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: 3rd

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

Chapter 1

Introduction

The term 'personal finance tool' encompasses various software, such as budgeting tools, investment management software and credit score calculators. This project aims to investigate strategies that help build a user's financial capability and confidence, followed by implementing a web application to support these strategies. In particular, the research will focus on strategies to help individuals to manage their, often several, bank accounts in one place.

1.1 Financial Capability

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 fulfils an individual's personal, family and global community goals" [16]. Consequently, the application aims to give users the tools to view their financial circumstances as part of the knowledge, and with it comes the confidence to make informed, beneficial decisions.

Financial capability and financial literacy will be used interchangeably for the remainder of this document.

1.2   Motivation

Part of the motivation for this project comes from the recent open banking technology movement [17]. This movement is where thousands of major banks have opened a set of endpoints allowing third-party applications to access their customers' financial data with explicit consent securely. It allows developers to build personalised financial applications and services tailored to their users, which is most appropriate for this project. Further details and other motivations are discussed in the background chapter below (2.4).

Chapter 2

Background

This section provides background information on the problem area by introducing and explaining concepts used throughout the rest of the document.

2.1 Current services

Understanding the currently available applications and their limitations is essential. In addition, it helps to narrow down the problem area and identifies the requirements for the new system.

A wide range of personal finance applications are available; however, they are not all suitable for the same use case. Therefore, this project focuses on those that help provide users with an overview of their finances, as these are most applicable in improving their financial literacy.

The three main categories of these existing tools are:

1. Applications that do not utilise open banking and so require manually importing the data

2. Mobile banking applications that sometimes use open banking but do not give an extensive overview

3. Applications that lack analytics and are expensive or filled with advertisements

2.1.1 Manual Importing Applications

The first category of applications is those that do not use open banking. Often, these tools are old and lack updates, so they have not been improved to utilise the endpoints offered in open banking. The most prominent example of this is GNU Cash [4].

GNU Cash is a free and open-source desktop application that allows users to track their finances. However, the downsides include that it is not very user-friendly and requires considerable experience to be used effectively. Switch to Linux mentions that it is a "great FOSS tool [...], but it can be complicated to set up" [27], 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 by looking outdated and complex (see figure 2.1 below).

Figure 2.1: Example GNU Cash UI [8]

The main weakness of software like this is requiring users to import their data manually. Most users do not have all their accounts and transactions readily available in a structured format, and this is even worse when the user would want it to update live with their recent transactions. Few aspects of these applications help improve financial literacy, so few will be used in the final web application.

2.1.2 Mobile Banking Applications

The next category is mobile banking applications. These 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 performs the same functions. Examples include the major banks such as HSBC, NatWest, Lloyds, and online-only banks such as Monzo and Revolut. Often, these applications do not use open banking; instead, they work with the accounts from that specific company. Some do connect with other banks; however, they often do not incorporate these accounts into the analytics and instead, just display the balances.

Taking Revolut [25] 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 Revolut's app functionality, users must open a bank account with the service and prove their identity. Although this gives the user confidence that their information is secure and personalised, it is also more challenging to use, slower to access the information, and overall is a catch. It also means that the app's primary purpose is not to provide a user with an overview of their finances but to provide a banking service, meaning it is not entirely focused on improving financial capability.

Some features of Revolut which are worth acknowledging, such as the UI and ability to track expenditures. A slick and intuitive user interface enables users to understand all aspects of their finances and helps build confidence. In addition, expenditure tracking is beneficial, as it allows users to customise a budget for a set period and shows what and when they have spent money during this period; overall, aiming to help them stay within the budget.

Revolut is a service that utilises open banking as it allows users to connect the app to their other bank accounts; however, it does not incorporate these into the budgets and only includes them in the net worth section. Revolut is an excellent example of a service that does use open banking but does not utilise it to its full potential.

2.1.3 Paid Applications

The final category of applications is just the 'other' section; however, most of these 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. An excellent example is Quicken which won the awards for the best budgeting app in 2020 and 2021 [12], but costs up to £10/month.

Quicken 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 quick-toggling is a feature which would be particularly useful in giving better analytics, as a user would be able to segment all their savings accounts.

Despite having some useful budgeting features, Quicken cannot perform budgeting prediction from patterns in expenditure. This feature also would help improve 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 helpful in the web application. However 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] enable third-party developers to build applications and services around the financial institution" in this paper [22]. Therefore, the open banking movement is the recent pressure on banks to open their data to third-party developers. They do this by creating a set of endpoints,

which developers design websites to query, and will respond with accurate and live data.

Each bank has its own set of endpoints, such that the authentication process and available information differ. Reading the documentation for each bank is time-consuming and not straightforward, yet learning how to interact with each bank's API is necessary to build an application that works with them all. This lack of standardisation is where Plaid [19] comes in.

2.3 Plaid

Plaid is a platform that effectively wraps all the endpoints provided by the individual banks and provides a single standardised set of REST API endpoints as URLs. Using Plaid allows a third-party application only to query Plaid and, in turn, can support all the banks that Plaid provides access to - over twelve-thousand institutions [20].

2.3.1 Plaid Tokens

A strict authentication flow must be followed to use Plaid's endpoints because it handles private data. Plaid has three different types of tokens as part of this flow. The first is the link token.

Link is the widget's name and pop-up used to authenticate the user with their bank. To start this process requires a link token. These can be requested from Plaid's API and are valid for 4 hours, but are not tied to any user and do not treated like 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. They are unique to that user and are not treated like passwords, as they are valid for only 30 minutes and cannot be used to access a user's private information directly. In addition, they must be exchanged for an access token, the third type of token.

Exchanging the public token for an access token is done via a Plaid endpoint. However, because the access token must be treated like a password and be kept exceptionally securely, it is done via the web application's server rather than the client. If it is done on the client, it risks being exposed, and anyone with this token can access that user's information. Finally, a set of bank accounts can be embedded within the access tokens, and a user may have several access tokens associated with them.

2.3.2 Plaid Authentication Flow

The authentication flow used when interacting with Plaid is a strict process that was designed by the developer. It is described here rather than in the design section because it forms the foundation of the web application, and it heavily influenced the rest of the application's design. It is as follows:

- The client requests a link token on behalf of the user.

- The response link token is then passed to the link widget

- The user sings into their bank on the link widget pop-up

- The link widget returns a public token

- The client passes this public token to the backend.

- The backend exchanges the public token for an access token

- The access token is stored in the database.

Then suppose the user wishes to access their transactions:

- The client queries the server acting as a proxy.

- The server identifies which user is asking for the information

- That user's access token(s) are retrieved from the database.

- The sends a request to Plaid with the access token(s) attached.

- The Plaid response is then forwarded back to the client.

- The client displays the information to the user.

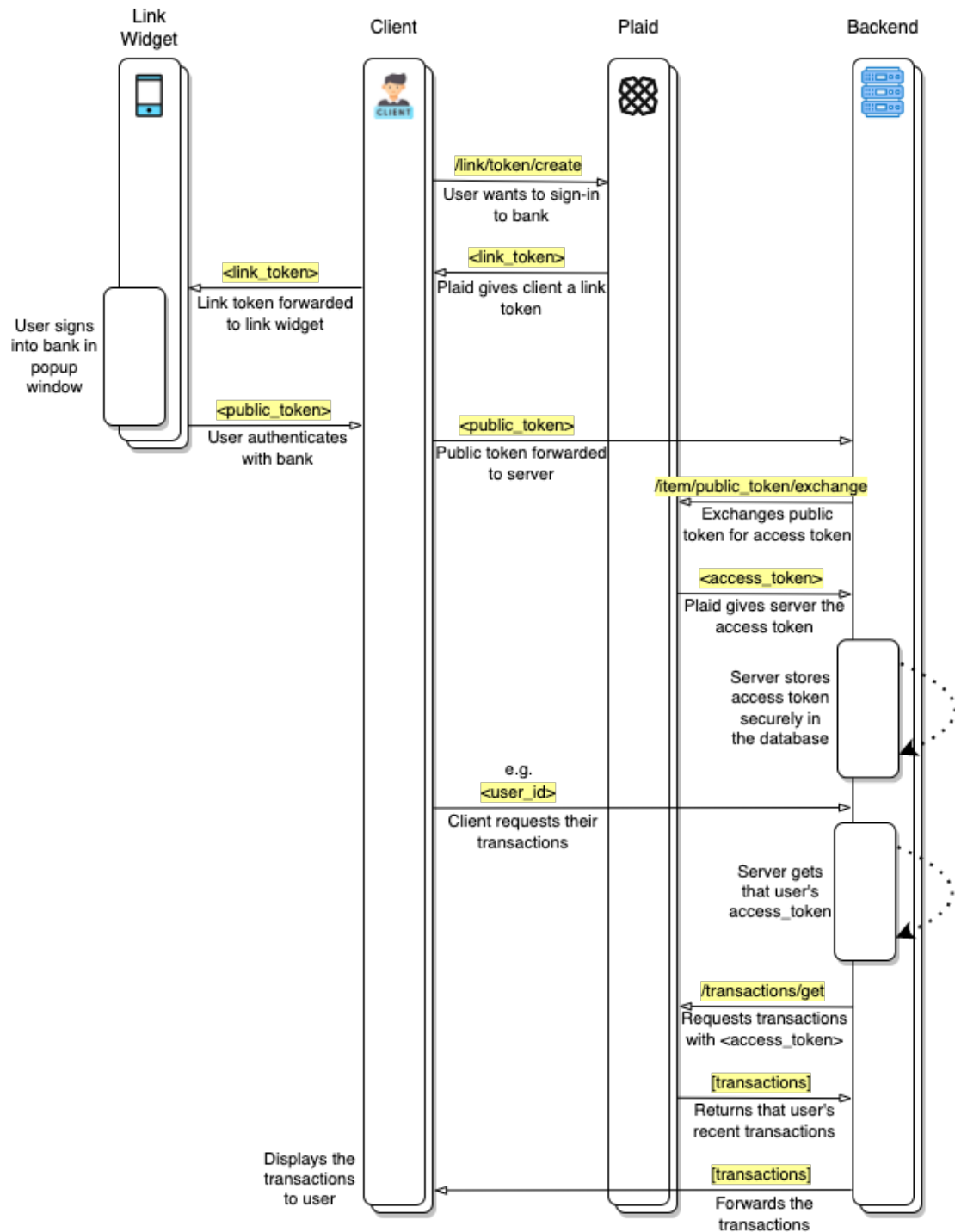This flow can be seen as a sequence diagram in the figure below (2.2).

