Blockchain Payment App: Developing a payment app with blockchain and progressive web app technologies

Stephen Gordon
N00202054

Supervisor: Catherine Noonan

Second Reader: Joachim Pietsch

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

Declaration of Authorship

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Programme Chair.

WARNING: Take care when discarding program listings lest they be copied by some- one else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

Declaration

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

 Recoverable Signature

X Stephen Gordon

Stephen Gordon

Signed by: f4a75aca-955c-4007-ad62-fec694f580ca

Date: 03/05/2024

Failure to complete and submit this form may lead to an investigation into your work.

Contents

Abstract

This project aims to develop a payment app using blockchain and progressive web app (PWA) technologies to streamline transfers of USDC, a stablecoin cryptocurrency. The

research explores the significance of blockchain, blockchain wallets, and the potential of account abstraction to enhance user experience. It covers requirements gathering, design, implementation, and testing, detailing similar app studies and development sprints.

Similar applications are studied, and personas are created to gather the required functionality to build a payment app. Design choices are detailed through wireframes and user flow diagrams to ensure a smooth user experience. The implementation chapter describes the development process, frameworks and SDKs used and challenges of building the app.

The report outlines the research, requirements gathering, design, implementation, and testing phases. The app utilizes account abstraction for blockchain wallets, with a progressive web app architecture using React, Next.js, Redux, and an Express.js backend for push notifications. Native functionalities like the camera and push notifications are integrated, and thorough unit and user testing are conducted post-development to ensure functionality and usability.

Acknowledgements

I would like to thank my first supervisor Catherine Noonan for providing weekly feedback on the project, sharing ideas, and providing many useful resources when writing the thesis document. I would also like to thank Joachim Pietsch for his constructive criticism around the security and usability of the app.

List of Figures

List of Tables

Introduction

The inspiration for this thesis project came from my travels through several countries in Asia, where I observed that payment apps were widely used but inaccessible to foreign users due to the need for a local bank account for onramping. To address this challenge, I decided to create a payment app that is open and permissionless, built on blockchain technology.

The primary goal of this project is to explore and develop a payment application leveraging blockchain technology and progressive web app (PWA) technology. The project aims to create an easy-to-use payment app that allows users to store, transfer, and receive payments via a blockchain. In addition, this project will implement account abstraction to create a smart contract account for users, enhancing the user experience when interacting with the blockchain.

To achieve these goals, the project will use the Scrum methodology and will be broken down into several phases, including research, requirements gathering, design, implementation, and testing. Through these phases, the project will investigate various payment apps to identify the essential features required to create a user-friendly payment app.

One of the critical considerations for this project is ensuring the stability of the assets used for payment. To address this, the app will support the stablecoin USDC, which offers a secure and stable asset transfer method. As Liam Horne reports, stablecoins have even surpassed PayPal in total settlements in 2022, with over $11 trillion of volume settled on blockchains. This figure is almost the size of the entire Visa network, which is worth $11.6 trillion. This fact has further motivated the development of this payment app, which supports stablecoins such as USDC for seamless and secure transactions on the blockchain (Horne, 2024).

The desired outcome of this project is an app that provides a seamless payment experience to users. The project will leverage blockchain technology to create an app that is secure, permissionless, and accessible to all users. By developing a better understanding of blockchain technology and exploring the potential of stablecoins on blockchains, this project aims to contribute to the growing field of blockchain development and create a valuable tool for users.

Research

Introduction

This research will review literature on blockchain technology, wallets, and account abstraction to gain a better understanding of how blockchain wallets function. Currently, the global cryptocurrency industry is valued at more than 1.6 trillion, all of that value is secured by blockchain and accessed through wallets (Coinmarketcap, 2024). It will examine relevant research papers to understand how protocols such as Bitcoin and Ethereum use peer-to-peer technology, to come to consensus on the state of their blockchain in a decentralized way. It also explores how blockchain wallets are used to store and secure cryptocurrencies in a self-custodial manner. It will look at the types of blockchain wallets, from hot and cold wallets to smart contract wallets, their trade-offs, and how they are secured. It will investigate some common user experience problems users identified with wallets in the past and how these could potentially be fixed. It will explore account abstraction as a new technology and how smart contract wallets will open up more advanced features to blockchain wallets.

Blockchain

Blockchain Technology

In a paper published by Sarmah, they describe a blockchain as a decentralized ledger stored and maintained by a network of computers, known as nodes. Each node executes blockchain protocol software, responsible for validating batches of transactions, referred to as blocks. Every block contains a unique cryptographic hash of the block that came before it. The inclusion of this hash in each block ensures the integrity of the entire ledger, as altering any block would require changing every block before it. As each node checks the blockchain after processing each block, it only takes one honest node to detect and prevent unauthorized changes (Sarmah, 2018). The process of how nodes validate the state of the blockchain in a decentralized way is called the consensus mechanism. Different blockchains come to consensus by using different algorithms, Bitcoin uses proof of work, and Ethereum uses proof of stake (Buterin, Proof of Stake: The Making of Ethereum and the Philosophy of Blockchains, 2022).

Bitcoin was the first cryptocurrency to use blockchain technology to secure its ledger. Bitcoin is described by the creator Satoshi Nakamoto as: “A purely peer-to-peer version of electronic cash, that would allow online payments to be sent directly from one party to another without going through a financial institution.” (Nakamoto, 2008). In the Bitcoin whitepaper, Nakamoto outlines how the Bitcoin protocol will leverage the proof of work consensus mechanism to prevent the double spend problem. The double spend problem was a vulnerability in early digital currencies that would allow a malicious party to spend the same cash twice (Chohan, 2017). In proof of work, computers on the blockchain network known as miners compete to solve complex mathematical problems using their CPU, and in return receive cryptocurrency as a reward from the blockchain. As these miners spend lots of computational power, they are incentivized to not to attack the network (Buterin, Proof of Stake: The Making of Ethereum and the Philosophy of Blockchains, 2022).

While Bitcoin was the first cryptocurrency, it is limited to only supporting payments of the Bitcoin cryptocurrency from one user to another. The Ethereum protocol was proposed as an upgraded version of blockchains such as Bitcoin. Ethereum incorporates the idea of a Turing-complete programming language that can be used to create programmable decentralised applications on its blockchain, using smart contracts (Buterin, Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2014). In 1994, computer scientist Nick Szabo coined the term smart contract. He defines a smart contract as “A smart contract is a computerized transaction protocol that executes the terms of a contract” (Szabo, 2023). Smart contracts can be thought of as immutable programs on a blockchain where if certain conditions are met the program will execute (Chainlink, 2023).

Blockchain Wallets

How wallets work

Wallets are software applications that store private keys, which are used to sign transactions and prove ownership of cryptocurrency assets. When a user initiates a transaction, the

wallet creates a digital signature using the private key and broadcasts the transaction to the network. The transaction is then verified by other nodes in the network and added to the blockchain if it is valid (Houy, Schmid, & Bartel, 2023). The private key is typically a 12–24-word mnemonic phrase (Figure 1 Mnemonic seed phrase (Buterin, Vitalik, 2024)). By choosing a series of 12 random words from the Bip-39 word list (Palatinus, 2013), there are 2048^{12} possible combinations for the private key. This makes it practically impossible to brute-force guess a user's private key as the number of combinations is too large.

```
vote    dance   type    subject valley  fall    usage   silk
essay   lunch   endorse lunar   obvious race    ribbon  key
already arrow   enable  drama   keen    survey  lesson  cruel
```

Figure 1 Mnemonic seed phrase (Buterin, Vitalik, 2024)

There are two common types of wallets, hot wallets, and cold wallets. A hot wallet is a wallet that is connected to the internet. They are typically, on a desktop, web browser or mobile device. They are generally considered less secure as being connected to the internet opens up more attack vectors. A cold wallet, also known as a hardware wallet, is not connected to the internet. They are physical devices in the form of a USB, used to sign a transaction for the user. Hardware wallets are considered the safest type of wallet as the physical device is needed to sign a transaction (Biernacki & Plechawska-Wójcik, 2021).

Key management is very important for the security of your assets. While a mnemonic phrase is a secure way to protect cryptocurrencies, it also comes with unique problems. If a user's key is compromised, or a malicious actor was to gain access to a user's private key, they could potentially steal all cryptocurrency in that wallet. Timothy Peterson, a cryptocurrency analyst, claimed that 1500 bitcoins are lost every day, due to poor key management, and malicious attackers gaining access to a user's private key (Zimwara, 2020).

UX of Blockchain wallets

One common criticism of blockchain wallets is the 12-24 mnemonic phrase. In one study in 2021, a group of researchers analysed 45,821 reviews on mobile wallet apps, to study the UX of wallets. They found that a lack of account recovery was a big issue for users. Many users complained that if their phone broke or was lost and they didn't have a backup of their mnemonic phrase, all their cryptocurrency assets would be lost. This study also suggests several ways to improve blockchain wallets, suggesting wallets should try to mimic the user experience of already popular apps, such as social login, easier-to-understand terminology, and better security practices (Masoud , Wiese, Beznosov, Roth, & Voskobojnikov, 2021). Some solutions to this are multi-signature wallets such as Gnosis safe, a smart contract wallet with multiple private keys, it can be thought of as a group wallet. Currently Gnosis Safe is securing over 65 billion of cryptocurrency assets. Using a multi-signature wallet allows for account recovery services for its users. If a multi-signature wallet has 3 signers, and one of them loses their key, the other two signers can come together to recover their funds. A practical example is a user who keeps two of the private keys and gives one to a trusted third party. The third-party can never steal the assets, but if the user loses one key, both parties can come together to recover the assets in the wallet (Safe, 2024).

Account Abstraction as a UX Solution

Another solution for better wallet user experience is Account Abstraction, an Ethereum Improvement Proposal (EIP) proposed in 2021. On the Ethereum blockchain, there are two types of accounts. The default account is an externally owned account (EOA), a wallet controlled by a single private key. The second is a smart contract account, like a multi-signature wallet mentioned previously, which is controlled by smart contract code on the blockchain (Buterin, Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2014). The aim of account abstraction is to enhance the security and user experience of interacting with the Ethereum network. On Ethereum, a user's transaction must be initiated by an EOA. Account abstraction aims to enable smart contracts to initiate transactions themselves, reducing the need for EOAs. This will allow users to control their Ethereum account by using smart contracts.

As account abstraction wallets are contracts on the blockchain, they can be fully customized and programmable (Ethereum.org, 2023). Some features of an account abstraction wallet include account recovery where if a user loses their private key, it can be recovered. Batching user transactions so multiple blockchain transactions can be executed at once, improving user experience. For every transaction on the Ethereum blockchain a user must pay a fee in the native cryptocurrency Eth to the network to be executed, using account abstraction, a user's transaction fee can be sponsored by a third party or paid in another cryptocurrency, further simplifying the user experience. Account abstraction also brings security enhancements such as session keys, spending limits and customizable payment logic. Session keys allow for a user's wallet to have session expiry providing a similar experience to JSON Web Tokens (Wang & Chen, 2023). Security features such as spending limits can be coded into the contract. Users can set daily thresholds for how much value can be transferred from the account in a day/week/month. This means that if an attacker gains access to the account, they cannot steal everything at once, and the user has opportunities to freeze and reset access to the account. Users can also whitelist transactions to only addresses they trust, so if an attack accessed the user's private key, they wouldn't be able to transfer assets (Ethereum.org, 2023). As all these features are on the blockchain as immutable decentralized code, it is impossible to tamper with the user's account.

