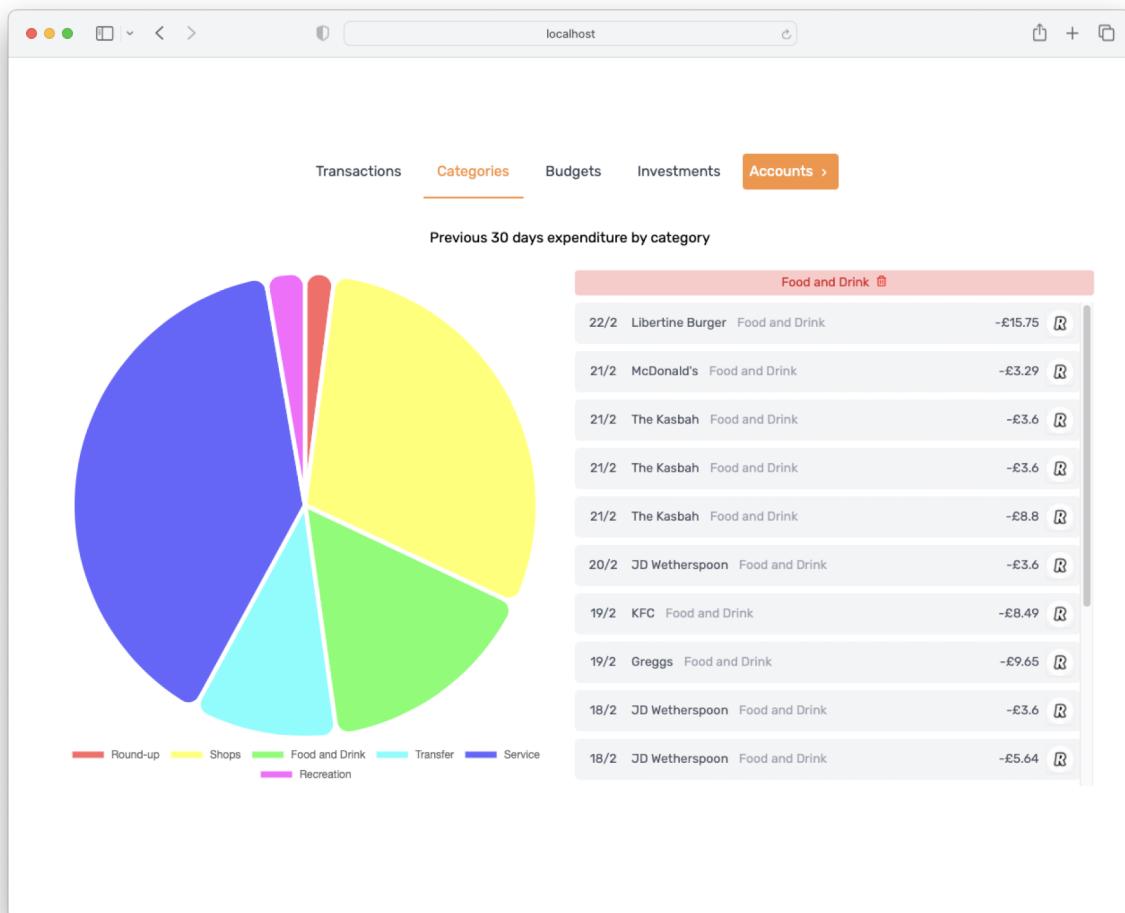


Figure 4.7: Categories strategy in development mode with a filter

#### 4.2.4 Budgets

In the original sprint for the budgeting strategy, there was no budget prediction. Instead the sprint resulted in a graphical view of the previous expendit-

ure by displaying the amount spent each day, with a total amount on the side. The user could also enter an amount to budget for the month, and on the line chart, this budget would be displayed. The research portion of this sprint did identify that budget prediction was a useful feature, but it was not considered initially as would involve more time than the single sprint. After the end of the investment sprint, the budget strategy seemed lacking so a new phase was proposed to incorporate budget prediction.

### **Time Series Forecasting**

As mentioned in design section (3), the best option for budget prediction was to use a neural network. The decision that was most key in turning the problem of budget prediction, into a more well defined problem was to turn the expenditure data into cumulative amounts spent. This altered the data into a time series and meant that there was much more research and literature available on how to tackle this kind of problem. The cumulative amounts would also reset every month to match the budgeting period, but also made no difference to the model as the model would learn this.