Figure 2.2: Plaid Authentication Flow Sequence Diagram

2.3.3 Operation Modes

Further aspects of Plaid relevant to this project include the different modes of operation. When the web application queries Plaid's endpoints, one of three modes can be specified, and each produce different results.

The first is sandbox mode which is the default. All the endpoints and request formats remain the same, but the responses are fake data generated by Plaid themselves. This mode is helps test the web application without connecting to a bank, so do not worry about leaking any private information, or getting fast and reliable responses.

The second is development mode, where the responses are real data. The entire authentication flow must be followed in this mode, and a user will log in to their bank. To gain access to this mode, it must be requested by Plaid and they review the application to ensure 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. It is the same as development mode but with no limits. Further vetting is required to access this, and the queries now cost. For this project, only sandbox mode was initially used, and then following the completion of the prototype, development mode was requested and given for further testing and demonstrations.

2.4 Further Motivation

As previously mentioned, the primary motivation for this project came from the recent open banking movement, so using modern technology to help push the field. In addition, the currently limited services available were 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' [10].

The Money and Pensions Service's recent report advised that people get a

budget planner to help cope with the current times [14]. They are the government-sponsored financial guidance entity for the UK. The report helped concrete an objective for the web application to include budgeting features and would also help improve financial capability.

Chapter 3

Design

This chapter describes the design decisions of the web application and how it supports the project's overall goal.

3.1 Technologies

As mentioned in the background section (2.3), the web application utilises Plaid for the open banking aspect. It requires a frontend and a backend to follow the authentication flow for ensured security. Most of the tool is written in JavaScript and uses the Next.js framework; however, the backend also features a neural network model written in Python.

3.1.1 Next.js

Next.js is a full-stack React framework by Vercel [28]. It incorporates the React library for the frontend, and 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 and intelligent code optimisations.

Server-side rendering (SSR) is when the server computes what files are needed and sends them all in a single response to be displayed instantly. This contrasts with the default client-side rendering (CSR), where a single HTML file is sent

to the client. This file may reference JavaScript and CSS files, so the client then downloads these from the server before being able to render the page. The first contentful paint times can be slower for SSR as the server needs to compute what files are needed and send them, whereas for CSR the client can render the HTML only as soon as it is received. However, in CSR the page is not able to be interacted with until the other files are downloaded and executed. Therefore, the total loading time is faster for SSR. In addition, using SSR makes the application's component development less complicated.

Next.js also has inbuilt automatic optimisations that improve building and serving times. For example, a query to get a user's transactions goes through 3 stages. The first is the initial query from the client to the Next.js API route; then, this accesses the database and queries Plaid's endpoints; finally, Plaid makes the query to the bank, and the response propagates back. Therefore, any optimisations that improve the speed and help avoid slow data loading times are valuable and what Next.js provides by default.

Next.js is an extremely popular web application meta-framework with over one-hundred-thousand stars on GitHub [29]. It is well documented, and there is a large community, so it is easy to find solutions to common problems that may arise. Finally, Next.js is a framework that focuses on the ease of development to maximise productivity, which means the whole experience was more enjoyable and good software engineering practices were followed.

3.1.2 TailwindCSS

As part of the Next.js environment, the website serves JSX components that can be styled as standard HTML elements with CSS. TailwindCSS is a framework that contains a large amount of utility CSS classes. The application incorporates these classes to customise the way the website looks completely.

A different consideration for helping to build the UI is to use a component library. This may have cut down on development time, but they often look generic and unappealing due to the limited ability for customisation. TailwindCSS is more flexible and enables the UI to be built to the exact specification of the

design and the developer's vision. Like Next.js, TailwindCSS is also quite popular, with many resources and cheat sheets that aid development. Next.js and TailwindCSS are often used in combination, such as in the t3 web stack, as they synergise well [26].

The UI is a big focus for this project, as it 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. The UI design focuses on being appealing and minimal, so the user has more confidence in the application and, therefore, their finances. Furthermore, the user's data is displayed in an informative and easy-to-understand way, so they are more likely to absorb the information and make better financial decisions.

3.1.3 PocketBase

Part of the authentication flow with Plaid requires that the access tokens are stored to be used later. Each access token can only be associated with one particular user, but a user may have several access tokens. On top of this, a database is needed for basic user authentication where users can sign up and log in. PocketBase was determined to be most appropriate for these use cases due to its simplicity and fast setup.

PocketBase is an "open source backend consisting of embedded database (SQLite) with real-time subscriptions, built-in auth management, convenient dashboard UI and simple REST-ish API" [21]. 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 enables users to log in and persist their sessions across refreshes, making the user experience much more seamless.

Pocketbase features an interactive UI dashboard for the developer to manage users and aid in bug fixing. This dashboard improves the development experience as it helps to visualise the database structure and manage the tables inside. Finally, PocketBase has a JavaScript SDK that can connect to the database from the API routes. This means that simple JavaScript function calls can be used to interact with the database, rather than complicated REST requests with embedded SQL queries that have more potential for errors.

3.1.4 Python and TensorFlow

Budget prediction was proposed after the research into strategies that help build financial capability. The thought process and reasoning of why a recurrent neural network is used for the budget prediction is explained later (3.4); but for this section, all that is needed to know that it is implemented in Python, and in particular, using the TensorFlow library.

TensorFlow is a machine learning platform used to help build and optimise many machine learning models. Using this and Python allowed for faster development of the machine learning aspect of the project. Python is a language that is often used for machine learning for a variety of reasons. Nazar Kvartalnyi [13] comments that some of these reasons include the variety of libraries and frameworks, such as TensorFlow, that support the process; that there is an excellent community for giving support during development; and that it is simple, consistent and intuitive. Furthermore, the application developer already has experience with Python and TensorFlow, so there was a lesser learning toll when implementing the budget prediction.

Using Python for the budget prediction does have some drawbacks, such as the added complexity by using a second language, and the extra layer in communication between the application and the neural network. After training the neural network, it is pickled and saved to a file. This file is then loaded and hosted with a Python flask server to be treated like an API. It has a single route that takes the input data and then responds with the neural network's output. This simple solution meant more time was focused on ensuring the neural network is accurate and precise.
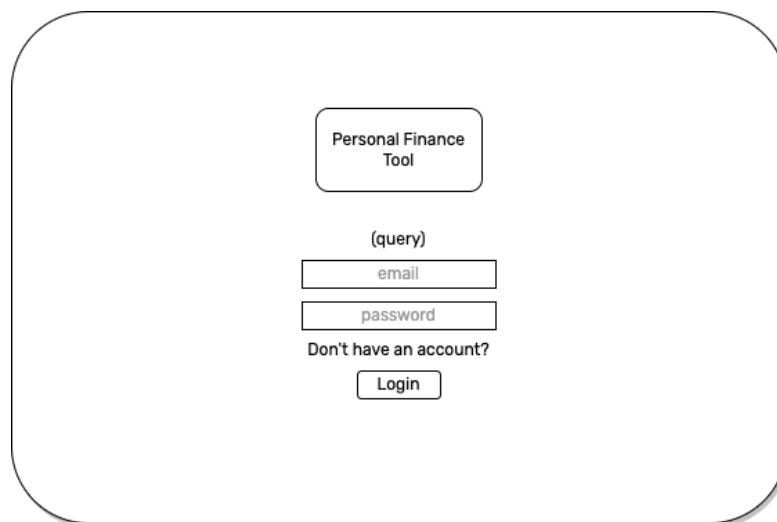
3.2 UI

As mentioned previously, there was a big focus is 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 precisely with appropriate CSS classes and JavaScript functionality.

Despite this, the design process that was instead followed was first to give the components their basic functionality; perform adequate component-based styling; go back to the functionality and modify if appropriate; and finally finish off the styling with the whole page in mind to make sure that is cohesive. This process was more suitable for the project because the styling did not compromise the website's functionality. In addition, on-the-fly styling is an added benefit of TailWindCSS as it allows for quick viewing of designs and fast changes.

3.2.1 Authentication Pages

When a user first visits the site, they must create an account. Research into the limitations of other services showed that having a quick and seamless experience at this stage is vital for the user. This contrasts with applications like Monzo and Revolut, where the signup process is long as they do not only sign up for the analytics but also to make a bank account. The focus for this application is simplicity, so the design for these pages is simple input boxes and a function button to log in or create an account.

Figure 3.1: Login Page Design

Above is the basic design for these pages. Most parts are self-explanatory;

however, where it says "(query)" is where any message or response can be displayed to the user. Examples of these include "you have successfully logged out", "that user does not exist", or "incorrect password"; and are given the relevant colours. In addition, the "don't have an account?" and "login" text are modified appropriately for each page that it is on - for example, if instead, the user is creating an account, it says "already have an account?" and "signup" respectively.

These pages were the first and only to be styled from a draw.io specification. This is because the process was relatively slow and tedious as it did not maximise the value of using dynamic styling feature of TailwindCSS. The more efficient process outlined in this section's introduction (3.2) was followed for the remaining of the pages.

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 would be presented on a single central page, and the ability to switch between the strategies would be done via a header bar. This is so quick access to all the information is possible, but also so that strategies not being viewed are preloaded in the background, avoiding long waits for the information to load and ultimately improving the user experience.

Transactions   Categories   Budgets   Investments   Accounts ›

Figure 3.2: Dashboard Navigation Bar

The header bar above (3.2) was the initial design for this dashboard. By default, the transactions strategy is chosen. When hovering over the other strategies, a light-grey bar underneath the strategy name appears; when selected turns this bar orange and removes the bar from the previous strategy. Underneath this

bar is the strategy itself; for example, all the recent transactions are below in the transactions strategy.

A limitation in the current systems outlined earlier (2.1.2) 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. This aspect was vital when thinking about the design of the header bar.

On the right of figure 3.2, in a slightly different button, there is an accounts dropdown. When clicking on this, all the bank accounts that the current user has linked are displayed here. Each account has a toggle icon to enable/disable the account which can be seen in figure 4.4. Only enabled accounts are included in the analytics, and when switching between strategies, only the selected accounts 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. First, they can disable all the other accounts to do so. Then, after getting this analysis, 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 strategy is implemented as a standalone component, and only one at a time is shown on the dashboard. However, as they are all on the same page, they are all styled in a similar way to be cohesive and match the general theme.