Conclusion

This research has looked at decentralized blockchain technology to gain a deep understanding of the wallets that are used to interact with and secure digital assets such as cryptocurrencies on the blockchain. For blockchains to function they use consensus algorithms such as proof of work or proof of stake to come to an agreement on the state of the blockchain, along with a set of peer-to-peer nodes to validate these transactions and store the state. This research also looked at how blockchains such as Ethereum can run decentralized programs called smart contracts that can execute code without the need for a third party. Using the knowledge from blockchain technology and smart contracts the research then further looked into the different implementations of blockchain wallets from externally owned accounts that use a mnemonic phrase as a private key, and smart contract accounts that can be used as wallets with greater functionality. Wallet security practices are examined, the types of wallets such as hot and cold wallets and the devices that the software is run on, and how important private key management is, as poor management can lead to the loss of all assets in the wallet. Furthermore, a new technology, account abstraction is examined on how it can improve the functionality and usability of smart contract wallets, allowing users to have social recovery, session keys, spending limits and other features, improving overall security and user experience of wallets.

Requirements

Introduction

This chapter describes the process of designing and developing the app's requirements. By studying similar applications, one can determine their advantages and disadvantages and the app's requirements.

Requirements gathering

This section aims to gather functional and non-functional requirements by studying similar applications. Furthermore, personas will be created to help identify potential users of the application and their reasons for using it. Similar applications and personas will be used to find the required and non-required functionality of the application and build a use case diagram that can be used as a reference when developing the application.

Similar Applications

Two similar applications, Revolut (Revolut, 2024), a financial application, and Uniswap Wallet (Uniswap, 2024), a blockchain wallet, will be examined to help understand what is required to build a payment application. Although both are fundamentally different, they have many overlapping components and functionality.

Revolut

Revolut is a financial technology company that offers a range of financial services such as sending money, investing, currency exchange, and many others. Revolut makes it easy for users to transfer money across borders, making it especially useful for remittance payments.

Once a user signs up to Revolut, they can add money to their account using various methods, such as bank transfer, Visa card, and Apple Pay. Every user has a unique Revolut username, which they can use to receive payments from other users.

The app has several different screens (Figure 2 Revolut research): a home screen where a user can see their account balance and recent payments and a transfer screen where users can see all their recent activity and payees. Clicking on a payee, the user can see all transfers and notes with that payee.

From the home screen, a user can add money, where they will be promoted to deposit money into their Revolut account via several payment methods. From the transfers screen, a user can choose to receive money via several different methods, either by their Revolut tag, a unique ID for each user, by QR code or phone number, or by copying a URL link that can be sent to a user.

When paying a user, the payee's name is shown, the amount to be sent is shown, and an optional note can be sent with the payment. Above the keypad are some predefined common transfer values, €10, €20, €50, €100, that can be sent.

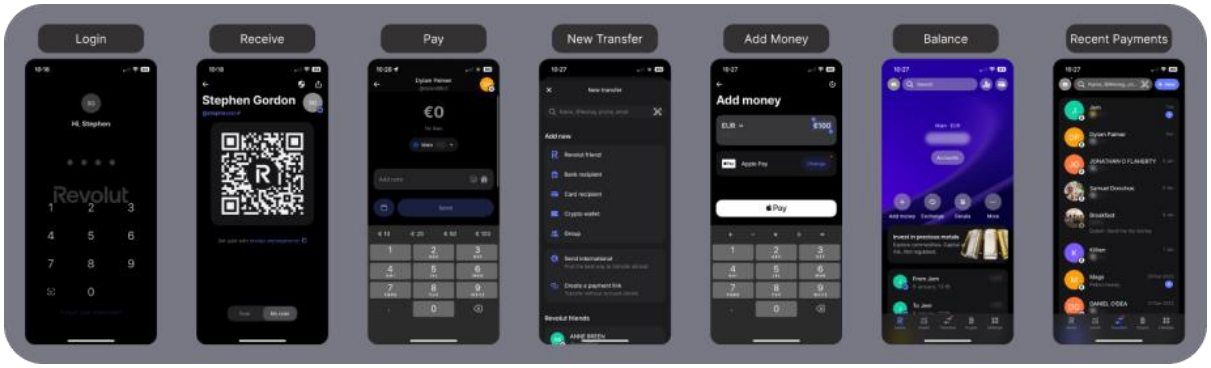


Figure 2 Revolut research

Advantages

- Easy to use.
- Lots of onramps are supported, including bank transfers, visa debit cards, and payroll.
- Free transfers

Disadvantages

- The signup process is long due to KYC (Know your customer).
- Users must rely on Revolut to secure and insure their money.

Uniswap Wallet

Uniswap Wallet (Figure 3 Uniswap Wallet research) is a self-custodial cryptocurrency wallet built for the Ethereum blockchain and any EVM (Ethereum Virtual Machine) compatible blockchains. Once a user downloads the app, they are prompted to create a new wallet or import an existing wallet using their mnemonic seed phrase. Uniswap Wallet provides a very easy-to-use interface for creating a new wallet. The new wallet's seed phrase is encrypted and backed up to the user's iCloud storage. This makes the onboarding for a new user very easy as they don't have to pay too much attention to storing their seed phrase.

Uniswap Wallet has functionality that is very similar to Revolut. Users can transfer cryptocurrency from their wallet to another user either by scanning a QR code or typing the user's blockchain wallet address or ENS (Ethereum Name Service), receive cryptocurrency via QR code, top up their wallet via a Visa debit card, see their recent activity and NFTs, and buy and sell cryptocurrency assets.

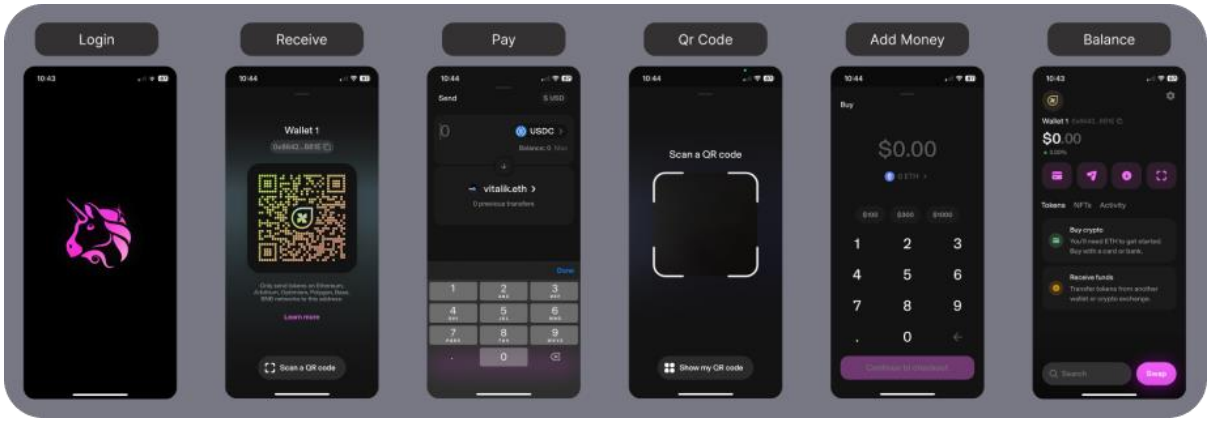


Figure 3 Uniswap Wallet research

Advantages

- Easy sign-up process.
- The account is secured by blockchain technology, which makes it very secure.
- A user can have multiple accounts within the app.

Disadvantages

- As the app is blockchain-based, the user experience and terminology can be unfamiliar to new users.
- Not easy to add money from a bank account to the wallet.

Requirements modelling

Personas

Persona 1

For the app, I designed two personas to understand better what users want from the application and how they intend to use it. The first persona (Figure 4 Persona 1) is a freelance software engineer who needs to transact with clients and contractors all around the globe. He needs to make cross-border payments in a frictionless, fast, and cheap manner.

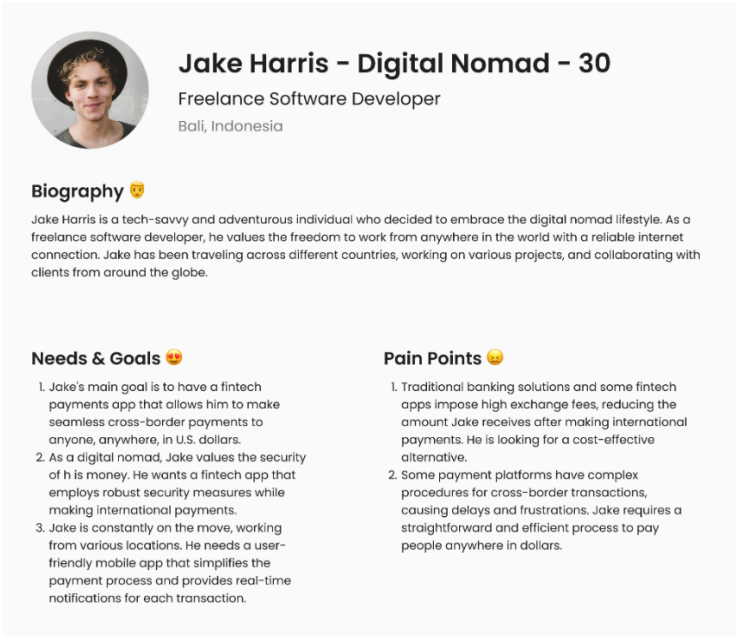


Figure 4 Persona 1

Persona 2

The Second persona (Figure 5 Persona 2) is a service industry employee. She needs to send remittance payments to her family in a reliable, secure way.

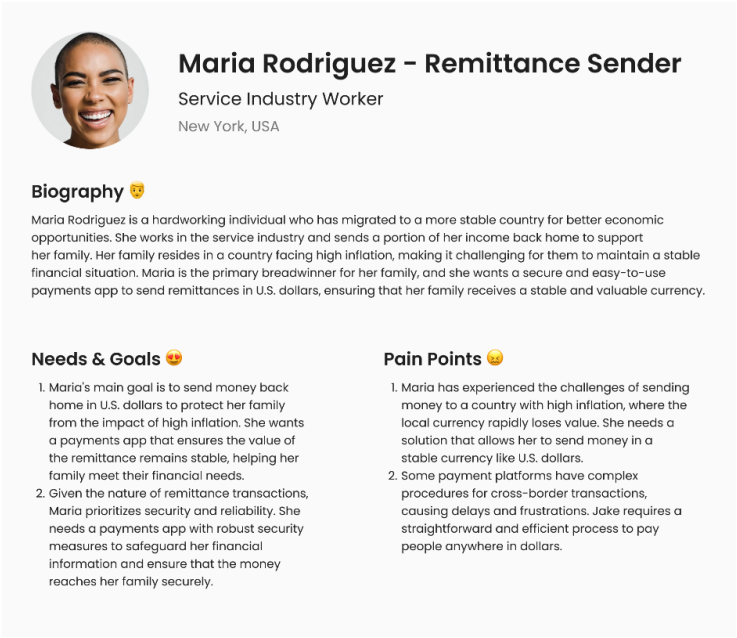


Figure 5 Persona 2

Interviews

Revolut

When choosing interviewees, it was required that they had the Revolut app already downloaded and had an account created, as the signup process can take days to complete.

Several interviews were conducted to gather information on why users of Revolut use the app, how they use it and interact with it, and if they are satisfied with the app. The interviewees were asked a series of questions while performing tasks. Some of the tasks included sending money to a user, checking their recent transactions, and receiving money from a user.

The first interviewee was a male in their 30s. Before starting the tasks, the users were asked what they used Revolut for. The interviewee said they mainly used Revolut to transfer money to their friends.

Task analysis

They were first asked to transfer money to a user. The goal of this task was to find out the usability and navigation of the app. After completion, they were asked if they had found it easy. At first, the user struggled to find the send button, not realising they had to navigate away from the home screen to another screen. However, after finding the transfers screen, the user found the process of sending money very easy. In the second task, the user was then asked to view his recent transactions to confirm the transaction went through. The user found it easy to find recent transactions. One pain point the user had was when receiving money from another user. The user said there were too many steps to find their unique Revolut ID or their QR code for the other user to scan.