The book *Forecasting: Principles and Practise* (3rd edition) [10] was a particularly useful resource for this section. Not only did it talk about the effects of seasonal time series - in this case is the cumulative resets, but also about how to apply linear regression techniques (which helped in the deciding to use a neural network) and using a neural network to make predictions.

### **Long Short-Term Memory**

The term that often repeatedly cropped up during the research was LSTM, meaning Long short-term memory. They are "a type of recurrent neural network capable of learning order dependence sequence prediction problems" [2]. The cumulative data could be seen as a order dependent sequence, and the problem is just predicted the next value to appear in the sequence.

What makes LSTM's different from ordinary recurrent neural networks is that the individual units have single value memory cells that can be controlled by

three gates. The first gate is an input gate which determines if the current value in the cell should be summed into memory. The second gate is the output gate which determines if the memory is added to the input as output. The final gate is the forget gate which determines if the memory should be cleared. The weights which dictate if these gates are open or closed are learnt along with the other weights in the network.

## **Training**

To train the LSTM model, the input training data consisting of the past three years of transactions first needed to be modified. The data was turned into the cumulative data and resetting at 0, but in addition it was split in such a way to maximise the amount that could be learnt. This was done by taking the first value up to the thirtieth value as one training example, and the thirty-first value being the predicted output. Then by taking the second value up to the thirty-first value as the next training example, and the thirty-second value being the predicted output. This was done for up to the end of the data resulting in 1064 training examples. Techniques such as test-train split and validation sets were also used.

## **Architecture**

Throughout the research and testing phase, LSTM models were found to be the most accurate relative to other neural network architectures as well as other machine learning models like linear regression. This also complimented the fact that there is support for LSTM models in the Keras library. The structure of the network that was found to be the sufficiently simple that it could be trained in a realistic amount of time, but also sufficiently complex to be able to learn the data was a single LSTM layer with 64 units, followed by a dense layer with 1 unit as output. To make the predictions realistic, but not too overfit on the training data, 30 epochs were used to train the model. Finally, a window size of 30 was used, meaning the model has 30 inputs, as to match the modified training data and budgeting period.

## Results

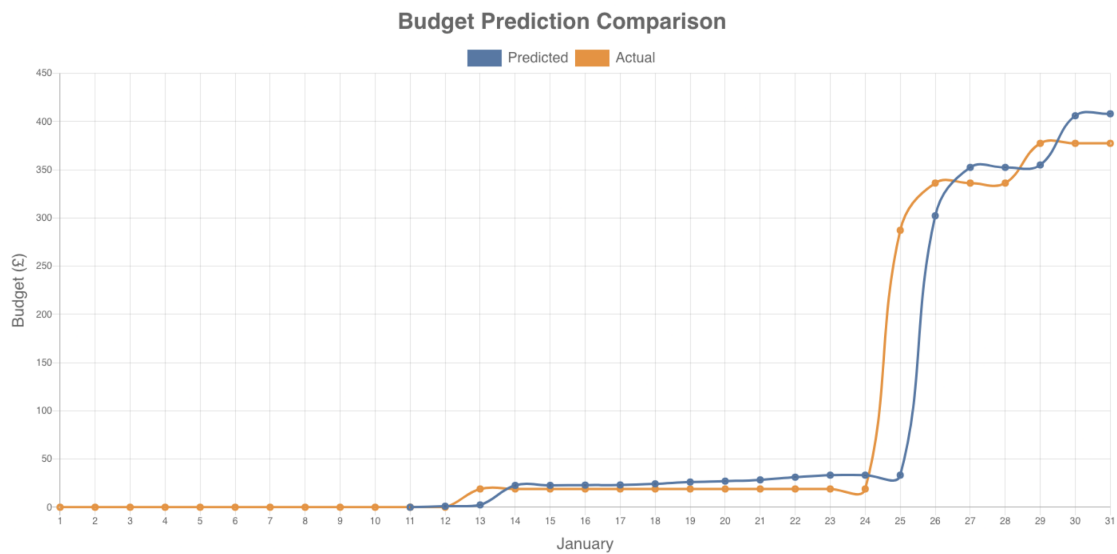


Figure 4.8: Budget prediction results

The above figure (4.8) displays the difference between the predicted expenditure vs the actual expenditure. In this instance, the data used for this graph is the fake data generated by Plaid, so isn't even related to the data that the model was trained on. The model was able to predict the expenditure very accurately, and also followed the pattern effectively. There is a slight offset in the two lines, but this eventually was found out to be due to the way Plaid generate their transactions. They assume that each month has thirty days in it, but the graph is for the month of January which has thirty-one days, explaining the offset.