3.3.1 Transactions

The transactions strategy is the default strategy shown on the dashboard because it contains the information that is 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 financial adjustments accordingly. Many traditional banks do 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 unnecessary expenses.

From Plaid, the endpoint returns a list of transactions in reverse chronological order per account. Each transaction has an account_id, amount, date, time, location, currency code, merchant name, category (and more) associated with it. The web application groups the transactions from all toggled accounts by their date and then displays each transaction in reverse chronological order.

The transaction is displayed with the merchant name, category, amount and the bank logo of the account it is from. In addition, there is a separate label, for each date above all the transactions for that date. Finally, a distinction between income and expenditure is made by colouring the whole transaction background light-green for income and light-red for expenditure.
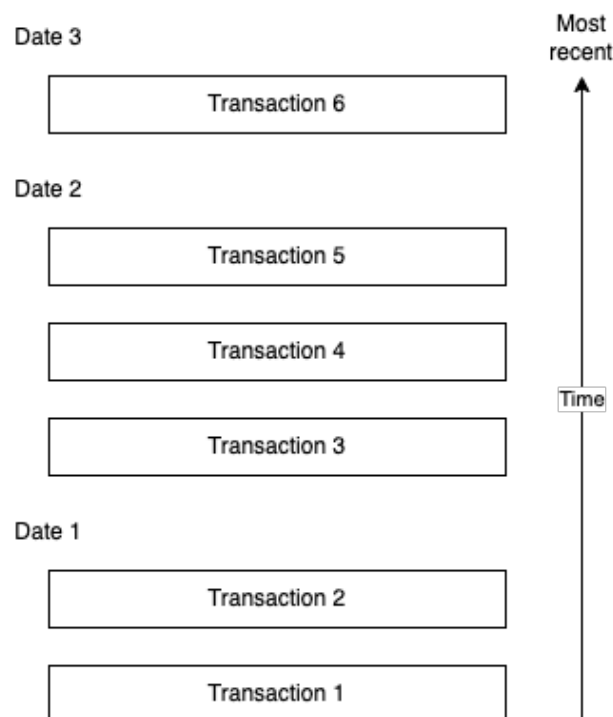
Figure 3.3: Example of Reverse Chronological and Date Grouped Transactions

3.3.2 Categories

The categories strategy is also helpful in identifying unnecessary expenses. In this strategy, the application only takes the previous thirty days and also only takes expenditure. It lists all the transactions in reverse chronological order again, but this time they are not grouped by anything. A pie chart is also displayed, where the values are the total amounts for each category. The user can then see the category they spend the most in and the relative amounts. In addition, they can use a filter to show only the expenditure for that category. All this data, again, is only for the toggled accounts.

The difference between this strategy and the transactions strategy is that the transaction strategy emphasises whether the transactions were income or expenditure and to which bank account. In contrast, in the categories strategy, the emphasis is on the categories of the expenditure and the total amount spent in each category. The input data is the same, but it is a different way of presenting it so that the user can get different perspectives on their finances and ultimately make better decisions.

3.3.3 Budgets

Two different methods of performing budget analysis were considered for this strategy. The first method is giving users 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 highly effective at helping people spend money on necessities while still having money for some treats and putting some aside in savings. The other method is predicting future expenses to incorporate into a user's budget calculations as budget prediction.

50:30:20 Budget Strategy

The article [15] by N26, a well-respected European bank, explains the strategy in more detail. In essence, it is when a person spends 50% of their income on needs, spends 30% of their income on wants, and puts 20% of their income

into savings. Several other articles back up the effectiveness of this strategy and help give it credibility; [24] even talks about how this method carries over into the cost of living crisis.

The method does have some disadvantages, however. The major one is that narrowing down which category each expense falls under is challenging. Often people disagree with what is considered essential, so it is already subjective. A typical example, found in several articles, is a gym membership. Some consider it essential, while others put it in 'the wants' section. Other disadvantages include that it is tricky for someone to alter their spending habits, and often if their income is relatively low, it may not even be possible to save 20% of their income.

The design for this strategy has an input element where the user can enter their income; however, also determining this value from past transactions. The website then breaks down their income into three sections and gives them values for what they can spend on each. For their previous thirty days of transactions, the usual period between each payday, the user can drag and drop each transaction into the three categories to help them see how much they are currently spending. If they spend too much on any, they can adjust accordingly.

Budget Prediction Strategy

The other method considered for a budgeting strategy was to perform budget prediction. This involves viewing how much a user has spent in the current month and performing pattern recognition to predict the remaining days. The user can then compare this to a budgeted amount they planned to spend that month and identify if they are on track.

Budget prediction is helpful for people who may, halfway through the month, see that they are under budget and then, for the rest of the month, overspend and exceed their budget. The strategy identifies that they often spend more later on in a month and incorporates it into the prediction. The user can then identify this pattern and account for this so saving money. A different use case is when someone tries to save money for a specific goal. They can see if they

are on track to reach their goal by setting a budgeted amount for each month to help adjust their spending habits to reach it.

Conclusion

When comparing the two methods, they each have different advantages and disadvantages; however, it is key to consider that the project's main aim is improving financial capability. Therefore, the budget prediction method is more appropriate than the budget breakdown method because it appeals to more users. In addition, it provides analytics that is extremely difficult for the user to replicate themselves, unlike the 50:30:20 method. Also, when predicting future expenses, the user can adjust what they spend sooner before it becomes a problem, thus making it more effective.

3.3.4 Investments

According to Danny's article on Finder.com, a reputable source for financial information and statistics, "Almost 1 in 5 Brits have owned stocks or shares" [3]. This is from a survey completed in 2023, which is recent and fact-checked by other journalists from the site. With over 13 million people in Britain alone, adhering to this demographic is valuable and popular. A handy tool, which continues the idea of getting an overview of the user's finances, is a central portfolio manager, where logged-in users can view and track their investments.

This strategy is the only one that does not utilise Plaid to access the data; instead the user adds it themselves. However, it is not a problem as, unlike the other strategies, once the asset has been purchased, they only need to add it once. Following this, when users check their investments, they can see the updated live price and the profit or loss they have made.

In order to perform this strategy, once the user has added their investments, they must be stored. One option is to store them in local storage within the browser. This method is the simplest to implement and, as stored offline, is somewhat secure; however, the disadvantages include that it is only accessible from the same device and browser. Users must add the investments again to

access the tool from another device. In addition, if the user clears their browser data, they lose the investment entry, so they must manually add it again.

The other option is to persist the investments using a database. As a Pocket-Base database is already being used for the application, this only requires a little extra work and complexity. The significant advantages include that, unlike local storage, when using the database, the information is accessible and up to date from any device or browser, so the investment only needs to be added once. Also, by linking the investments to only be accessible by the user who created them, they are incredibly secure and can be treated like the Plaid access tokens.

Ultimately, the database option was chosen because it synergises well with the system's current design and has the same advantages as local storage. The local storage option became the contingency plan, so if during implementation a problem arises, the local storage option can be used.

As well as persisting the investments, the application also gets the current price of the investment to determine if it is profitable. It gets the current price by requesting a service that provides live investment information. Financial modelling prep is an easy-to-use service that provides quick and live stock prices [23], which is ideal for this task. Given the price that the investment was bought for, and the current price, the application can calculate the profit. This information can be displayed as green for positive and red for negative. These colours allow the user to identify which investments are profitable and can continue and which ones are not, so action can be taken.

3.4   Budget Prediction

Once decided that budget prediction was the method for the budgeting strategy, the actual underlying prediction method was designed. There were three considerations: a basic manual pattern identification method, using linear regression to find the relationship and a neural network to find the trends. Unfortunately, for each of these methods, the only accessible real training data is the author's past three years of transactions. Therefore, this factor was also be

considered when deciding which method to use.

3.4.1 Manual Pattern Identification

Manual pattern identification is the simplest and involves manually identifying the patterns. Given the training data, set periods are suggested. The data is split into these, and it is determined which one and by how much is the most significant period. For example, given the three years of transactions, the data is split into the different seasons and it is identified in which season the most is spent. Using this knowledge, the standard prediction value can be increased if a query is requested during this most expensive season.

This method can be repeated on different features, each of different periods, such as weeks, weekends, months, and national holidays. Once many different factors have been analysed, the predictions can be made by combining all of these.

This method is swift at making predictions but is extremely complicated to implement. Furthermore, the predictions may not be very accurate as some features may not be factors, and some features may not have been considered. Ultimately, this method is a trade-off between 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]. When applying this to budget prediction, the output expenditure is the scalar response, and the explanatory variables include the budget set out by the user as well as the current date. The model finds the relationship based on the past three years of data, and then, when a query comes in, it can quickly use the learnt relationship to output an expected expenditure.

The main advantage of this method is that it does not involve any manual pattern identification because, instead, the relationship is computed. Despite

this, it is still not that accurate as the relationship between expenditure and explanatory variables almost definitely is not linear, which is the major downfall. A solution is to project the input data into a higher dimension. This method allows the model to find a linear relationship in this projected space but becomes a better non-linear relationship in the original space. This solution may improve the accuracy but requires much more computation and may be considered over-engineering for this problem. In addition, it does not even guarantee a better result.

3.4.3 Neural Network

The final option that was considered was to use a neural network. Neural networks take some input data and, with lots of training data, automatically identify the features and learn the relationships. This option has many variations as each model and structure can differ. The neurons themselves can have different activation functions, but also the connections between them. The networks can be feed-forward, with no loops, or be a recurrent neural network, with some feedback (loops). Each layer can have a various number of neurons, and the number of layers can also be varied.

Acknowledging this, provided the network is designed effectively, it can outperform the other two methods in terms of accuracy, which is ideal for this problem. However, the drawbacks include that it involves a lot of testing and tuning to get the best results. The limitation on input data is, therefore, more prevalent.

3.4.4 Conclusion

Out of the three options described above, the most applicable model for the problem of budget prediction is a neural network. This is because it does not rely on manually finding the features to use, so it identifies even the hidden and complex patterns and, after being trained, can generate quick predictions with high accuracy. The implementation of the budget prediction is described

in much more detail in section 4.3.3, but a quick overall explanation is given below.