Usability

The interviewee addressed some usability issues. When sending money to another user, the interviewee was afraid that they would type the wrong Revolut ID. The interviewee was asked what functionality they used in the app and whether they used the investing or lifestyle tab. The user said they rarely used these parts of the app, saying they mainly used the app for payments. A follow-up question was asked if they thought Revolut was trustworthy and secure. The interviewee said they have grown to trust it over time as the longer it's around, the more secure it makes it. However, they still trust their traditional Irish bank more.

All users expressed their enthusiasm for features such as sending and requesting money and highlighted the usability and easy-to-use interface. However, they pointed out that many features were not used, such as the lifestyle, crypto, currency exchange, and invest tab. Another pain point was that if a friend didn't have Revolut, they couldn't send money to them.

Uniswap Wallet

Several interviewees were asked to download and create an account with Uniswap Wallet. Users found the signup flow easy, however unfamiliar, as it contained blockchain-related terminology. Interviewees expressed that the signup process was much faster and simpler than Revolut as they didn't have to upload proof of ID and get verified. As Uniswap Wallet does not have the option to use test blockchains, a live blockchain had to be used for testing, which required real cryptocurrency assets. As testing required a live environment, users were not asked to send or receive money but instead to navigate through the app and were asked questions about the UI. The interviewees said that they liked the UI much more; however, it contained features that they were not interested in, such as the NFT (Non-Fungible Tokens) related section of the app.

Functional requirements

Table 1 Functional requirements table

#	Functional Requirement	Priority
1	As a user, I want to be able to sign up and log in to the application.	High
2	As a user, I want to be able to see my account balance	High
3	As a user, I want to be able to see my recent transactions	High
4	As a user, I want to be able to send and receive money	High
5	As a user, I want low transfer fees	High
9	As a user, I want to be able to receive push notifications	Medium
10	As a user, I want to be able to receive payments via QR code	Medium
11	As a user, I want to have a custom unique ID	Medium
12	As a user, I want to be able to recover my account if I lose access	Medium
13	An explanation of blockchain technology	Low
14	A tutorial on how to use the app	Low
15	An address book	Low
16	A user can set daily spending limits	Low
17	Implement session keys to enable session expiry	Low
18	The user can change blockchain networks	Low
19	An explanation of blockchain technology	Low
20	A tutorial on how to use the app	Low

Use Case Diagram

The application has two types of users: a User and a Recipient described in a use case diagram (Figure 6 Use case diagram). A user can sign up or log in using social accounts such as Google, Facebook, or GitHub. They can add money to their account. Once they have a balance, they can send money to another user using the user's unique ID, blockchain wallet address, or by scanning a QR code. A recipient can receive the money and check their updated balance. They can check their previous transactions and activity. They can view the hashed transaction on the blockchain to confirm the transfer.

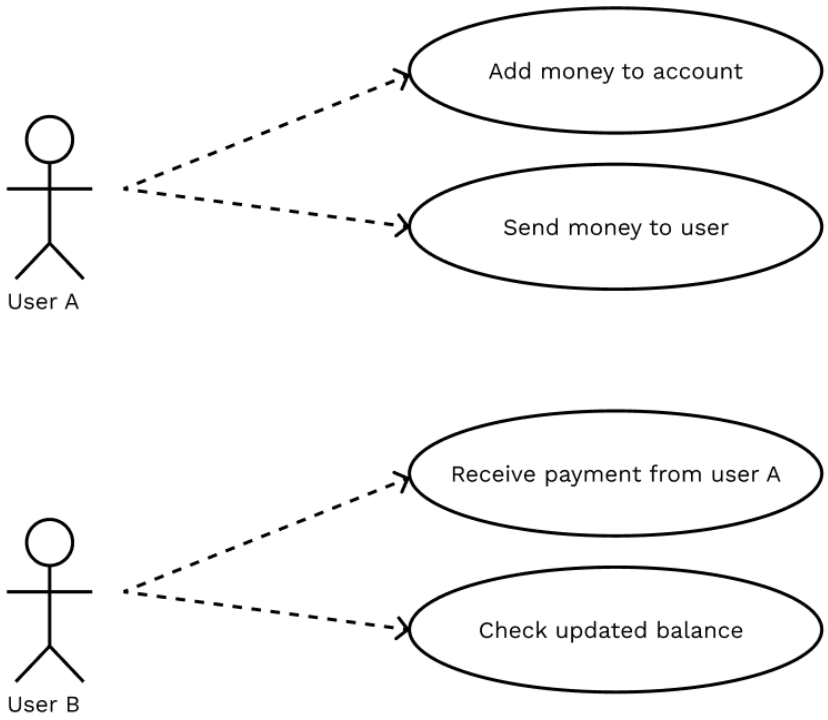


Figure 6 Use case diagram

Feasibility

The following are the technologies to be used in the development of the application:

- React
- Next JS
- ZeroDev SDK
- Wagmi React Hooks

React will be used for the frontend framework to build the user interface components and handle user interaction. Next JS will be used for the routing, building, and structuring of the application. The ZeroDev SDK (Software Development Kit) will be used to implement Account Abstraction, create the user's blockchain account and implement social login. Wagmi React Hooks will be used to interact with the blockchain.

This table (Table 3 Feasibility study table) outlines the possible challenges that may happen when developing the app.

Table 3 Feasibility study table

Challenge	Description	Solution

Many SDKs	There are many SDKs for implementing Account Abstraction.	Research must be done to see which SDK will be compatible with a NEXT JS PWA and has good documentation.
Inexperience with PWA development	As PWA development is new to me, research and examples must be explored. The PWA must have the required functionality, such as installation, notifications, and support for the Account Abstraction SDKs	Several examples and implementations of PWAs should be examined to determine if they support the required functionality.
Time	This project has strict deadlines for delivering each phase of the project.	Use project management tools to ensure all deliverables are delivered on time.
Lack of experience with new technologies	Many of the technologies in this project will be new to me, such as Next JS, Account Abstraction, Wagmi, and PWAs.	Following the technologies documentation and following the requirements document, it should be possible to implement these technologies.

Conclusion

In this section, both similar applications, Revolut and Uniswap Wallet, were examined. Their functionality, usability, advantages, and disadvantages were examined to understand better what is required to build a payments app. Two user personas were created to identify the reasons a user might use the application. After examining similar applications and using the personas, functional and non-function requirements were gathered so a model use case diagram could be designed.

Design

Introduction

This chapter outlines the application's design. Using the information gathered from the requirements section and by studying similar applications, the application’s technologies, structure, and architecture, along with its wireframes, prototype, and design system, will be developed. The design chapter will be used as a reference when coding the app going forward in the implementation chapter.

Program Design

The program design section aims to describe the application technologies and frameworks that will be used, along with the application's structure. By outlining the structure, architecture, and design, this section can then be used during implementation for further reference.

Why build an app on a blockchain

Blockchain technology was chosen to build the app on top, not just for its open source and community-driven nature but also for its developer-friendly features. This technology empowers developers by providing them with auditable and transparent tools, reliable and self-custodial systems, and globally accessible platforms, making them an integral part of the blockchain ecosystem.

Most blockchains are open-source and developed by community developers. This ecosystem has created a lot of free public infrastructure to help develop and build on top of blockchains. This developer ecosystem has created lots of free, open-source, well-documented packages and tools designed to interact and work with blockchains.

One of the key benefits of blockchains is their public nature, which ensures full transparency and auditability. This means that users can see exactly where their money is and what is happening in each transaction. This transparency eliminates the need for users to blindly trust that their money is being stored safely or that their transfer is being processed or completed, instilling a sense of trust and security.

Blockchain wallets provide a way for users to hold cryptocurrency assets in a self-custodial way, meaning that the developers of the app do not have to hold any of the users' money, and even if they wanted to, they can't as the user's private key is needed to access the wallet.

Unlike the US financial system, which often restricts non-US citizens, blockchains do not censor users based on location. An application built on a blockchain would allow users worldwide to participate, regardless of their origin.

Stablecoins, a type of cryptocurrency, allow blockchain users to hold a dollar equivalent. Stablecoins reduce users' exposure to the volatility of cryptocurrencies like Bitcoin. It also allows users in other countries access to the dollar, particularly in countries with high inflation.

Technologies

Frameworks such as React, Next, and Redux will be used to build the front end of the application, as well as the core components, routing, and state management. For the blockchain-based functionality of the application, toolings such as the ZeroDev SDK, Wagmi React Hooks, and Viem will be used to interact with the blockchain.

React (React, 2024) is a JavaScript library used to build interfaces based on reusable components. It will be used to build the front-end application, handle the functionality, and handle how the user interacts with it.

Next JS (Next, 2024) is a framework for building full-stack React applications. It will be used with React to provide a better development experience. Next JS will be responsible for building, hosting, and routing. Next JS also provides a built-in router named Next Router, allowing for clean navigation through the application.

Redux (Redux, 2024) is a state management tool used in React applications. It helps developers share and update state across components. Redux creates a global state object named the ‘store’. To change the state, Redux uses actions, dispatches, and reducers to update the application's state.

The ZeroDev SDK (ZeroDev, 2024) will be used to implement account abstraction. The SDK will be responsible for creating the user's blockchain account, sponsoring user blockchain transaction fees, and sending transactions to the blockchain network. The SDK can also be used to implement session keys, social login, and other advanced features in the application.

Wagmi React Hooks (Wagmi, 2024) is a useful Web3 library for React that provides hooks to interact with the Ethereum blockchain, abstracting away many low-level complex operations. To provide the application with blockchain functionality, Wagmi will be used in combination with Viem, another Web3 library, and the ZeroDev SDK.

In the beginning, React Native was chosen instead of React to develop the application so that the application could be supported natively on IOS and Android devices and have access to native device components. However, a React Native environment was not supported by a lot of SDKs. The main problem when developing with these SDKs was that they were not compatible with React Native as they needed a Node JS run time environment to execute along with differences in the supported APIs. Further, the SDKs that supported React Native had very little to no documentation, making it difficult to develop.

Several Account Abstraction SDKs, such as Alchemy SDK and Biconomy SDK, were examined before ZeroDev’s SDK was chosen. However, ZeroDev was chosen because it had very comprehensive documentation, an easy-to-use dashboard, and an active developer support channel. After cloning one of their test applications, they found they were the best SDK.

Structure

React applications are built around the concept of components. Breaking the app down into components allows for changing one section of the app without affecting others and allows for reusable and organised components. The application will have a dedicated folder for components. React can also have nested components, so it is good practice to organise the components into subfolders. Some useful components will be the ‘Login’ component, which will handle all the login functionality, including deploying the user's account to the blockchain

or a ‘Transfer’ component, which will render and handle all the functionality required to transfer payments from one user to another. Another common structure in React applications is a folder to store custom hooks. Any useful hooks created to simplify the codebase can be stored in a hook folder and can be reused across the application. For example, a hook could be created to see if the user is logged in and can be used anywhere in the application to show components conditionally based on whether they are logged in or not.

Next JS will be used as a framework for React to help with rendering bundling, routing and much more. Next, JS helps with the implementation of SSR (server-side rendering) components in React. SSR allows for components to be rendered on the server side rather than on the client side, creating a faster web application and better experience. However, a lot of the application will need to be client-side rendered as the device will need to interact with and make calls to the blockchain. Next JS will also be responsible for the application's navigation. Routes can be defined inside the pages folder. Each route will have its own folder with an “index.tsx” file. The route will automatically render the index page and use the folder name as the path name Figure 7 Next JS routing example). A “Link” component can be used to navigate between the pages. Next also allows for a “Layout” file where fixed components such as the navbar and footer can be rendered across the application.