## Budget Strategy

Using this accurate budget prediction model, it is incorporated into the budget strategy by making predictions for the rest of the month. Suppose the user is accessing the tool on the 15th day of the month. For the past fifteen days, the tool will have access to their actual transactions and can plot the cumulative

expenditure. For the remaining days in the month, a prediction is made and plotted.

The prediction is made by taking the past 30 days expenditure and predicting the amount on the 16th day of that month. Then using this predicted value and the true past 29 days of expenditure, an amount is predicted for the 17th, and so on. All of these values are then displayed to the user in the graph which they can compare to their budgeted amount.

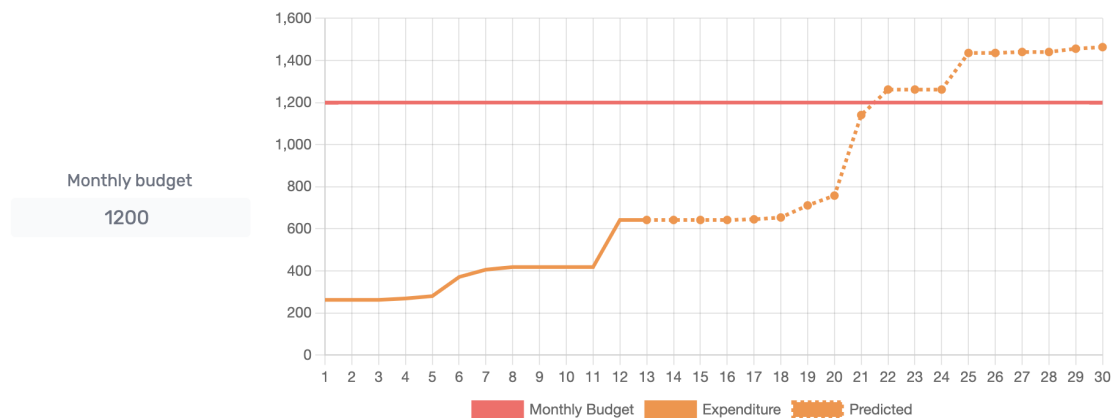


Figure 4.9: Budget strategy in development mode

The figure above (4.9) shows the budget strategy in development mode with the budget prediction incorporated. The date of access is the 13th April, so there is actual expenditure up to this date. The remaining days contain the predicted expenditure. The set monthly budget is £1200, but according to the graph, it is predicted that they will exceed their budget so should make adjustments accordingly.

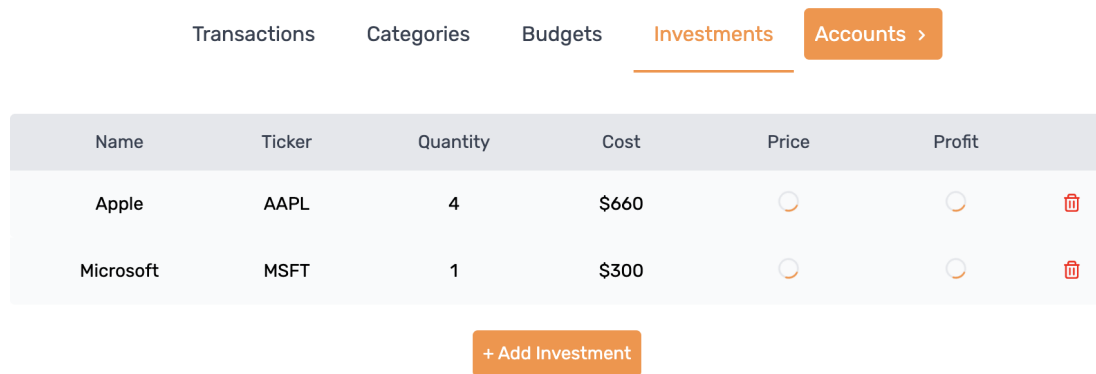
#### 4.2.5 Investments

The final sprint before the last phase was dedicated to implementing the investments strategy. As outlined in the design phase (3), the aim for this section was to give an overview of the user's portfolio. Unlike the other strategies, this one didn't use Plaid, so a needed a lot more custom functionality.