Deciding on the neural network's architecture involves lots of trial and error to identify what is most appropriate. Due to its niche nature, there is little academic research on predicting expenditure given past transactions. This meant the problem needed to be generalised. The vital idea that helped was to think of the problem as a time series problem.

The input data can be treated as a time series by modifying it from a list of chronological transactions to the cumulative amount spent each day and resetting every it month. This problem has much more research because time series prediction is a significant area for many industries, such as stock market prediction. Many ideas applied to time series data found that using a recurrent neural network is the most effective.
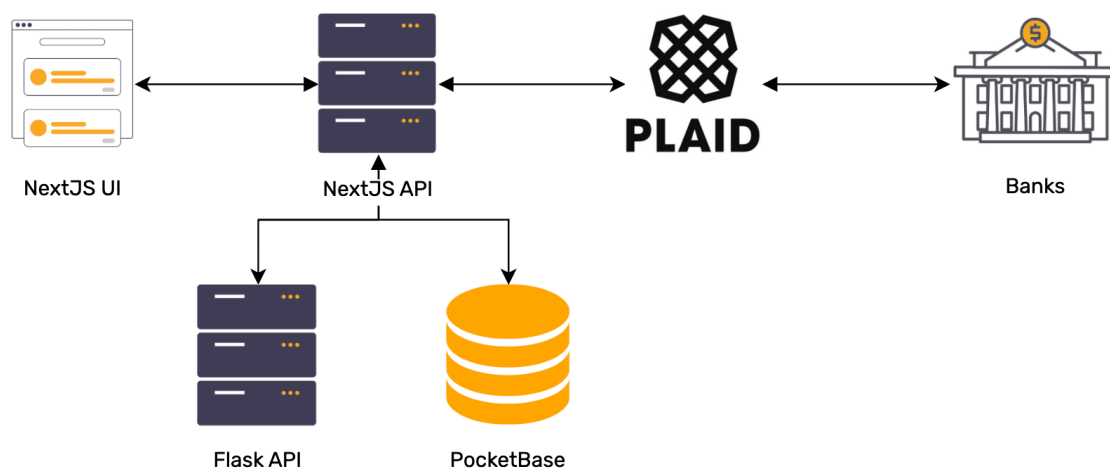
3.5   Software Architecture

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 parts 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 is required for the Plaid authentication. The Next.js API routes are how the UI communicates with Plaid and the database. An example API route is "get_transactions", where the backend returns the transactions received from Plaid, which returns the transactions from the bank.

The access tokens and other long-term information, such as the user's login details, are stored in the PocketBase database. Only the backend can access this; furthermore, the access tokens can only be accessed by the user who created them in the first place as an added security measure.

Finally, there is the Python Flask API which contains the budget prediction neural network. This API only has one route. It takes in the input data, propagates it through the neural network, and returns the output. Only the Next.js API routes query it as the input data requires the user's past month's transactions; therefore, they must be requested from Plaid.

When the Budget prediction page is accessed, the frontend queries the Next.js endpoint. The Next.js endpoint first queries Plaid for the transactions, interacts with the database for the access token and then queries the Python Flask API. The Python Flask API returns the prediction to the Next.js API, which returns it to the frontend to be displayed.

3.6 Endpoints

Various Next.js API routes are required for each of the strategies that the web application exhibits. All of these routes are prefixed with "/api" to be accessed by the client and all only accept requests using the HTTP POST method. This is because the requests often contain contextual information for the server to decide what to do; for example, they contain the user identifier to determine which user's transactions are being requested.

The information passed between the client and server is in JSON format because it helps represent the data in a structured way, and is 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 is written such that it can only perform its intended function or return an error. This is because all the error checking is done on the client side to improve security, reduce the amount of code written, and improve the user experience for graceful error handling.

3.7 Legal, Ethical and Professional Considerations

As the application handles sensitive user data, many important considerations were taken to ensure the application is secure and ethical. The most important of these is ensuring that the application only gives financial guidance rather than financial advice.

3.7.1 Financial Guidance vs Financial Advice

In the eyes of the Financial Conduct Authority, financial advice is a "personalised recommendation" where the provider is "responsible and liable for the accuracy, quality and suitability" [9]. On the other hand, financial guidance is a "suggestion" or "service to help you identify your options", where the provider is still responsible for the accuracy but no longer liable.

In the case of this application, all the information is completely personalised to the user, so the emphasis is made that there are no recommendations. The strategies instead aim to give the user as much information as they want to make financial decisions, rather than making the financial decisions for them. This means each strategy displays information more clearly and practically, rather than taking their information and giving a recommendation.

3.7.2 Data Protection

The correct responsibilities and procedures were followed as part of handling user data. These include the amended Data Protection Act and the General Data Protection Regulation. Despite the application not currently being public

facing, the website aims to be eventually, so the application is designed with this in mind. As well as following the strict laws, general data protection techniques are used. For example, the passwords are hashed and salted before being stored in the database, or information specific to a user is only accessible by the creator of that information, not even the developer.

As well as this, there was also an emphasis to try and avoid storing the financial information in the database. In the case of the transactions strategy, some design decisions were adopted to prevent this. For example, rather than storing the recent transactions and only getting the new ones each time, which would have been faster, the application was designed to get all the recent transactions every time as to avoid storing them. Only where necessary, like in the investments strategy, is user-specific financial information stored.

Chapter 4

Implementation

4.1 Project Management

The table below is the initial project timetable from the project specification document. It was used as a rough guide to keep the project on track by helping to 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 more evident that the project had four key phases instead. The first phase was research into

the development methodologies and design. The second phase was the development of two proof-of-concepts and the merging of 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 was 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, but it is still worth discussing how the research was conducted and the structure of this process. This phase was performed before the progress report and timetable change, so it aligns with the original project specification.

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

Plaid is a relatively new technology, but it is still like any framework, so a standard research methodology was used to understand how to use it. Vinicius' article [1] talks about the best approach for learning a new language or framework and has almost ten thousand claps (the measure of Medium's popularity), so many others agree. The article is a good summary of the whole process. The main points that were followed when learning Plaid are: start with the basics, read lots of documentation, find examples on GitHub and Stack Overflow, and finally, build something. In addition, due to Plaid's YouTube channel [18], watching videos was also included because it provided an effective way to learn.

Plaid has a getting started video which helps explain the service's basics and the terminology used in the documentation. Following this, the Plaid docu-

mentation was read to understand the service's architecture and how it works. In addition, learning its features helped identify what features could be added to the application to improve financial capability. Unfortunately, due to the recency of the revised architecture of Plaid, there were few examples 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 that helped to understand the service.

Learning to use Plaid helped inform many other design decisions; for example, the Plaid authentication flow dictates the need for separate frontend and backend services to be developed, and a database to store the users' credentials.

The other major development technologies researched here included Next.js features that the developer had not used before, such as server-side rendering; the combination with TailWindCSS; and the use of PocketBase. The research method into these technologies was very similar to when learning Plaid, as it had proven effective. The main difference is that more documentation and resources were available, making learning slightly easier. As well as technologies, new 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 primary reason for the project's next phase of building two proof of concepts. It was to test and improve the knowledge of the technologies and concepts learnt, but also provided a reasonable basis for the actual application itself.

4.1.2 Proof of Concepts

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

This implementation phase was the first deviation from the original plan in the specification. In this instance, the development followed the waterfall methodology. This is because it lends itself nicely to how this phase is structured. A clear step-by-step plan was constructed with the knowledge of what the technologies can do and a clear goal for each proof of concept. In addition, there was only one developer, so the waterfall methodology was an excellent way to ensure that the project stayed on track and that the developer was not overwhelmed with too many tasks.

Waterfall is known for being effective in projects with concrete objectives. Both applications were worked on simultaneously, performing each stage of the waterfall process for both applications. This was done to ensure that the applications can interact with each other when the time came to merge them. For example, the second stage of the methodology is designed, so both the applications' designs were set out simultaneously and both before the implementation (the next stage).

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

User Authentication

User authentication is needed to allow only logged-in users to access their dashboard, and only each logged-in user can access the transactions for the access tokens they created. The design for this application was to have three pages. The first page is the signup page, where they can create an account. The second page is the login page, where they can log in to their already-created account. The final page is the dashboard which displays the user's email if

logged in and redirects to the login page if they were not. The signup and login pages have links for users to navigate between them.

The implementation of user authentication is simple yet secure. The signup and login pages have two inputs for the email and password and a submit button for the form. When a user creates an account, the client first checks that the email is valid and unique and that a password has been entered. If so, the database is populated with the new account and the user is sent to their dashboard. The user is notified via an error message if the email is not valid or unique. The PocketBase npm package is used to make the appropriate calls when creating the account or logging in. In addition, this package automatically hashes and salts the passwords for security.