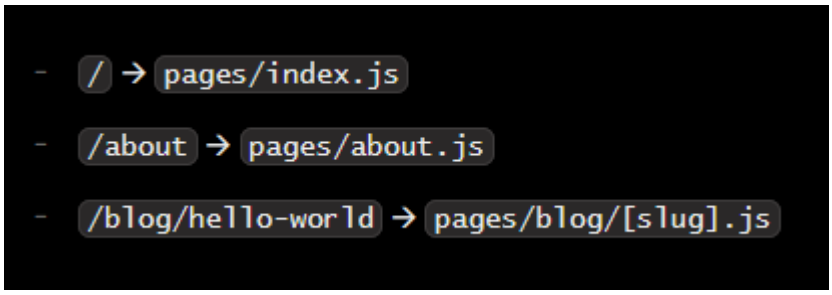


Figure 7 Next JS routing example

Redux is used for state management within the application. As the application will need to share data across the components, state management tools are necessary. The basic way of doing this is prop drilling and callback functions, where data is passed down as props from one component to another and updated with callback functions. This is not an ideal way for larger applications as it becomes difficult to track the flow of data and becomes hard to maintain. Another way to avoid prop drilling is by using the context API provided by React, which acts similar to Redux but is recommended only for smaller applications. Redux was chosen as it creates a centralised global state that can be debugged and managed with developer tooling, making the codebase cleaner and more reliable.

In the root of the application, all components will be wrapped in a context provider for Wagmi React Hooks, so the whole application has access to the hooks. These hooks are used in combination with React to read and write data to the blockchain.

Application architecture

An application architecture diagram (Figure 8 Application architecture diagram) was designed to help with the implementation phase. In this application, React and Next will be used to build the front end on the client side. The client will communicate with Redux, which will store the global state of the application. Wagmi, Viem, and ZeroDev will be used as the backend to connect to the blockchain. When a user submits a transaction to the blockchain, ZeroDev will relay the transaction on the user's behalf.

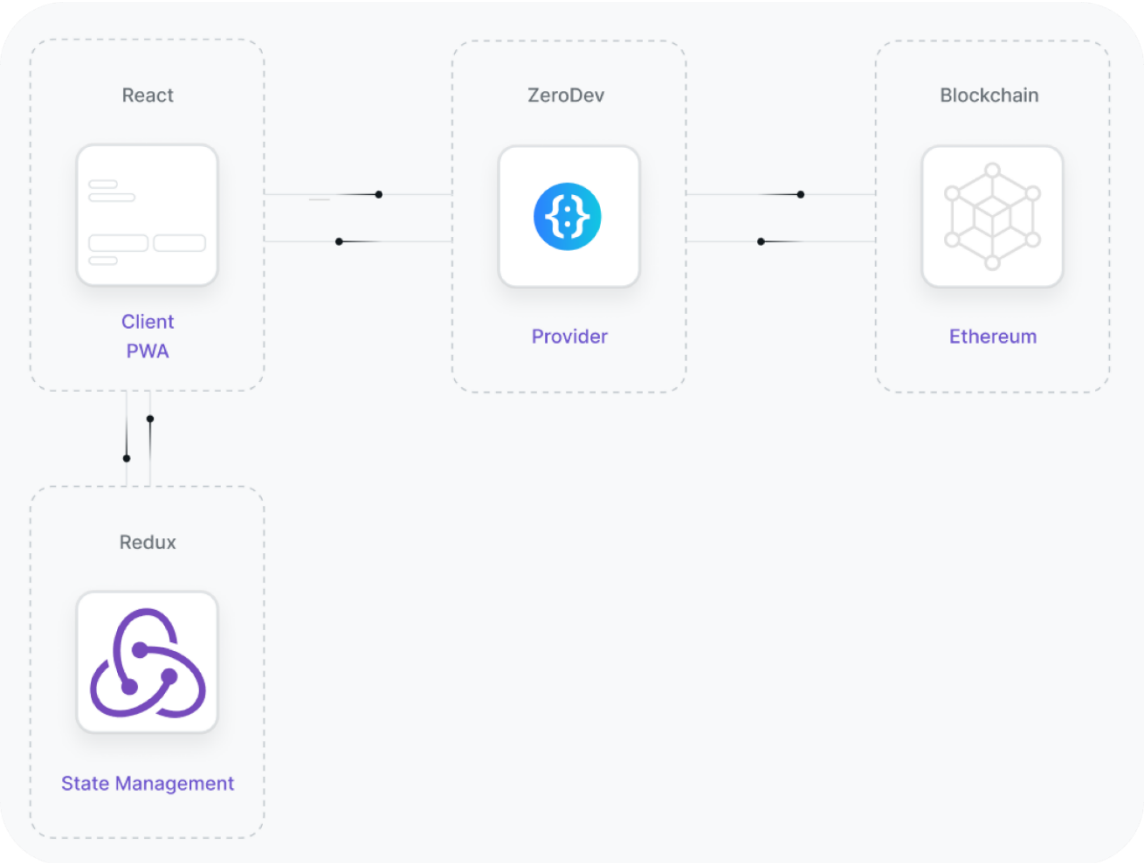


Figure 8 Application architecture diagram

Database design

Rather than using a traditional centralised database like MongoDB, all the user's data will be stored on the blockchain. Wagmi React Hooks, Viem, and Alchemy will query the user's data from the blockchain. These technologies will be used to get the user's account balance, ERC-20 balance, transaction history, and receipts. Wagmi provides a "useBalance" hook that can be used to get the balance of an account on the blockchain.

An example User schema (Figure 9 User Schema) will be used to develop the application and will be used as reference.

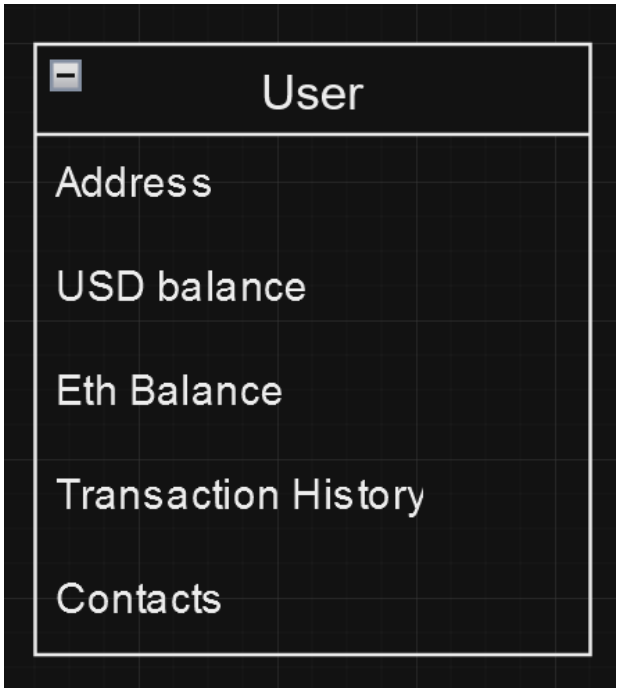


Figure 9 User Schema

User interface design

Wireframe

After exploring other similar applications and figuring out what functionality would be required, a basic paper wireframe prototype (Figure 10 Low fidelity wireframes) was hand-drawn.

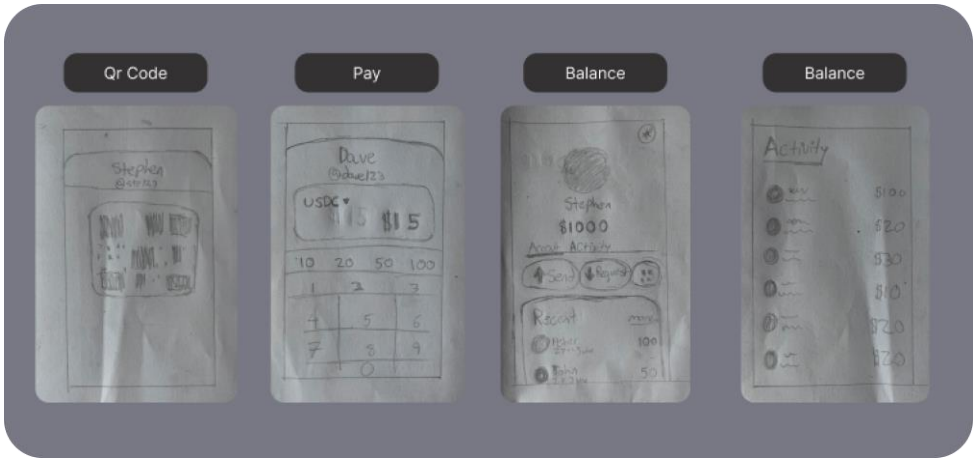


Figure 10 Low fidelity wireframes

Then, these prototypes were further developed in Figma, with the intent to build on them and work towards a medium (Figure 11 Medium fidelity wireframes) and high-fidelity (Figure 12 High fidelity wireframes) prototype.

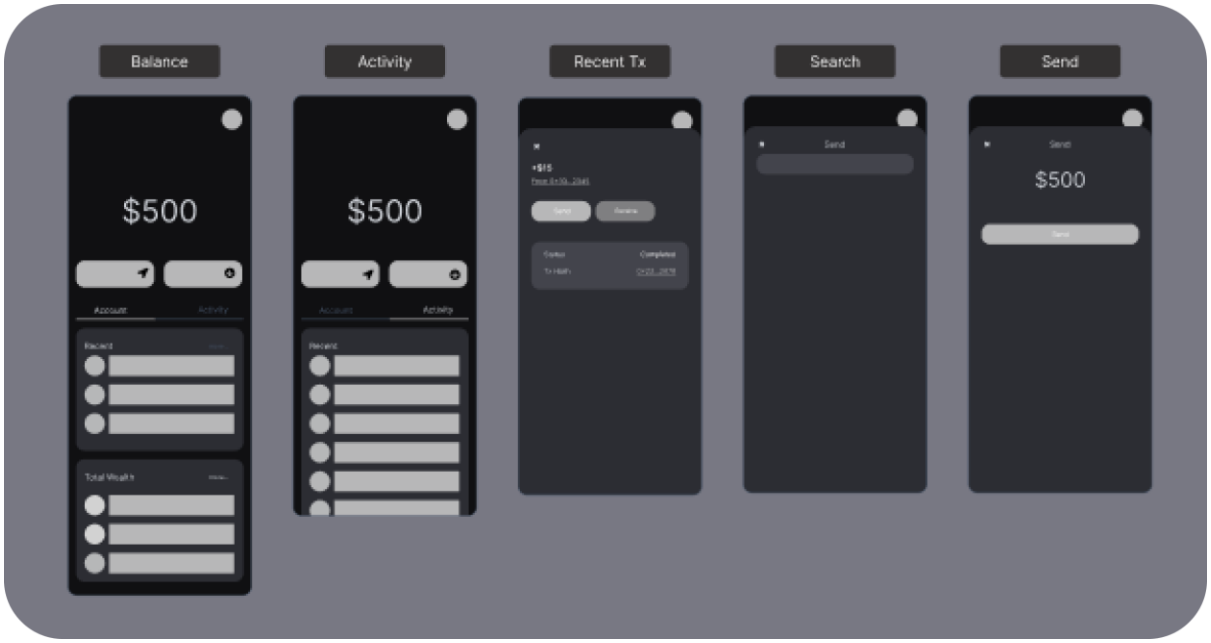


Figure 11 Medium fidelity wireframes

Figure 12 High fidelity wireframes

User flow diagram

A user flow diagram (Figure 13 User flow diagram) was designed to help better understand how the user will interact with the application step by step. This will help us design the user experience, reduce user error, and improve software development flow.