The user is expected to add their investment initially, however the tool should automatically update following this. This meant storing the investment in the database. To accommodate for this, a new Next.js endpoint was required which would take the relevant information about the asset, such as ticker and the price it was bought for, and stores it in the database. Like the access tokens, this entry is linked to the user's account and is only accessible by the creator of the entry. Similarly, another endpoint was required to retrieve all the investments for a user and one to delete an investment.

When a user accesses the investment strategy, the UI should display all of their added investments. This is a request made to the new endpoint to get the investments. For each investment, once the stored data is retrieved, it will then access a live API to get the actual current price for the asset. This is used to calculate the profit and percentage change. For an investment that has made a profit, it is shown in a light-green like income in the transactions strategy. An investment that has made a loss is shown in a light-red like expenses also in the transactions strategy. Each investment also has a delete button which calls the delete endpoint and updates the UI accordingly.

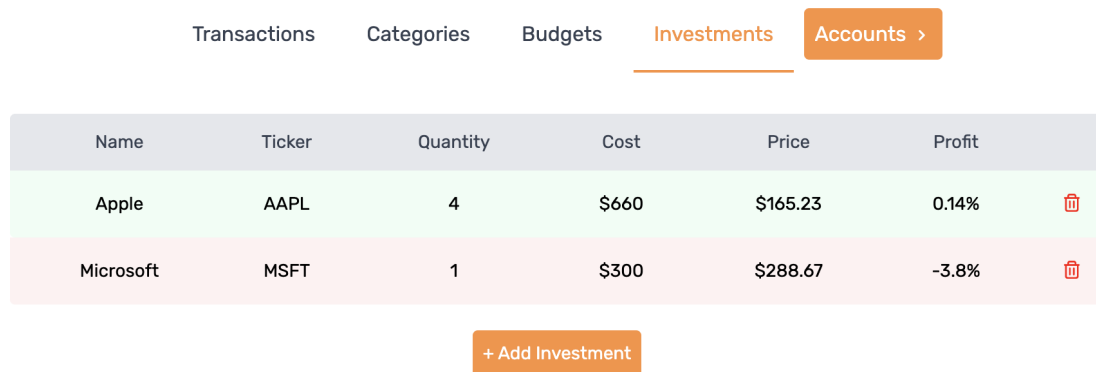
When accessing the Financial Modelling Prep service, the information retrieved from the database can still be displayed. This means that initially a loading icon is displayed for the data that takes longer to access. The figure below (4.10) shows the table with demonstration information retrieved from the database, but still loading the actual price. The figure below this (4.11) shows the table once the investments have been loaded and profits calculated. Each investment is treated independently, this means that they can load independently and so if one investment takes longer to load, the others will still be displayed as soon as they are ready.



Name	Ticker	Quantity	Cost	Price	Profit	
Apple	AAPL	4	\$660			
Microsoft	MSFT	1	\$300			

+ Add Investment

Figure 4.10: Investments loading



Name	Ticker	Quantity	Cost	Price	Profit	
Apple	AAPL	4	\$660	\$165.23	0.14%	
Microsoft	MSFT	1	\$300	\$288.67	-3.8%	

+ Add Investment

Figure 4.11: Investments loaded

The table of investments has a max height, this means that if the user has a lot of investments, they do not extend the page downwards to display them all. Instead the table is made scrollable. This is to allow a portion of the UI, below the table, to be for how the user adds an investment. The ticker is a dropdown and allows the user to select a supported ticker (effectively an investment identifier). The user can enter the quantity and the amount they paid for the investment. Once a ticker is selected, when entering, the price is automatically filled in with the current price as an aid. These inputs all have client-side validation to ensure the user enters acceptable data. If it does

not pass the validation, the box will turn red as a visual indicator of what is wrong, so can be corrected. Once the user has entered this information, they can click the add investment button which adds it to the table above as well as the database.