The login page works similarly, but instead of creating a new account, it 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. Using these tokens means they can hold the user's ID and allows the database to limit access to the user's data by only allowing requests from that user. The ID is encrypted in the JWT, which itself is stored in a cookie. In addition, the JWT is only valid for a short period 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.
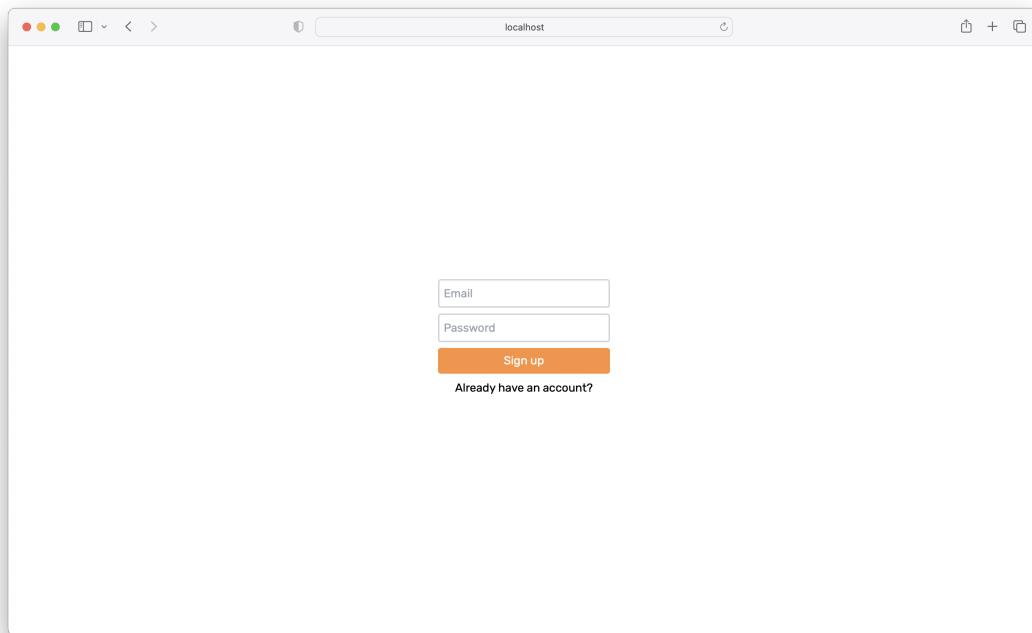
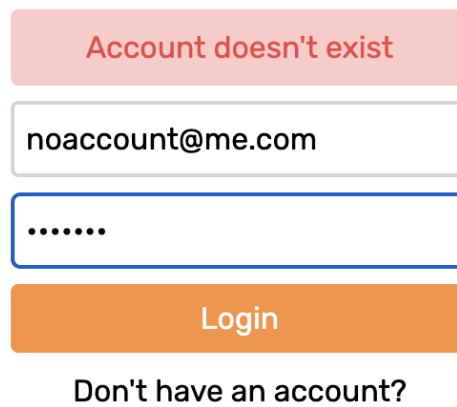Figure 4.1: Minimal Signup Page for the User Authentication

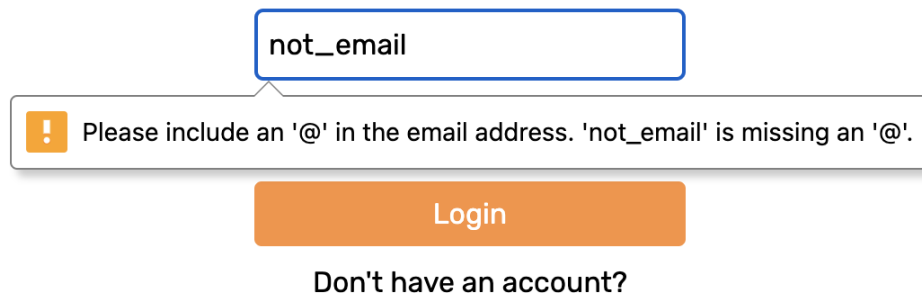Figure 4.2: Login Component, with an Example Error Message

Figure 4.3: 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 allows a user to follow the authentication flow of Plaid in linking a bank account and getting the resulting access token. Then, with this token, access their recent transactions. To begin the process, a user clicks on a button to request the link token and has the link widget pop-up. The rest of the authentication flow is as described in the background section (2.3.2), and after linking, the access token is just stored client side at this stage.

This system required minimal design and little styling as the authentication flow was already decided. This proof of concept was aimed at having functionality, so there was 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 there was little deviation from the complete plan created in the phase before.

Merging

At this stage, there were two independent proof-of-concepts that served different purposes; however, the project needed 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 were on separate branches. The basic user authentication was deemed more fragile, so the Plaid interaction was decided to merge

into that application. The idea was to have the user perform the basic authentication; then, on the dashboard, they can link their bank accounts, and then each access token generated is stored in the database. Furthermore, if that user already had an access token available, then on the dashboard, they can request their transactions, which is displayed in the console.

With the waterfall methodology and knowing how to implement these systems independently, each stage was meticulously planned. Hence, the merge went smoothly, and the bare-bones application was ready to add the strategies. 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. It was similar to performing basic user authentication, such as getting a user's email, and there were many resources online to help. This aspect was added as a feature branch and then merged into the main branch, to follow the idea of GitHub Flow explained below (4.1.3).

During the implementation, something that was not considered is when a user has multiple access tokens and what to do. This oversight was easily mitigated, as the rest of the functionality was implemented per the original plan, and then, only after this was completed, the new feature was added. The slight modification was made so that when requesting the transactions, instead of getting the first access token, it returns all the access tokens, and then for each, a query is made to Plaid. This differs from a user with several linked bank accounts, as each access token can contain multiple bank accounts. However, at this point the ability to toggle the individual bank accounts was not worried about.

4.1.3 Repeated Strategy Implementation

By this stage, the application was elementary and only could 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 was not any particular sub-methodology like scrum or lean, but instead just had very agile practises

and was worked on in sprints. Each sprint was for a different strategy, so they were independently added but came together to form the final application. This separation was also shown during development as each strategy was implemented on a separate branch and then merged after passing the tests and being mostly complete.

Agile Development

Generally, the waterfall methodology should be prioritised for a project of this scale, especially where there is only one developer. However, agile was more appropriate for the repeated strategy development due to the requirements being challenging to define prior to each strategy. Each strategy, and therefore sprint, contained research into 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 were 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, especially for research, so it was modified accordingly. Agile's ability to change on the fly was a significant benefit, particularly for this example. The sprint was reduced to roughly ten days, and there were no set periods for each section (research, development and testing); instead, the time was flexibly 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 was 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 and the secure authentication flow, the application was approved. As outlined in the background section (2.3.3), development access means that real live financial information can be accessed and used. This was a significant milestone, but it also came with further things to manage.

With development access, the application can have one-hundred access tokens in total. The limitation meant the credentials were treated as a resource and used infrequently to ensure that enough were available for the final testing stages and the demonstration in the presentation. On top of this, all unit tests were only ever run in sandbox mode and only once these were passed did integration testing begin. Integration testing was first done in sandbox mode, but once happy, the same tests were performed in development mode. More often than not, no issues arose in development mode, but it was still essential to test in both modes to ensure the application's reliability.

GitHub Flow

As mentioned earlier, each strategy had its own branch and only merged once it was complete. This was an 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, 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 and ones that need stable releases at set 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 additional features like budget prediction. Each 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. Some integration tests were 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; however, 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. A valuable feature that does this is predicting a user's expenditure and allowing them to adjust earlier, ultimately making better decisions.

Unlike the previous strategies, the budget prediction was added as a separate major feature because it was much more extensive. In addition, more research and planning was required as it involved creating machine learning models and a separate server to host the model. By this time, only minor tweaks were being made to the MVP as hotfixes, and 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. Going into the prediction, there was little certainty about how effective it would be or what the requirements were outside of predicting future expenses.

As part of managing this section, the machine learning models needed real financial data for training to be the most accurate. This has privacy concerns as the model learns from this data, and so the resulting predictions can leak information. Furthermore, there was a limitation on the amount of real financial data accessible, so it was decided to mix the data. The model was trained using only the developer's financial data and some of the data from Plaid's sandbox mode. This was in an attempt to still create a useful model, but not leak any information and 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   Active Bank Accounts

As part of the navigation bar shown 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 a toggle button, the name and type of the account, and a logo representing which bank chain it is from. The add accounts button allows users to link more bank accounts by beginning the Plaid authentication flow. The logout button lets the user log out, and the application will clear the cookie.

This functionality was implemented as part of the transactions sprint as it was not a part of the initial proof of concepts; however, it is necessary for all the other strategies. All the bank accounts are active by default; however, the user can toggle them off to exclude them from the analytics. Each bank account has an associated ID, and so a list of active bank account IDs are stored as state in the navigation bar. This array is passed to each strategy component as a prop to filter the data and only include the active accounts.
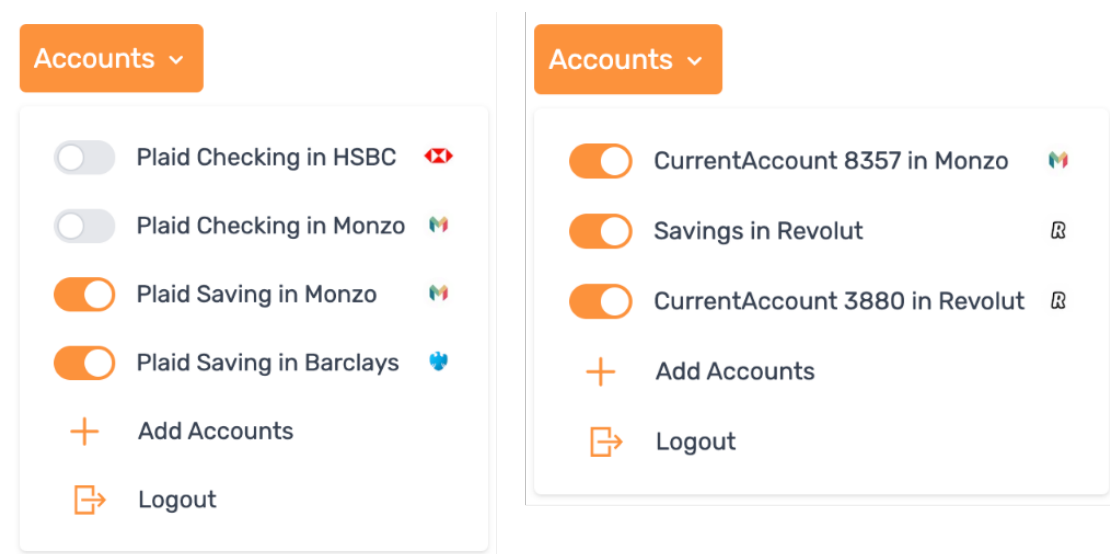
Figure 4.4: Account Dropdown Component for Sandbox Mode (left) and Development Mode (right)

In figure 4.4 above, 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 only to view the analytics for those accounts. In development, all the bank accounts, a Monzo and two Revolut accounts, are enabled. When switching between a toggled account, the icon has a slight animation and when hovering over the account, the background colour changes to a light-grey.

4.3 Strategies

This section aims to outline the strategies that were implemented in the application. Only the final option that was decided in their respective 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.

4.3.1 Transactions

The transactions strategy was the first implemented, so there was a lot of experimentation and learning how best to get the data from Plaid, propagate it to the frontend, and then display it 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 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 the order of the accounts, so some extra data preparation was needed.