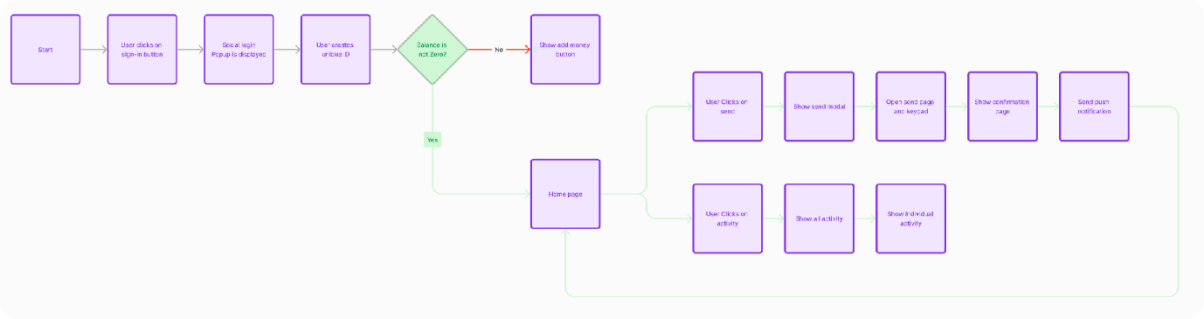


Figure 13 User flow diagram

Style guide

The application will use Tailwind CSS's design system (Tailwind, 2024). Tailwind is a CSS framework for React. It was chosen because it allows for customisation and can be used to build web applications quickly and easily. Tailwind also comes with a set of colours, icons, typography, and other useful features.

Type scale

The type scale (Figure 14 Type scale) design was taken from Tailwind CSS so that it can be replicated easily when coding the app.

Figure 14 Type scale

Font family

Nunito Sans (Figure 15 Font family) was chosen as the font. It is a sans serif font that is easy to read, open source, and widely supported by browsers. Other fonts looked at included Inter and Roboto.

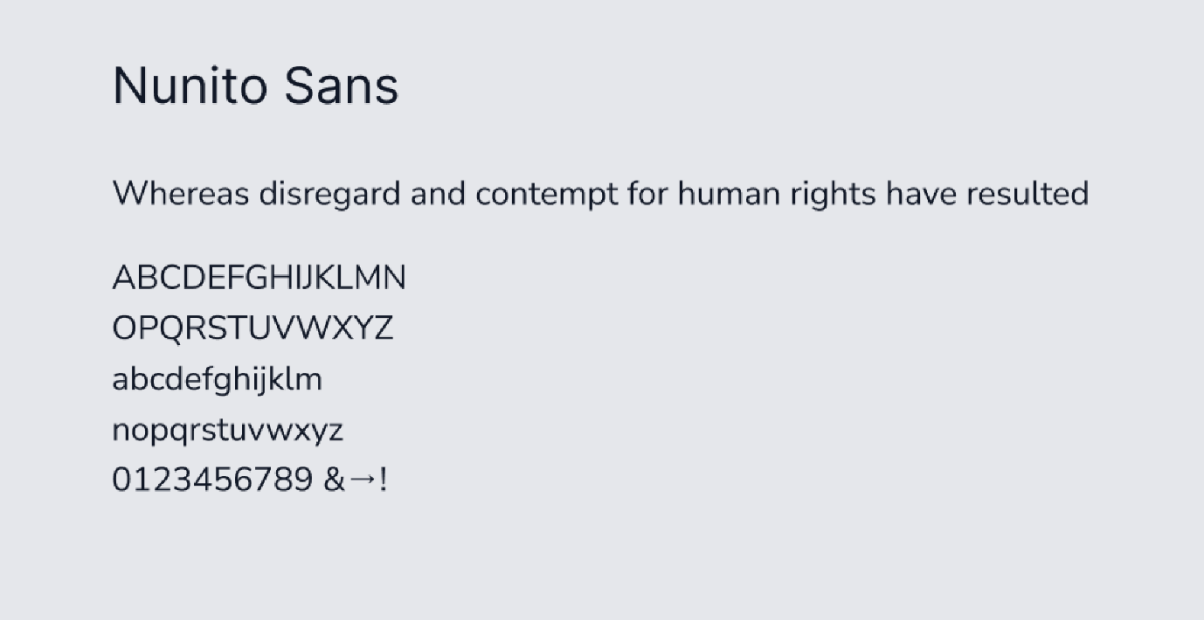


Figure 15 Font family

Colours

The colour palette (Figure 16 Colour palette) similar to the type scale was also taken from Tailwind CSS so that it can be replicated easily when coding the app.

Figure 16 Colour palette

A dark mode colour scheme (Figure 17 Text Colours) was chosen for this application. Dark mode has been a common trend in app design, gaining popularity because of its simplicity, aesthetic appeal, and reduced eye strain. As part of the dark mode colour scheme, black

colours will be used for the background, while grey and white will be used for the text. The main colours chosen are Purple for the primary colour; it will be used in titles, buttons, checkboxes, and any other important components in the application. Purple was chosen as it represents royalty luxury, and also trust and security. Purple can convey a sense of reliability, which is essential for a payment application. Along with purple as the main colour, a purple gradient will also be used to highlight certain elements while also providing more visual interest in the design. The Tailwind green, red, and yellow will be used for success, error, and warning colours.

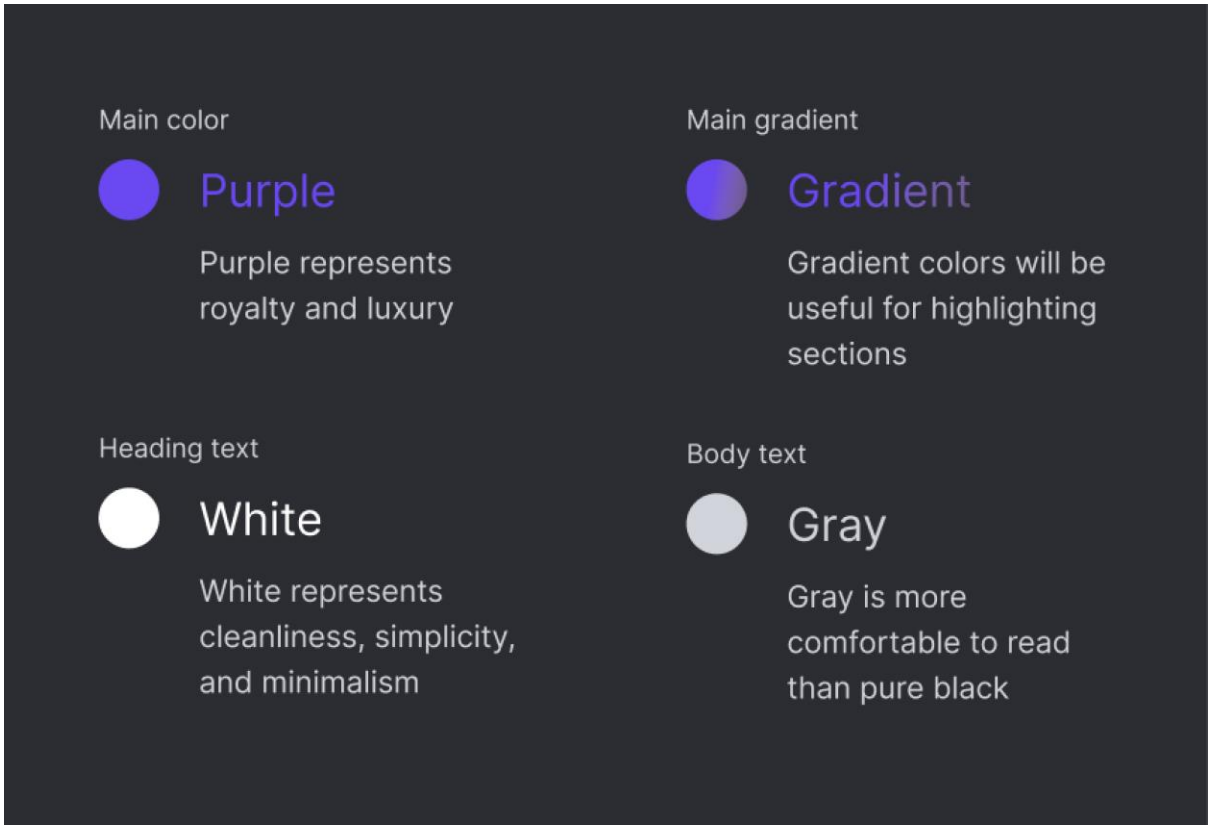


Figure 17 Text Colours

Conclusion

This chapter outlines the program and user interface designs. The program design outlines the technologies used, the structure and architecture of the application, and the database design. The program design is crucial to have as a reference when coding the app in the implementation phase. The user interface design will also be very helpful when coding the app and developing a UI library. As the design system was designed with the Tailwind CSS in mind, the design should be easier to implement in the app.

Implementation

Introduction

This chapter will describe the implementation and technologies used in the application.

- **React**
  React is a free and open-source front-end JavaScript library for building user interfaces based on components. React can be used to develop single-page, mobile, or server-rendered applications with frameworks like Next.js.
- **Next JS**
  Next.js is an open-source web development framework for building React-based web applications with server-side rendering and static website generation.
- **Redux**
  Redux is a state management library for JavaScript applications that provides a predictable way to manage application state and facilitate data flow across components.
- **Tailwind**
  Tailwind CSS is a utility-first CSS framework that provides pre-designed utility classes to quickly build responsive and customisable user interfaces with minimal CSS code.
- **Viem**
  Viem is a TypeScript interface for Ethereum that provides low-level stateless primitives for interacting with Ethereum. Viem is focused on developer experience, stability, bundle size, and performance.
- **ZeroDev SDK**
  ZeroDev is an SDK for developing account abstraction (AA) wallets and DApps using Kernel, a smart contract account.
- **Aceternity**
  Aceternity is a library of useful animated components that can be added to a project to save time when developing complex animations.
- **Privy**
  Privy is a library that handles the application's login and authentication flow. It works alongside ZeroDev to create users' accounts from social logins.
- **ShadCN**
  ShadCN is a set of reusable components that can be added to the project to develop a component library.
- **Web3Auth**
  Web3Auth is an SDK for creating blockchain wallets using social login.

- **Framer Motion**
  Framer Motion is a library that handles complex motion animations for React.

The application is a blockchain-based payment application that allows users to send and receive money.

Scrum Methodology

The scrum methodology is a popular agile methodology built around a cycle of repeating development periods called sprints. Sprints typically last one to four weeks. In each sprint, the team adds new functionality and improvements to the product. Although the scrum methodology is normally for teams, it can still be implemented in an individual project. With Scrum, solo developers can break their work into manageable chunks, setting achievable goals for each sprint.

In my case, I chose two-week development sprints during which specific app features were developed. At the beginning of each sprint, I planned out which functionality would be implemented. At the end of each sprint, my project supervisor and I reviewed the features.

Development environment

Visual Studio Code was used as the IDE to develop the application, as it has an extensive plugin ecosystem and syntax highlighting. Plugins such as Tailwind CSS IntelliSense and GitHub Copilot were used during development.

Git was used for project version control. GitHub was used to host the code repository.

Chrome, Firefox, and Brave were used to test the application's cross-browser support. Safari on iPhone was used to install the PWA on the user’s device.

Sprint 1

Goal

In this sprint, the goal was to gather basic requirements for the application. Once the basic requirements were gathered, low-fidelity and high-fidelity wireframes were created that could be developed into prototypes to serve as a reference when developing the app.

Low fidelity wireframes

Basic low-fidelity wireframes (Figure 10 Low fidelity wireframes) were hand-drawn on paper to understand the application's layout. Sketches were first used to generate possible layout ideas. These sketches were then further developed to create a basic wireframe in Figma.

Requirements gathering

Some similar applications were studied, and requirements for the application were gathered. These requirements were recorded and used to design a high-fidelity prototype in Figma.

High Fidelity prototype

The high-fidelity design was designed and prototyped in Figma (Figure 12 High fidelity wireframes). This prototype would then be used to create the application and serve as a reference.