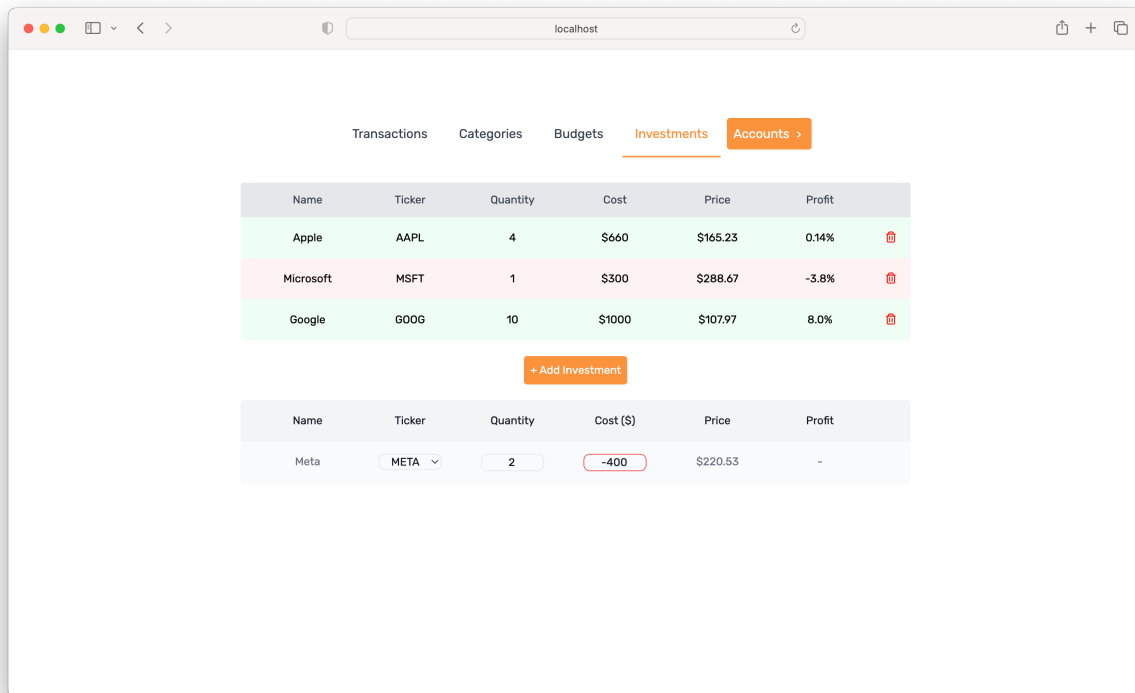


Figure 4.12: Adding an investment

The figure above (4.12) shows the user attempting to add a Meta investment. Everything is valid, except for the cost which has been mis-typed with a negative sign. This is picked up by the validation and the box turns red. The user can then correct this, then click the button and the investment will be added to the table.

## 4.3 Endpoints



URL	Input	Output	Functionality
add_investment	userID, ticker, quantity, cost	recordID	Adds the investment to the database and returns the record ID
create_link_token	userID	plaid_response	Requests a link token from Plaid and returns the response
get_accounts	userID	[accounts]	Returns a list of accounts associated with that user
get_balances*	userID	[balances]	Returns the balances for all accounts
get_date_grouped_transactions	userID, startDate, endDate, activeAccounts	[transactions]	Returns all the transactions for accounts in the activeAccounts between the startDate and endDate, grouped by date
get_investment_price	ticker	price	Returns the current price for the asset of that ticker
get_investments	userID	[investments]	Returns all the investments that user has
get_prediction	userID	[expenditure], [predictions]	Returns the expenditure so far, and the predictions for the rest of the month
get_split_transactions	userID, startDate, endDate, activeAccounts	[incomings], [outgoings]	Returns all the transactions for accounts in the activeAccounts between the startDate and endDate, split into income and expenditure
remove_investment	investmendID	{}	Removes the investment from the database
send_public_token	publicToken	{}	Exchanges the publicToken for an access token and stores it in the database

Figure 4.13: Endpoints

## Chapter 5

### Evaluation

Describe the approaches you have used to evaluate that the solution you have designed in Chapter 3 and executed in Chapter 4 actually solves the problem identified in Chapter 1.

While you can discuss unit testing etc. you have carried here a little bit, that is the minimum. You should present data here and discuss that. This might include *e.g.* performance data you have obtained from benchmarks, survey results, or application telemetry / analytics. Tables and graphs displaying this data are good.

## Chapter 6

# Conclusions

The project is a success. Summarise what you have done and accomplished.

### 6.1 Future work

Suggest what projects might follow up on this. The suggestions here should **not** be small improvements to what you have done, but more substantial work that can now be done thanks to the work you have done or research questions that have resulted from your work.