For this strategy, a single Next.js API route was required. It takes only the user ID as a parameter and queries the database for all the access tokens associated with that user. It can only see the access tokens that it created, so in this case, it would just get all the access tokens. With these, it requests Plaid to get the transactions for each access token. The transactions are then grouped and sorted in the API route so the frontend performs 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 from after the cursor. This reduces the data flow and means the transactions can be stored locally for faster loading. On strategy load, however, the UI needs to send two requests, one to the database to get the historical transactions, and one to Plaid to get any since.

The Plaid API call ended up being changed to use "/transactions/get" instead, which returns the transactions between an input start date and end date. This was mainly because the transactions did not need to be stored so improves security. Plaid encourages the use of the "transactions/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 Component in Development Mode

The above figure (4.5) is an example of the transactions strategy with two separate bank accounts enabled in development mode. Each transaction has its own bar and displays relevant information to help provide an overview. In addition, income is coloured light-green and expenditure is coloured light-red. Whenever a transaction is hovered over, it is coloured slightly darker as feedback. Often, enough transactions are shown that scrolling is required; however, there is no consistent way to style the scroll bar as it depends on the browser. To account for this, the scroll bar was removed entirely, but the bottom-most viewable transaction has an opacity gradient to indicate more transactions are below.

4.3.2 Categories

The categories sprint followed naturally 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 to automatically separate the data into income and expenditure (by positive and negative value). However, it would have involved some slight changes to the transaction strategy. So 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 sped up the data processing by creating a different API route.
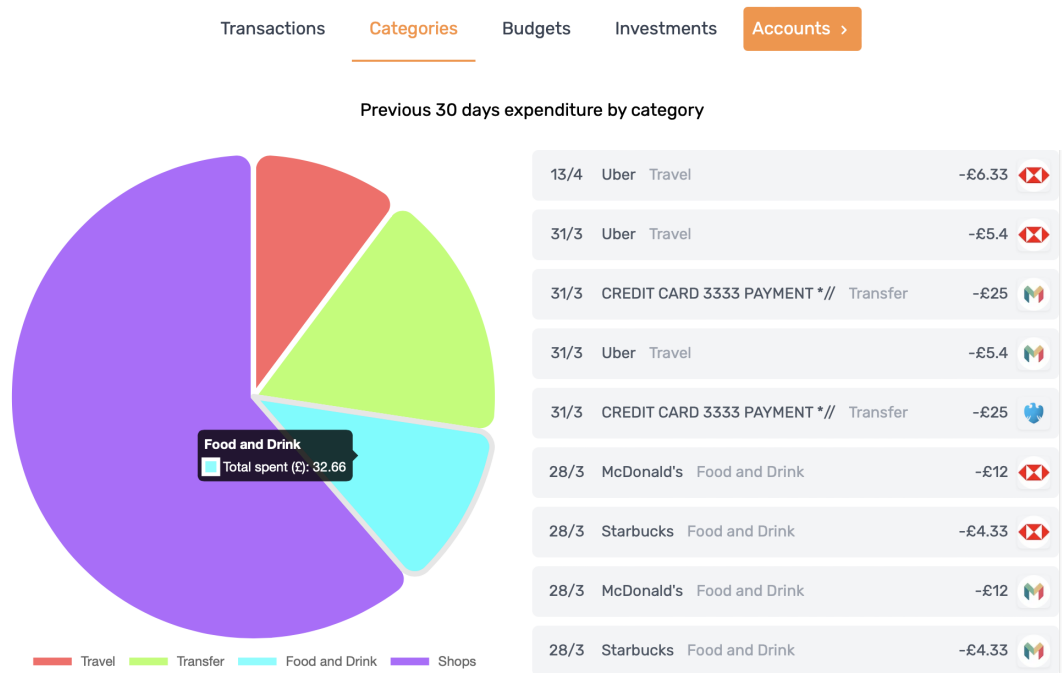
Figure 4.6: Categories Strategy Component in Sandbox Mode

The above figure (4.6) is an example of the categories strategy component with four accounts enabled in sandbox mode. 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 four categories, so four colours that are the most distinguishable from each other are chosen. In the figure, the user is hovering over the light-blue section, 'Food and Drink', so the total for this section is displayed. On the right is the list of all expenditures 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 the categories strategy page in development mode with two accounts enabled. In this case, the pie chart is more realistic as it comprises 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 to click 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.

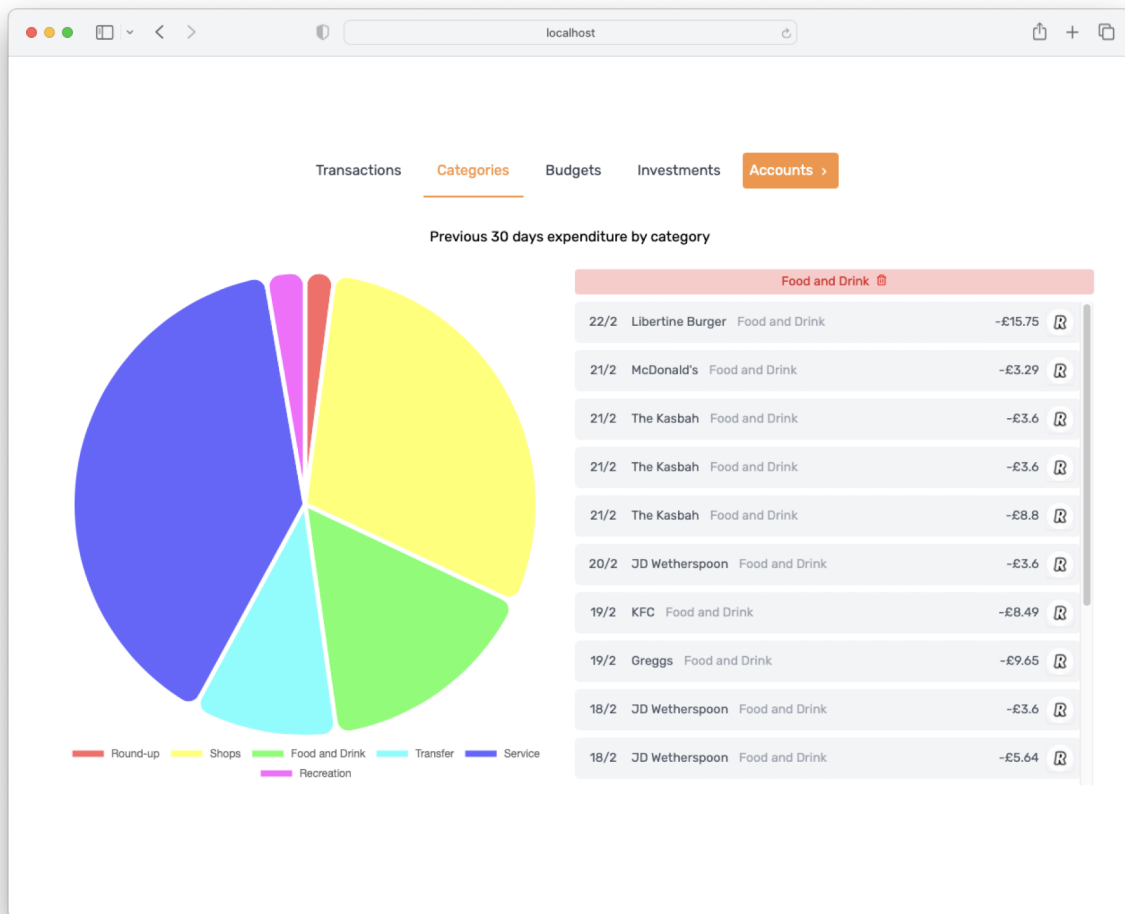Figure 4.7: Categories Strategy Page in Development Mode with a Filter

4.3.3 Budgets

There was no budget prediction in the original sprint for the budgeting strategy. Instead, the sprint resulted in a graphical view of the previous expenditure by displaying the amount spent each day, with a total amount on the side. The user can also enter an amount into the month's budget, which is also displayed

on the line chart. The research portion of this sprint did identify that budget prediction is a valuable feature, but it was not considered at first because it involves more time than a 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 Forcasting

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

The book Forecasting: Principles and Practice (3rd edition) [11] was a particularly useful resource for this section. It talked about the effects of seasonal time series, in this case, cumulative resets; how to apply linear regression techniques, which helped decide to avoid this; and using a neural network to make predictions.

Long Short-Term Memory

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

LSTMs differ from ordinary recurrent neural networks because the individual units have single-value memory cells that three gates can control. The first gate is an input gate determining if the cell's current value 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, determining whether memory should be cleared. Each gate has an associated weight which is learnt during training like ordinary neural networks.

Training

As part of training the LSTM model, the input training data, consisting of the past three years of transactions, was first modified. The data was turned into the cumulative data and reset at zero every thirty days, but in addition, it was split in such a way as 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 following training example, the thirty-second value is the predicted output. This method is shown more clearly in the figure below (4.8) and resulted in 1064 training examples. Techniques such as test-train split and validation sets were also used.
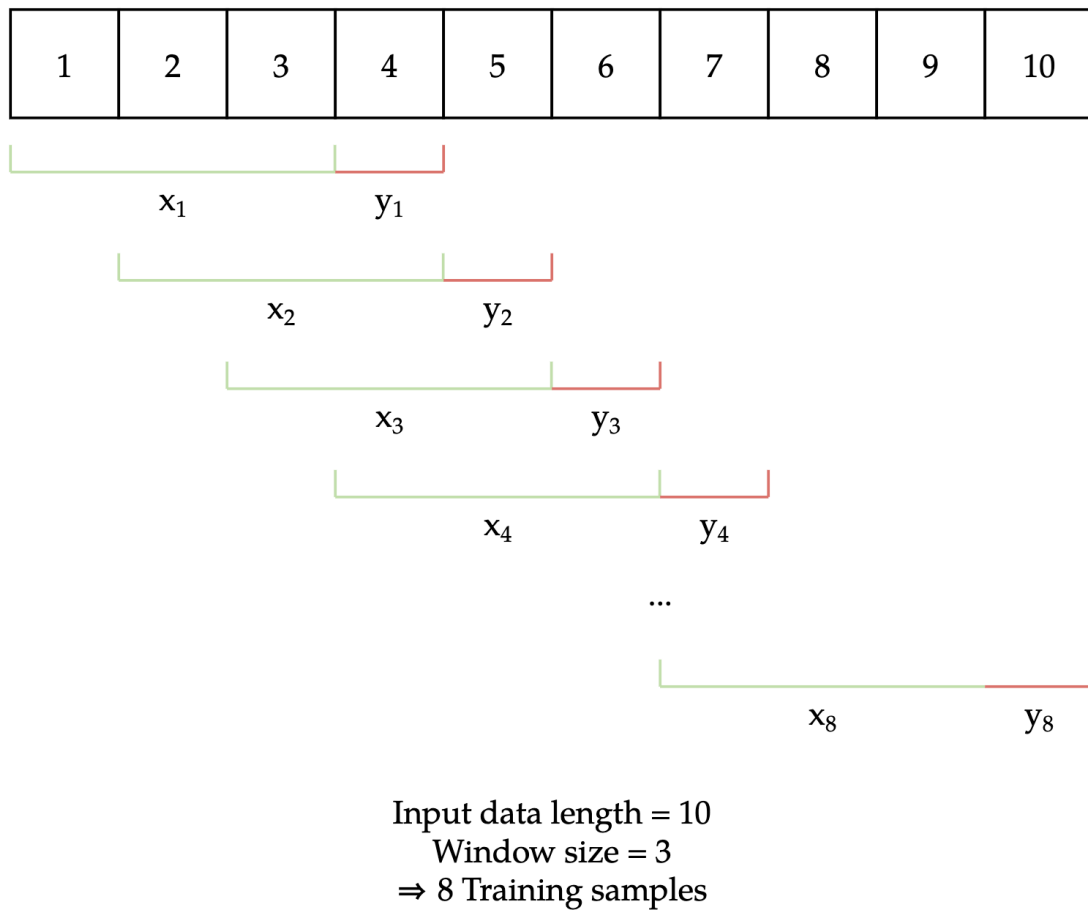
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