Sprint 2

Goal

In this sprint, the goal was to research the feasibility of using React Native to build the app. After researching, the next goal was to set up the basic project structure so coding could begin as soon as possible. Another goal was to create an ERD describing how a user's account would be structured, an application architecture diagram, and a user flow diagram, all to be used as references when developing components in the app.

SDK Research

Before starting to code, knowing what SDK would be used to implement Account Abstraction was important. Several packages were examined, but ZeroDev was chosen as the best SDK. Several websites were used during the research, including the 'Awesome Account Abstraction' GitHub repository by '4337Mafia' (4337Mafia, 2024) and 'erc4337.io' (erc4337, 2024). I also joined a telegram group chat for developers working with Account Abstraction. In the group chat, I asked which SDK is best for implementing Account Abstraction with React Native (Figure 18 Telegram React Native Question). I received several replies from developers (Figure 19 First Telegram reply) (Figure 20 Second Telegram reply) with their suggestions of which SDKs to use. Initially, this was helpful as I knew which SDKs to further research.

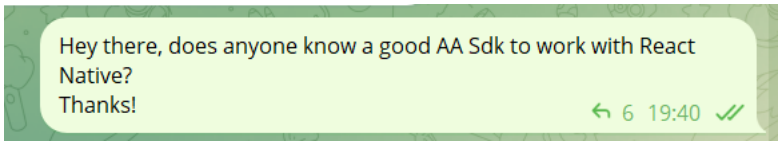


Figure 18 Telegram React Native Question

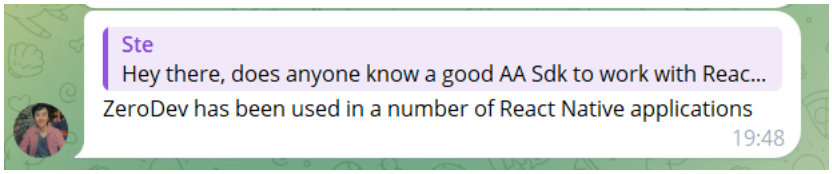


Figure 19 First Telegram reply

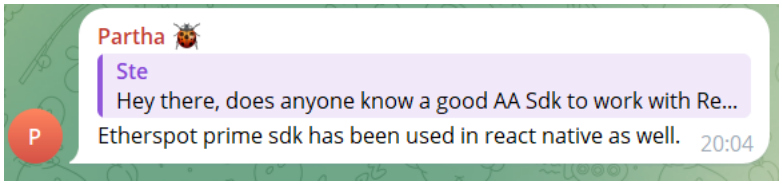


Figure 20 Second Telegram reply

React Native Research

It would have been nice to use React Native to build the app as it provides a more native experience for the user. Also, as a native app I would have access to more device APIs for example on iPhone, a React Native app can access iCloud storage, KeyChain, FaceID, haptic feedback, the native modal, and many other features.

After asking developers questions in the Telegram group chat, research was done to see if the application could be built with React Native and the suggested SDKs. Some SDKs suggested included ZeroDev SDK, Alchemy Account Kit, Biconomy SDK, and Etherspot SDK.

Recently, a new React Native workflow has come from a company named ‘Expo’, which increases developer experience by making the React Native workflow better and reducing build errors specific to each device. I tried to find an SDK compatible with React Native Expo but could not find one. I then tried to clone the recommended repositories, which were not developed with the intent to be run with Expo. Xcode on Mac and Android Studio on Windows emulators were used to run these repositories, but the codebase had too many errors and still didn’t work.

I then joined the company's Discord channels that made the SDKs to ask more questions (Figure 21 My question to SDK developers in Discord) about React Native support, but the developers did not reply. I shared a screenshot of a build error with their example repository when running on an Android Emulator.

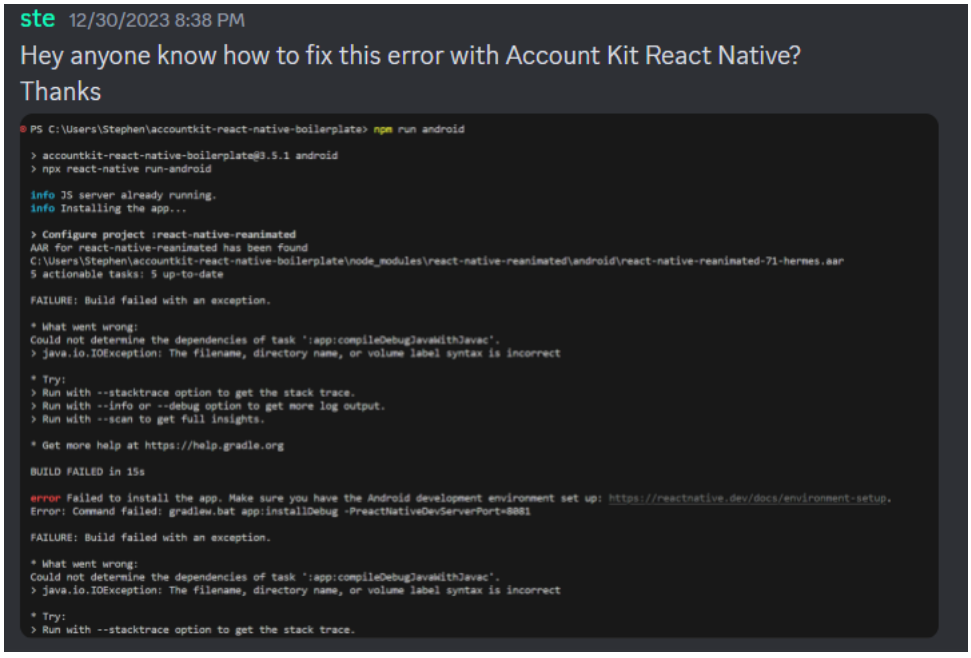


Figure 21 My question to SDK developers in Discord

However, I found a question from another SDK user (Figure 22 Developers response to another developer about the SDK not working) that suggested why the SDK didn’t work with React Native. The user also ran into problems when building a project with this SDK. The company's developer responded with a reason why it was not working as intended.

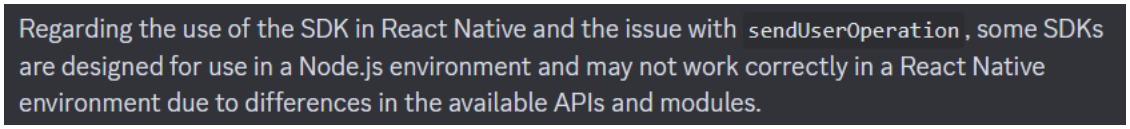


Figure 22 Developers response to another developer about the SDK not working

After some research and attempted implementations of each company's example React Native example repository, it was proven that the ZeroDev SDK, Alchemy Account Kit SDK, and Biconomy were not supported with React Native. This was because many of the Account Abstraction SDKs needed a node execution environment to run in. Also, as React Native is designed to work on multiple devices, it can cause many build errors and make development difficult; this is why an Expo implementation was initially desired.

Etherspot was not chosen because, after some research, I found that their SDK was too easy to use, and I would not be learning any new technologies. Etherspot made their SDK so that all of the account setup and blockchain interaction was abstracted away from developers. The Etherspot SDK would be helpful if minimal coding and learning were desired for the project. However, I thought I would learn more about Account Abstraction and building blockchain-based applications by choosing a different SDK and approach.

As React Native could not be used, Next JS and React were chosen to build a PWA to provide a similar user experience. Since a PWA is run inside a browser, the Account Abstraction SDKs should work as intended, as the browser is a node execution environment.

UI library

A basic UI library was designed in Figma, which would then be implemented in the app. This library included a typography system, colour palette, and choice of font family. Several design systems, such as Airbnb (Airbnb, 2024) and Uber (Uber, 2024), were studied. This design library was then used in my high-fidelity prototype.

Diagrams

The ERD was developed using ‘diagrams.net’ (Diagrams, 2024), and the user flow and application architecture diagrams were designed in Figma. These were developed by examining similar applications and their structures and tailoring them to this project.

Project setup

I began by cloning a template Next JS app (theodorusclarence, 2024) by ‘theodorusclarence’ on GitHub with several preinstalled packages, including Tailwind CSS and Jest, for testing. The template also had the basic folder structure and routing setup, including essential components such as a ‘page-not-found’ and ’error’. It was chosen to clone a template as it saves development time, ensures the proper installation of packages, and correctly sets up the folder structure. As my high-fidelity was designed using the Tailwind CSS design system, it was helpful to have it preinstalled and configured in the template.

Sprint 3

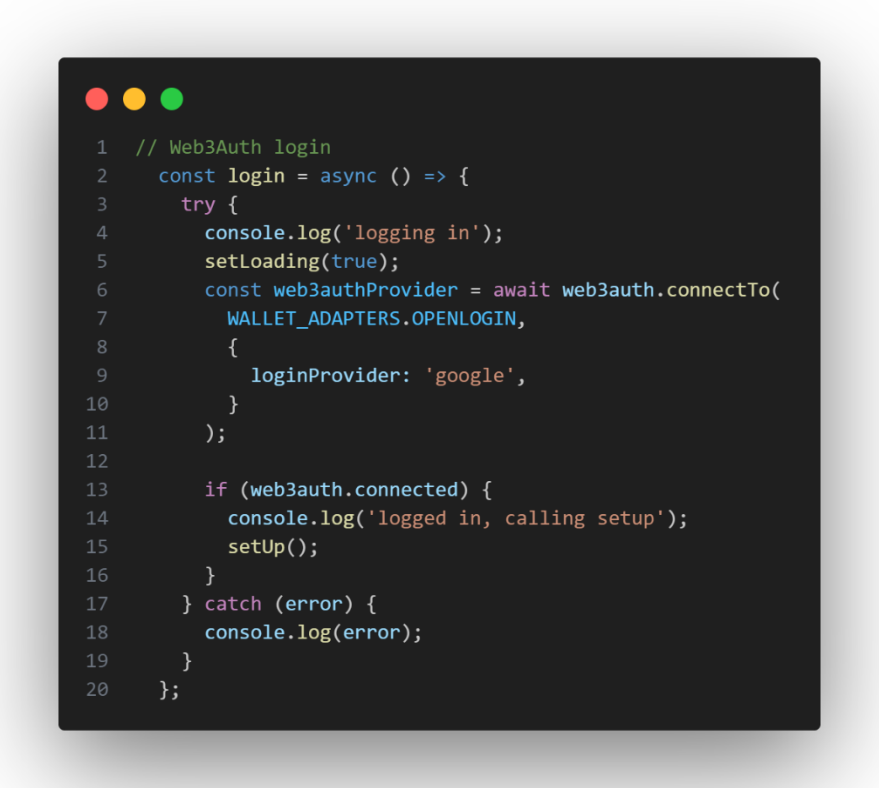
Goal

In this sprint, the goal was to set up basic login functionality, create a Redux store, implement the payment functionality, and get a user's balance. By the end of the sprint, the aim was to be able to have a user log in, their details stored in a Redux store, and let the user have the ability to send money on the blockchain and then see their updated balance in the UI.

Basic login functionality

To implement basic social login functionality and ensure blockchain account creation worked correctly, I installed the ZeroDev SDK and Web3auth SDK and added any necessary API keys to my environment.