## Bibliography

- [1] Brasil, Vinicius. How to learn a new programming language or framework. *Medium*, February 2018. URL <https://medium.com/hackernoon/how-to-learn-a-new-programming-language-faster-dc31ec8367cb>. Accessed: 8/04/2022.
- [2] Brownlee, Jason. A gentle introduction to long short-term memory networks by the experts. *Machine Learning Mastery*, July 2021. URL <https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/>. Accessed: 13/04/2022.
- [3] Butler, Danny. Investment statistics: What percentage of the uk population invests in the stock market? *Finder.com*, March 2023. URL <https://www.finder.com/uk/investment-statistics>. Accessed: 7/04/2022.
- [4] Clark, Robin. Gnucash, 1998. URL <https://www.gnucash.org/>. Accessed: 29/03/2022.
- [5] Driessen, Vincent. A successful git branching model, Jan 2010. URL <https://nvie.com/posts/a-successful-git-branching-model/>.
- [6] Freedman, David Amiel. Linear regression. *Wikipedia*, March 2023. URL [https://en.wikipedia.org/wiki/Linear\\_regression](https://en.wikipedia.org/wiki/Linear_regression). Accessed: 5/04/2022.
- [7] GitHub, . Github flow. June 2021. URL <https://githubflow.github.io/>. Accessed: 9/04/2022.
- [8] GNUCash, . Gnucash register page, April 2013. URL <https://www.facebook.com/GnuCash/>. Accessed: 30/03/2022.

- [9] Hourston, Peter. Cost of living crisis. *Institute for Government*, Feb 2023. URL <https://www.instituteforgovernment.org.uk/explainer/cost-living-crisis>.
- [10] Hyndman, R.J. & Athanasopoulos, G. *Forecasting: principles and practice*. OTexts, 2014. ISBN 9780987507105. URL <https://books.google.co.uk/books?id=gDuRBAAAQBAJ>. Accessed: 13/04/2022.
- [11] Intuit, . Quicken, 2022. URL <https://www.quicken.com/>. Accessed: 30/03/2022.
- [12] Kwartalny, Nazar. Why use python for machine learning? *inoxoft*, April 2022. URL <https://inoxoft.com/blog/why-use-python-for-machine-learning/>. Accessed: 3/04/2022.
- [13] n26, . The 50/30/20 rule: how to budget your money more efficiently. *N26 Blog*, Aug 2022. URL <https://n26.com/en-eu/blog/50-30-20-rule>. Accessed: 22/11/2022.
- [14] NFEC, . What is financial literacy?, 2022. URL <https://www.financialeducatorsCouncil.org/financial-capability-definition/>. Accessed: 29/03/2022.
- [15] OBIE, . What is open banking. *OpenBanking.org*, Sept 2020. URL <https://www.openbanking.org.uk/what-is-open-banking/>. Accessed: 29/03/2022.
- [16] Plaid, . Plaidinc, April 2020. URL <https://www.youtube.com/@PlaidInc>. Accessed: 8/04/2022.
- [17] Plaid, . Plaid api documentation, 2022. URL <https://plaid.com/docs/>. Accessed: 30/03/2023.
- [18] Plaid, . Plaid institutions, 2023. URL <https://plaid.com/institutions/>. Accessed: 30/03/2022.
- [19] PocketBase, . Pocketbase documentation, 2022. URL <https://pocketbase.io/docs>. Accessed: 2/04/2022.

- [20] Premchand, Anshu & Choudhry, Anurag. Open banking & apis for transformation in banking. In *2018 International Conference on Communication, Computing and Internet of Things (IC3IoT)*, pages 25–29, 2018. doi: 10.1109/IC3IoT.2018.8668107.
- [21] Starling, Team. The 50/30/20 rule: Is it realistic in a cost of living crisis? *Starling Bank*, Sept 2022. URL <https://www.starlingbank.com/blog/50-30-20-budgeting-rule-is-it-realistic-in-a-cost-of-living-crisis/>. Accessed: 22/11/2022.
- [22] Storonsky, Nikolay & Yatsenko, Vlad. Revolut, July 2015. URL <https://www.revolut.com/>. Accessed: 30/03/2022.
- [23] (Theo), T3 Open Source. T3 stack. *GitHub*, 2022. URL <https://create.t3.gg/>. Accessed: 2/04/2022.
- [24] to Linux, Switched. Introduction to gnucash - free accounting software, May 2019. URL [https://www.youtube.com/watch?v=wBPg\\_AKdlG0](https://www.youtube.com/watch?v=wBPg_AKdlG0). Accessed: 30/03/2022.
- [25] Vercel, . Next.js, 2016. URL <https://nextjs.org/>. Accessed: 2/04/2022.
- [26] Vercel, . Next.js github, 2016. URL <https://github.com/vercel/next.js/>. Accessed: 2/04/2022.