x_1 y_1

x_2 y_2

x_3 y_3

x_4 y_4

...

x_8 y_8

Input data length = 10
Window size = 3
⇒ 8 Training samples

Figure 4.8: LSTM Training Data Generation

Architecture

Throughout the research and testing phase, LSTM models were the most accurate relative to other neural network architectures and machine learning models like linear regression. This also complimented the fact that there is support for LSTM models in the Keras library. The best structure of the network, which was found to be sufficiently simple that it could be trained in a realistic amount of time but also sufficiently complex to learn the data, was a single LSTM layer with 64 units, followed by a dense layer with 1 unit as output. Furthermore, 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 to match the modified training data and budgeting period.

Results
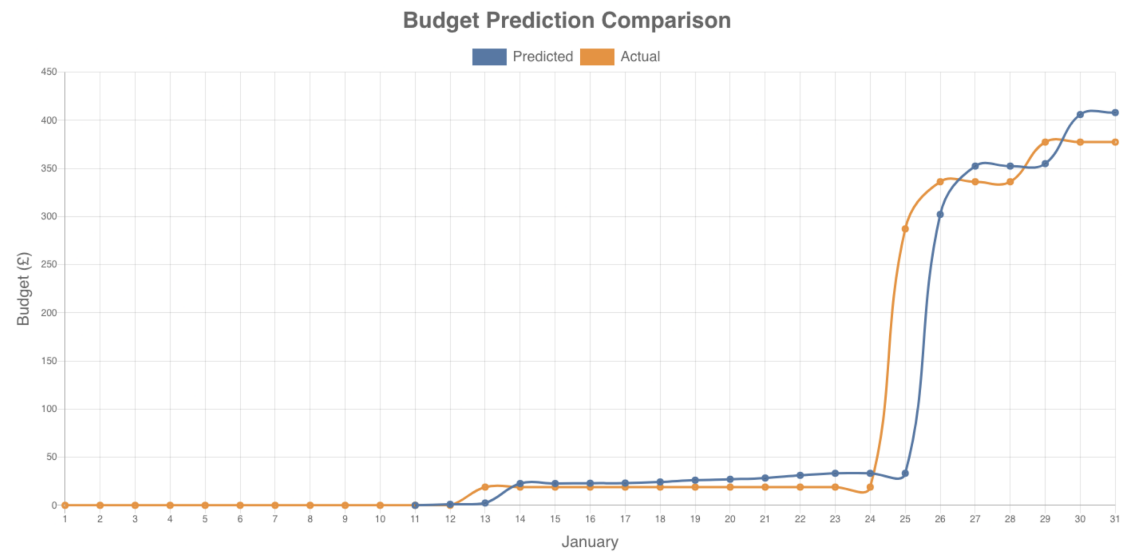
Figure 4.9: Budget Prediction Analysis

The above figure (4.9) displays the difference between the predicted vs. actual expenditures. In this instance, the data used for the graph is the fake data generated by Plaid, so it is not even related to the data on which the model was trained. Nevertheless, the model was able to predict the expenditure accurately and follow the pattern effectively. There is a slight offset in the two lines, but this was found to be due to how Plaid generates its transactions. They assume that each month has thirty days, but the graph is for January, which has thirty-one days, thus explaining the offset.

Budget Strategy

The accurate budget prediction model is incorporated into the budget strategy by making predictions for remaining days in the month. For example, suppose the user is accessing the tool on the fifteenth day of the month. For the past fifteen days, the tool will have access to their actual transactions and can plot the cumulative expenditure. Predictions can be made and plotted for the remaining days of the month.

The prediction is made by taking the past thirty days' expenditure and predicting the amount on the sixteenth day of that month. Then using this predicted value and the true past twenty-nine days of expenditure, an amount is predicted for the seventeenth day, and so on. These values are then displayed to the user in the graph, which they can compare to their budgeted amount.
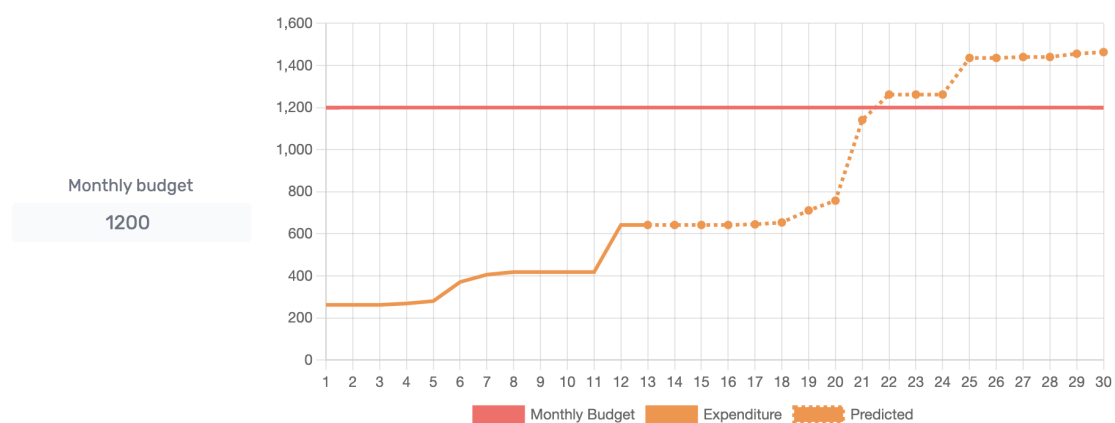
Figure 4.10: Budget Strategy Component in Development Mode

The figure above (4.10) shows the budget strategy in development mode with the budget prediction incorporated. The date of access is the thirteenth of April, so there is actual expenditure up to this date. The remaining days contain the predicted expenditure. The fixed monthly budget is £1200, but according to the graph, it is predicted that they will exceed their budget, so they should adjust accordingly.

4.3.4 Investments

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

The user is expected to add their investment initially; however, the tool should automatically update following this. This means storing the investment in the database. To accommodate this, a new Next.js endpoint was required to take the relevant information about the asset, such as the ticker and the price it was bought for, and store 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 another to delete an investment

The UI displays all the user's added investments when they access the investments strategy. This is done by requesting the new endpoint to get the investments. For each investment, once the stored data is retrieved, it accesses a live API to get the actual current price of the asset. This is used to calculate the profit and percentage change. An investment that has made a profit is shown in a light-green, like income in the transactions strategy. An investment that has made a loss is shown in light-red, like expenses in the transactions strategy. Each investment also has a delete button which calls the delete endpoint and updates the UI accordingly.

The information retrieved from the database can still be displayed whilst accessing the Financial Modelling Prep service. This delay means that initially a loading icon is displayed for the data that takes longer to access. The figure below (4.11) shows the table that has loaded the information from the database, but not the current price from Financial Modelling Prep. The figure below that (4.12) shows the table once the investments, price and profit have all been loaded and calculated. Each investment is treated independently; this means they can load independently, so even if one investment takes longer, the others will still be displayed as soon as they are ready.

	Transactions	Categories	Budgets	Investments	Accounts ›

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

+ Add Investment

Figure 4.11: Investments Component Loading

	Transactions	Categories	Budgets	Investments	Accounts ›

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 4.12: Investments Component Loaded