The login code (Figure 23 Web3Auth login function) was implemented first by calling the login function, which utilises the Web3auth SDK. The user can sign in using a Google account by passing 'google' as a parameter to the SDK. This creates a signer object, a basic EOA (externally owned account) wallet for the user.



```
// Web3Auth login
  const login = async () => {
    try {
      console.log('logging in');
      setLoading(true);
      const web3authProvider = await web3auth.connectTo(
        WALLET_ADAPTERS.OPENLOGIN,
        {
          loginProvider: 'google',
        }
      );

      if (web3auth.connected) {
        console.log('logged in, calling setup');
        setUp();
      }
    } catch (error) {
      console.log(error);
    }
  };
```

Figure 23 Web3Auth login function

This signer can then be used with ZeroDev's SDK to create the user's blockchain account, known as the 'Kernal'. Once the Web3auth login function is complete, the setup function is called. The 'setUp' function (Figure 24 ZeroDev setup function) calls a custom React hook that can be called to create the account. Once the Kernal setup is complete, the user is brought to the home page using the Next JS router.

```
// ZeroDev SDK setup
  const setUp = async () => {
    try {
      if (web3auth) {
        const kernal = await useCreateKernal(web3auth);
        setKernal(kernal);
        if (kernal.account) {
          setLoading(false);
          setLoginSuccess(true);
          setReduxKernal();
          setTimeout(() => {
            router.push('/home');
          }, 2000);
        }
        console.log('My account:', kernal.account.address);
      }
    } catch (error) {
      console.log(error);
    }
  };
```

Figure 24 ZeroDev setup function

In the custom React hook that creates the Kernal (Figure 25 Setup custom react hook), the Web3auth provider object is passed into the hook. The provider's private key is then used to create the Kernal to deploy the user’s account to the blockchain. Creating a custom hook makes the code much cleaner by separating the functionality.

```
'use client';
import { IProvider } from '@web3auth/base';

// rpc
import RPC from '../web3RPC';

// Zero Dev
import { createEcdsaKernelAccountClient } from '@zerodev/presets/zerodev';
import { sepolia } from 'viem/chains';
import { Hex } from 'viem';
import { privateKeyToAccount } from 'viem/accounts';

import { useDispatch } from 'react-redux';

const useCreateKernal = async (web3auth) => {
  // take in the web3auth provider as a param

  // create a new RPC instance
  const rpc = new RPC(web3auth.provider as IProvider);

  // get the private key
  const privateKey = await rpc.getPrivateKey();

  // create a signer from the private key
  const signer = privateKeyToAccount(`0x${privateKey}` as Hex);

  // create the Account Abstraction Kernal Client
  const kernelClient = await createEcdsaKernelAccountClient({
    // required
    chain: sepolia,
    projectId: process.env.ZERODEV_ID,
    signer: signer,
  });

  // return the kernal client
  return kernelClient;
};

export default useCreateKernal;

```

Figure 25 Setup custom react hook

Create a Persistent Redux Store

The Redux store is used to store state across the application's components. For example, it would be helpful to store the user's account balance, address, authentication state, recent transactions, and other helpful information. Redux holds all this information in a store that can be read and updated using the functions 'useSelector' and 'useDispatch'. A package named Redux Persist was chosen to implement the Redux store. Redux persist stores the Redux store in the device's Local Storage, so if the user refreshes or closes the application or has no internet connection, the application's previous state can be retrieved from storage. Each item in the redux store is known as a slice. The entire app is wrapped in a redux provider in my root Next JS layout file (Figure 26 Root layout wrapped in the Redux Provider). This gives every route and component access to the Redux store.

```
<body>
        <Providers>
          <PersistGate loading={null} persistor={persistor}>
            <WagmiProvider config={config!}>
              <QueryClientProvider client={queryClient}>
                <LayoutGroup>
                  <div>{auth}</div>

                  <main
                    style={{
                      backgroundColor: 'rgba(16, 16, 18, 1)',
                    }}
                    className=' h-screen w-screen text-gray-300'
                  >
                    {children}
                  </main>
                </LayoutGroup>
              </QueryClientProvider>
            </WagmiProvider>
          </PersistGate>
        </Providers>
      </body>
```

Figure 26 Root layout wrapped in the Redux Provider

Each Redux Slice was a TypeScript file located in the Redux features folder (Figure 27 Redux features folder).

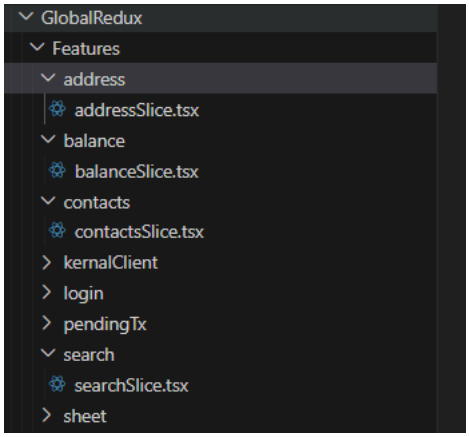


Figure 27 Redux features folder

Inside a Redux slice file, the initial value of the item is defined. In the example below (Figure 28 Address slice), the initial value of the user's address is an empty string as it has not been set yet. Then the addressSlice is defined, passing in its initial state, and the reducers, which are responsible for updating the state. The setAddress reducer is defined, which, when called with the useDispatch hook, the value passed in will set the value in the Redux store.

```
'use client';

import { createSlice, PayloadAction } from '@reduxjs/toolkit';

interface AddressState {
  value: string | undefined;
}

const initialState: AddressState = {
  value: '',
};

const addressSlice = createSlice({
  name: 'address',
  initialState,
  reducers: {
    setAddress: (state, action: PayloadAction<string>) => {
      state.value = action.payload;
    },
  },
});

export const { setAddress } = addressSlice.actions;
export default addressSlice.reducer;

```

Figure 28 Address slice

In my Redux store file (Figure 29 Redux store configuration), where the Redux Store is defined and set up, I passed in a configuration to store the Redux global state in the device's local storage. Then, inside the Redux store file, all the reduces for each slice are imported.

```
const persistConfig = {
  key: 'root',
  storage,
};

// Combine reducers with RootState type
const rootReducer = combineReducers<any>({
  login: loginReducer,
  address: addressReducer,
  search: searchReducer,
  kernalClient: kernalClientSliceReducer,
  transactions: transactionsSliceReducer,
  sheet: sheetSliceReducer,
  balance: balanceSliceReducer,
  pendingTx: pendingTxSliceReducer,
  contacts: contactsSliceReducer,
});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({
  reducer: persistedReducer,
});

export const persistor = persistStore(store);

export type AppDispatch = typeof store.dispatch;

```

Figure 29 Redux store configuration

Implement Payments

Once a user is logged in, one of the app's core functions is to send money. To implement this, a function named ‘sendTx’ (Figure 30 ‘sendTx’ function) was created and is responsible for all the payment logic. The function is called when a button is clicked. The function uses the Viem SDK to interact with smart contracts on the blockchain. The function requires an ABI (Application Binary Interface), a function to call, in this case, the ‘transfer’ function, and two arguments, the payee and the amount of money to transfer. The Kernal object created earlier has a method named ‘sendTransaction’, which will take the encoded function data and the token address and then send the transaction to the blockchain.

Once the transfer is complete, the user is routed back to a page where they can see the transaction receipt.

```
const sendTx = async () => {
    try {
      // Encode the data with Viem Function
      // Requires the abi of the contract, the function name, and the arguments address and amount
      // @ts-ignore
      const encoded: any = encodeFunctionData({
        // @ts-ignore
        abi: erc20Abi,
        functionName: 'transfer',
        args: [payee as `0x${string}`, parseUnits(usdcAmount, 6)],
      });
      console.log('Sending USDC');
      setLoading(true);
      const txnHash = await kernal.sendTransaction({
        to: usdc, // ERC20 address
        value: BigInt(0), // default to 0
        data: encoded,
      });

      console.log('Txn hash:', txnHash);

      if (txnHash) {
        setLoading(false);
        setTransactionStatus(true);
        setTimeout(() => {
          dispatch(setSheet(false));
          router.push(`/transaction?hash=${txnHash}`);
        }, 3000);
      }
    } catch (error) {
      console.log(error);
    }
  };
```

Figure 30 ‘sendTx’ function

Get a user’s balance

A ‘Balance’ component (Figure 31 Balance component) was created to get a user's balance, which would handle all the logic involved. First, the application was wrapped in a context provider for ‘Wagmi React Hooks’. Wagmi provides useful hooks for react apps, in this case I use the ‘useBalance’ hook. The hook takes in two parameters: the user's address and the token you want the balance of. The hook returns the user's balance, which is then returned in a div to be displayed in the application. This component could be turned into a reusable hook, returning the data rather than a div.

```
// wagmi
'use client';
import { useBalance } from 'wagmi';

// Redux
import { RootState } from '../../../GlobalRedux/store';
import { useSelector } from 'react-redux';

export default function Balance() {
  const addressState = useSelector((state: RootState) => state.address.value);

  const result = useBalance({
    address: addressState,
    token: '0x94a9D9AC8a22534E3FaCa9F4e7F2E2cf85d5E4C8',
  });

  return <div className='text-white'>{result?.data?.formatted}</div>;
}

```

Figure 31 Balance component

Sprint 4

Goal

In this sprint, the goal was to set up the application's routing, turn the web app into a progressive web app, implement searching for a payee address to send money to, allow a user to generate a QR code based on their address, and allow a user to scan another users QR code.

Routes

The app has been designed to have several routes, such as login, home, search, and send pages. To make a route in Next JS, a folder is created and named the route's name, and inside the folder is a file called index.tsx, which exports a ‘Page’. So, when a user goes to ‘/home’, Next JS will render the ‘index.tsx’ file in the ‘home’ folder. Some of these pages were designed to be modals and implemented as a parallel route in Next JS. A parallel route allows for two views to be rendered simultaneously. For example, in this app, the search page is rendered in a modal that appears over the homepage. So, when the user goes to the ‘/search’ route, it is rendered on top of the home route. Parallel routes were nonfunctional requirements for the app but provided a better experience and a feature often seen in similar applications.

A route group is created to implement parallel routes by putting an ‘@’ symbol in front of the folder. Then, in the route group, parallel routes are made by placing a ‘(.)’ in front of the

folder name. Next JS uses this convention to mark the route to be intercepted by the router and lets Next JS know that the route should be rendered in parallel. The parallel route folder structure can be seen below (Figure 32 Parallel route folder structure).

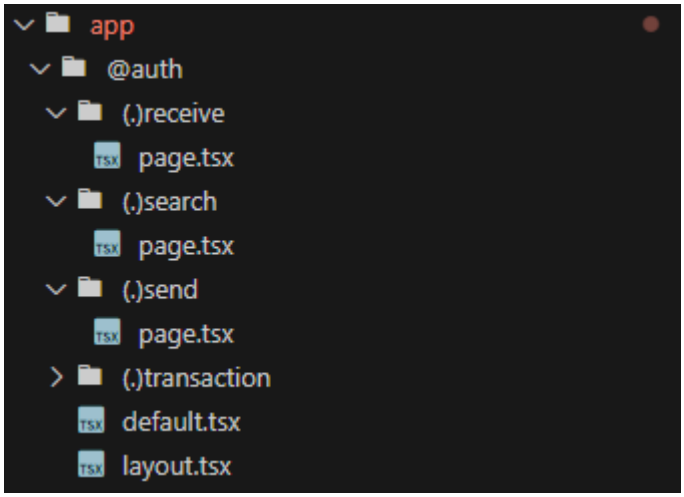


Figure 32 Parallel route folder structure

When creating the modal routes, I reviewed several options to make a modal. I thought about creating my own, using the new HTML Dialog (Mozilla, 2024) element, and creating custom CSS styles for it. I also considered using Headless UI (HeadlessUI, 2024), a component library for react, which would handle the opening and closing modal functionality. I also looked at ‘React Aria’ (Adobe, 2024) a component library developed by Adobe, focusing on accessibility.

I decided to go with an open-source package named react-modal-sheet. This package has several modals ready to use and already styled, saving time when designing the modal. The package uses framer motion to animate the modal in and out of the user's view.