The table of investments has a max height; this means that if the user has many investments, they do not extend the page downwards to display them all. Instead, the table is made scrollable. This allows a portion of the UI below the table to be for how the user adds an investment. The ticker input is a dropdown and allows the user to select a supported ticker (effectively an investment identifier). Once a ticker is selected, when entering, the price is automatically filled in with the current price as an aid. The user can enter the quantity and the amount they paid for the investment. 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 that it can be corrected. Once the user has entered this information, they can click the add investment button, which adds it to the table above and the database.
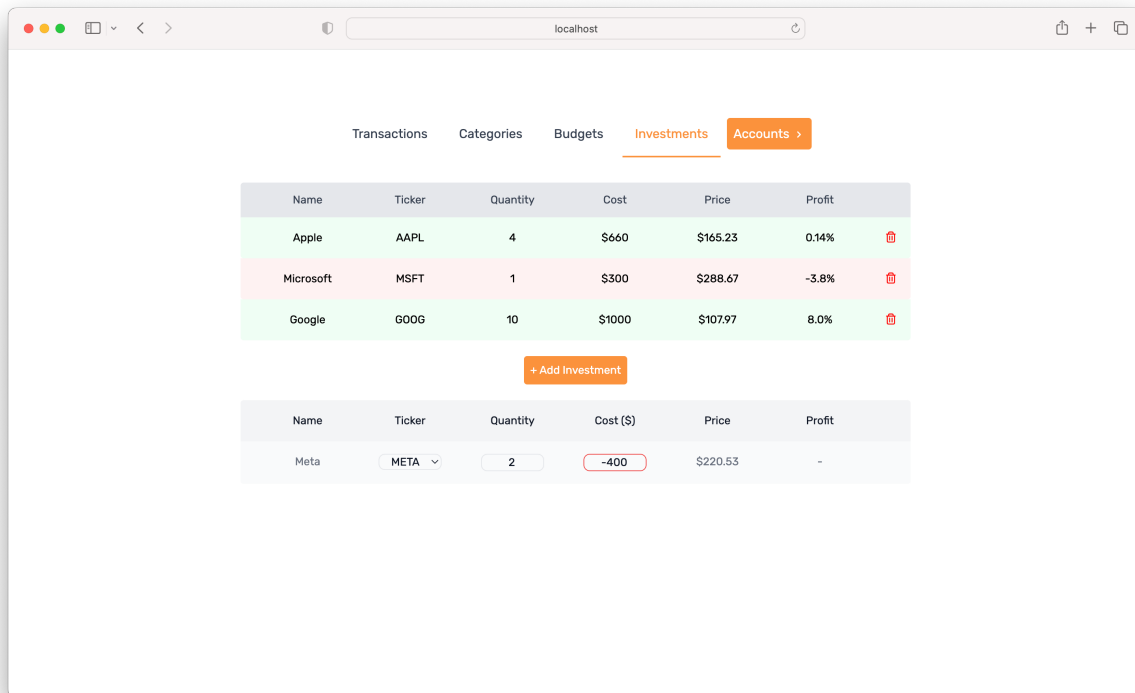

Figure 4.13: Investments Strategy Page with Invalid User Input

The figure above (4.13) shows the user attempting to add a Meta investment. Everything is valid except for the cost, which has been mistyped with a negative sign. The validation picks this up, and the box turns red. The user can correct this and click the button to add the investment to the table.

4.4 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 4.14: Endpoints Table

Chapter 5

Evaluation

5.1 Testing

Testing was performed throughout the development of the application. This has mainly been manual testing focused on unit tests such that each component completes its intended purpose and is robust. Some integration testing and error handling was also implemented to ensure the application can handle unexpected behaviour.

5.1.1 Unit Testing

Unit tests were performed in sandbox mode throughout the implementation to ensure that each component and API route performed as expected. As well as this, some unit tests were attempted in development mode, but this was a lot more difficult to test as there is a limited amount of credentials for Plaid, and the data is live so constantly changing. Similarly, for the budget prediction, there is a limited amount of data to use, and most of it was used for training.

5.1.2 Integration Testing

On top of unit testing, integration testing ensures that each application aspect works together as expected. Each strategy was tried with every bank account

available on sandbox mode and with as many bank accounts available on development mode. In addition, anywhere there was user input, all input types were tested; this includes valid, erroneous and edge cases. Ideally, this would have been done with a test suite, but this was not attempted due to time constraints and minimal areas where users can enter anything.

5.1.3 Graceful Error Handling

As mentioned previously (3.6), each endpoint only performs its intended purpose or fails. This allows the error handling to be done client-side to reduce unexpected behaviour and cleaner code. On the client-side, the user is directed to an error page if anything fails. Almost always, if the user retries their actions, it will work, so this error page has a redirect button back to the dashboard.

5.2 Other Considerations

5.2.1 Strategies

As shown in figure 4.14, a "get_balances" API and call was implemented to access the balances for all toggled accounts. A widget to display this information on the dashboard was also implemented such that it could easily be added. In the end, however, it was decided not to be included. Although it would have been helpful to the user to get a better overview of their finances and improve financial capability, it did not have an appropriate location to be put anywhere on the dashboard and would clutter up the UI. This demonstrates how the minimal design was not sacrificed for functionality in cases where the functionality would not greatly improve financial capability.

As well as a balances widget, other functionality was also considered to be included in the web application. Examples include adding an income tracker to calculate precisely how much money would be earned after tax and other loans, as this was found to be quite a prevalent feature of other financial tools.

In the aim to again have a minimal UI but also complete the project in time, the other strategies were prioritised over this due to their better effects in improving financial capability.

The budget strategy utilised the budget prediction in the end, but as outlined in the design chapter (3.4), another option that was considered was to utilise the research on the 50:30:20 budgeting solution. The idea was to have a way for the user to split their recent transactions into these three buckets to identify where money could be spent and saved. This would have been an excellent way to improve financial capability and was considered to be implemented; however, the budget prediction was a more exciting and challenging feature to implement and so was prioritised. Unlike the other widgets, this design could have been implemented on top of the budget prediction as part of the same strategy, but due to time constraints, this was not attempted.

5.2.2 Budget Prediction

The current solution for budget prediction is one model trained on the developer's financial data. This is acknowledged as not the best solution, but the system can be improved if more and various data sources can be accessed. An ideal design would have been implemented with two different neural networks.

The first would have been a macro model. This would have been trained on several, and ideally many, different users' data. It would aim to identify the macro trends in expenditure, for example, the cost of living crisis, and the patterns it identifies would be incorporated into the prediction.

On top of the macro model, there also would be several micro models, which are trained per user. When users sign up, they will be asked if they consent to a personalised model being trained on their data. If they accept, an independent micro model on the server would be created based on only their transactional behaviour.

The predictions made when utilising the this budget prediction method would incorporate the macro trends as well as the individual's micro patterns to give a more accurate prediction. This macro/micro model structure was inspired

by the research into time series prediction as some papers focus on macro prediction whereas others focus on micro prediction.

5.3 Project Management

The project utilised a hybrid approach of research methodologies in the first phase, the waterfall methodology in the second phase, a general agile methodology in the third phase, and an extreme-programming-like methodology in the final phase. The first three phases were intelligently chosen based on the work done in each phase to synergise effectively. However, the final phase was not really chosen, but rather a result of what happened, so in retrospect, a more detailed plan would have been helpful to give it more structure.

In addition, this four-phase plan was the second plan that was created. The first one, designed in the specification, was too ambitious and did not include certain aspects of the project that were later essential, such as budget prediction. This was a factor of not having a concrete plan of what the software should do, but also from lack of experience in project management for a software project. Now, with this experience, a better plan could be constructed. Furthermore, it is also clear that deciding precisely what the software should do is a crucial part that should come before the final timetable is created.

Chapter 6

Conclusions

6.1 Summary

Overall, the project was a success as the initial requirements were met and the title of this dissertation was successfully completed. A performant web application was made with minimalism and elegance in mind. The website features live transactional information in the form of strategies which aim to help increase financial capability. An overview of all areas of a user's finances will help inform their better decisions. The ability to toggle accounts on and off is a new feature not found in current personal finance tools. The proper Plaid authentication flow was followed with a clear separation between the frontend and backend. An accurate, by some measure, machine learning model was built to make expenditure predictions for all users. Finally, this is all incorporated, with a database, into a single deployable web application bundle.

6.2 Future Work

Follow-up work that could continue from the end of this project includes building further strategies. More strategies would mean more information is available for the user. There would still be an emphasis on avoiding complexity in an attempt not to confuse the user, so these would have to be carefully considered, but there is potential for more.

In addition to this, Plaid also offer a money transfer service between bank accounts. A potential feature that could be included in the application would be the ability to make transactions between linked accounts. For example, a user could discover that they have surplus money in their spending account, and then using the tool, they could transfer this money to their savings account.

Finally, after developing a web application, the next step is maintaining it. The application will need to be kept up to date with security measures, as it handles sensitive data, and fix any bugs that may arise. In addition, new and better libraries are often released, so there is always the potential to simplify and improve the technologies used in this application.

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`. 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] Group, Financial Advice Working. Consumer explanations of "advice" and "guidance", March 2017. URL `https://www.fca.org.uk/publication/research/fawg-consumer-explanations-advice-guidance.pdf`. Accessed: 18/04/2022.

[10] Hourston, Peter. Cost of living crisis. *Institue for Government*, Feb 2023. URL `https://www.instituteforgovernment.org.uk/explainer/cost-living-crisis`.

[11] 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.

[12] Intuit, . Quicken, 2022. URL `https://www.quicken.com/`. Accessed: 30/03/2022.

[13] Kvartalnyi, 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.

[14] Money, & Service, Pensions. Money and pensions service annual report and accounts for the year ended 31st march 2022, April 2022. URL `https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1114992/money-and-pensions-service-annual-report-and-accounts-2020-2021-print.pdf`. Accessed: 16/04/2022.

[15] 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.

[16] NFEC, . What is financial literacy?, 2022. URL `https://www.financialeducatorscouncil.org/financial-capability-definition/`. Accessed: 29/03/2022.

[17] OBIE, . What is open banking. *OpenBanking.org*, Sept 2020. URL `https://www.openbanking.org.uk/what-is-open-banking/`. Accessed: 29/03/2022.

[18] Plaid, . Plaidinc, April 2020. URL `https://www.youtube.com/@PlaidInc`. Accessed: 8/04/2022.

[19] Plaid, . Plaid api documentation, 2022. URL `https://plaid.com/docs/`. Accessed: 30/03/2023.

[20] Plaid, . Plaid institutions, 2023. URL `https://plaid.com/institutions/`. Accessed: 30/03/2022.

[21] PocketBase, . Pocketbase documentation, 2022. URL `https://pocketbase.io/docs`. Accessed: 2/04/2022.

[22] 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.

[23] Prep, Financial Modeling. Financial modeling prep, 2017. URL `https://site.financialmodelingprep.com/`. Accessed: 20/04/2022.

[24] 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.

[25] Storonsky, Nikolay & Yatsenko, Vlad. Revolut, July 2015. URL `https://www.revolut.com/`. Accessed: 30/03/2022.

[26] (Theo), T3 Open Source. T3 stack. *GitHub*, 2022. URL `https://create.t3.gg/`. Accessed: 2/04/2022.

[27] to Linux, Switched. Introduction to gnucash - free accounting software, May 2019. URL `https://www.youtube.com/watch?v=wBPg_AKdlG0`. Accessed: 30/03/2022.

[28] Vercel, . Next.js, 2016. URL `https://nextjs.org/`. Accessed: 2/04/2022.

[29] Vercel, . Next.js github, 2016. URL `https://github.com/vercel/next.js/`. Accessed: 2/04/2022.