To implement ‘react-modal-sheet’ (Temzasse, 2024) in my project, I created a component named ‘SheetLayout’. Inside SheetLayout, the react-sheet-modal components from the package are imported and rendered. The Sheet component then renders React children

inside the modal. Then, in my Next JS layout.tsx file (Figure 33 Layout.tsx), I render the SheetLayout.

```tsx
'use client';
import SheetLayout from '@/app/components/Layouts/SheetLayout';
import { Metadata } from 'next';
import * as React from 'react';

export default function ComponentsLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <>
      <SheetLayout>{children}</SheetLayout>
    </>
  );
}

```

Figure 33 Layout.tsx

In Next JS, when a layout file is in a folder, the layout is applied to all routes in the same folder. The structure for using a layout file can be seen below.

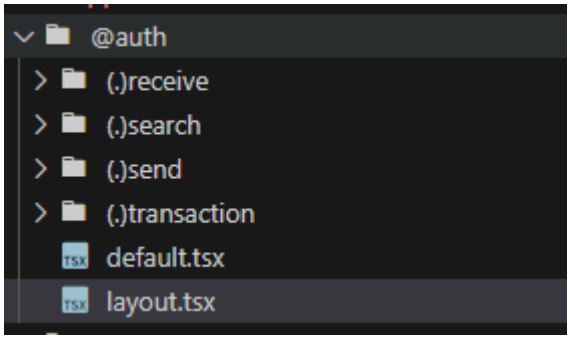


Figure 34 Layout structure

The ‘SheetLayout’ (Figure 35 SheetLayout component) component is conditionally rendered by a prop named ‘isOpen’. The isOpen prop is a boolean variable stored in my Redux store. This allows the sheet to be opened or closed from anywhere in the app.

```
'use client';

import { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import Sheet, { SheetRef } from 'react-modal-sheet';
import { setSheet } from '@/GlobalRedux/Features/sheet/sheetSlice';

const SheetLayout = ({ children }) => {
  const isOpen = useSelector((state) => state.sheet.value);

  useEffect(() => {
    console.log('Sheet is open:', isOpen);
  }, [isOpen]);

  const dispatch = useDispatch();

  return (
    <>
      <Sheet isOpen={isOpen} onClose={() => dispatch(setSheet(false))}>
        <Sheet.Container
          className='blurios'
          style={{
            backgroundColor: 'rgba(29, 31, 39, 0.8)',
            backdropFilter: 'blur(20px) saturate(100%)',
          }}
        >
          {/*  <Sheet.Header></Sheet.Header> */}
          <Sheet.Content>{children}</Sheet.Content>
        </Sheet.Container>
        <Sheet.Backdrop onTap={() => dispatch(setSheet(false))} />
      </Sheet>
    </>
  );
};

export default SheetLayout;
```

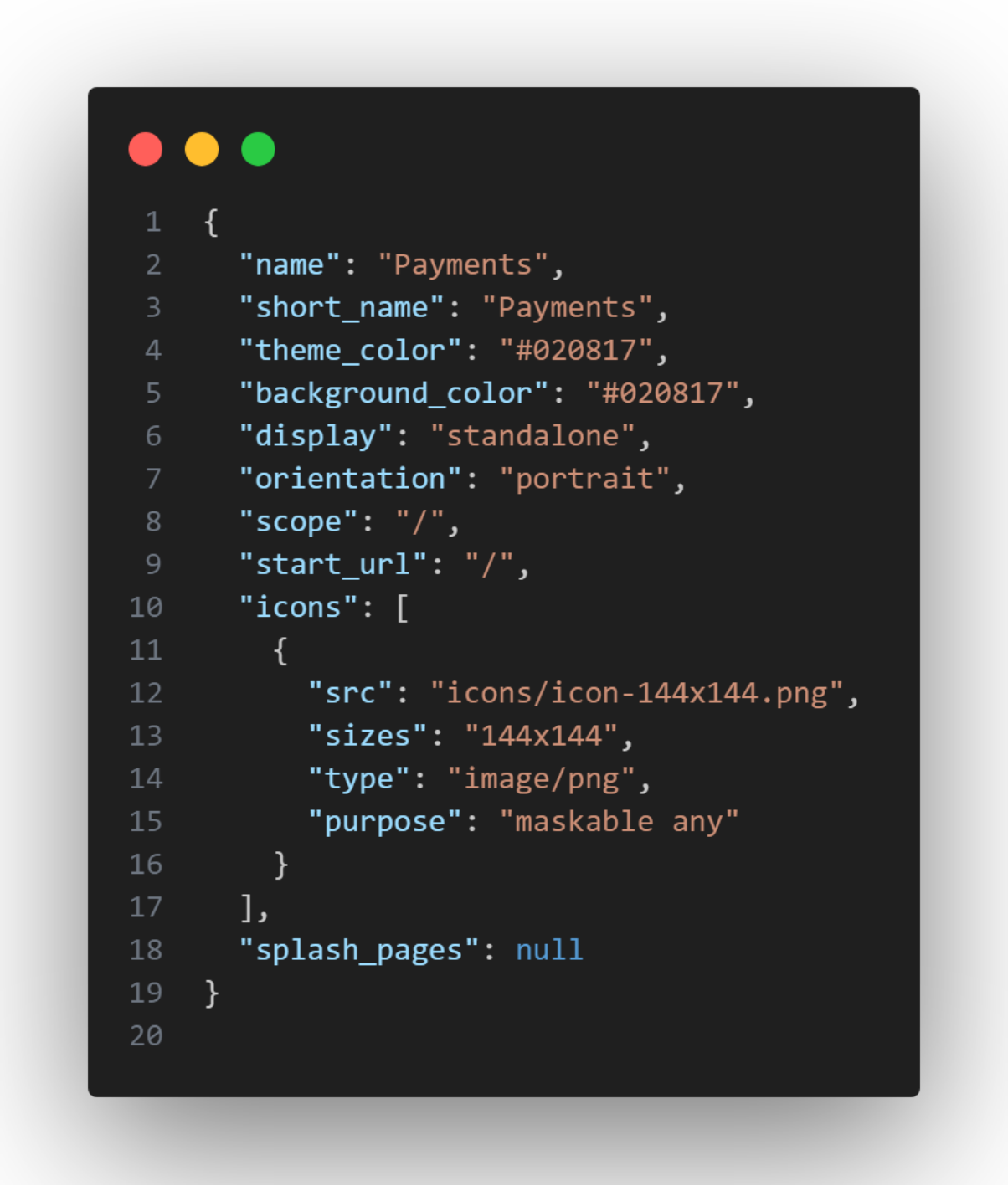
Figure 35 SheetLayout component

Implementing Progressive Web App Features

To turn my web app into a progressive web app, I first used a package named ‘next-pwa’ by ‘shadowwalker’ (Shadowwalker, 2024). This was chosen as it had the most stars on GitHub and lots of resources on how to implement it. However, after implementing it, I realised that the package had stopped being maintained, and support for multiple service workers, which I would later need for push notifications, was not supported. After some research, I found an

up-to-date fork of the package named ‘serwist’ by ‘serwist’ (Serwist, 2024), which had forked the ‘next-pwa’ repository and added onto it. This package was better as it was still maintained and had more examples, including an example of how to implement push notifications.

To implement the PWA functionality in my app, I first created a ‘manifest.json’ file (Figure 36 manifest.json file) that contained information about the app, such as the title, background colour and link to the app's icon. Then, I altered my Next JS next.config.js file (Figure 37 Next config file) to be wrapped in the ‘withSerwist’ method. The config file is responsible for providing a path to the service worker. When the app is built, using npm run build, the service worker is created in the public folder.



```
{
  "name": "Payments",
  "short_name": "Payments",
  "theme_color": "#020817",
  "background_color": "#020817",
  "display": "standalone",
  "orientation": "portrait",
  "scope": "/",
  "start_url": "/",
  "icons": [
    {
      "src": "icons/icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png",
      "purpose": "maskable any"
    }
  ],
  "splash_pages": null
}
```

Figure 36 manifest.json file

Then, to ensure that Next JS builds the app using the PWA features, the Next JS config file (Figure 37 Next config file) is wrapped in a ‘withSerwist’ function, which marks the path to the app's service workers.

```
// @ts-check
import withSerwistInit from '@serwist/next';

const withSerwist = withSerwistInit({
  cacheOnFrontEndNav: true,
  swSrc: 'src/app/sw.ts', // Path to your service worker file, default: 'src/app/sw.js
  swDest: 'public/sw.js',
});

/** @type {import("next").NextConfig} */
const nextConfig = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
  eslint: {
    // Warning: This allows production builds to successfully complete even if
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
};

export default withSerwist(nextConfig);

```

Figure 37 Next config file

When building the app, the process failed for a reason I couldn't figure out. After a lot of debugging, it turned out that if the path to the app icon in the manifest.json file was wrong, the app's PWA features would be broken. I found this bug by running a lighthouse report in my Chrome browser, leading me to the problem's root.

To confirm that the PWA is working, an icon should appear on the right of the search bar (Figure 38 PWA installation icon in search bar). Clicking this icon will allow you to install the application on your device.

Figure 38 PWA installation icon in search bar

Implement Search Address

To implement search, a payee React state variable (Figure 39 Payee state variable) is declared in the search page. The payee is an Ethereum blockchain address, which can be typed into an HTML input element.

Figure 39 Payee state variable

The input element (Figure 40 Search address input) will listen for a change in the input and set the payee to whatever the user types.

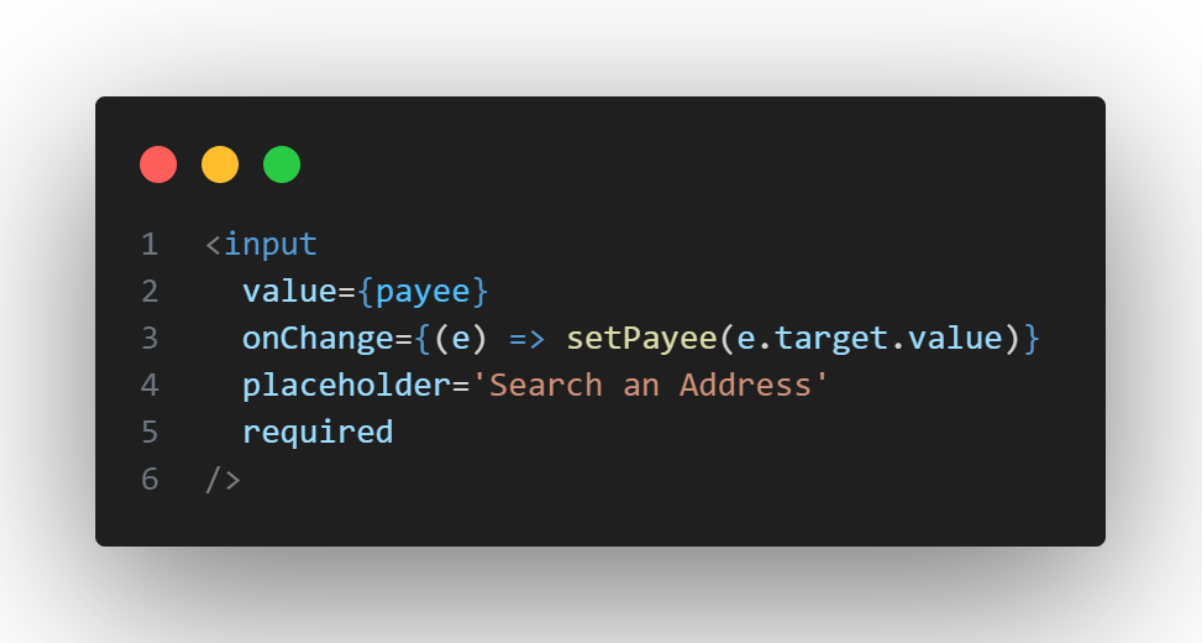
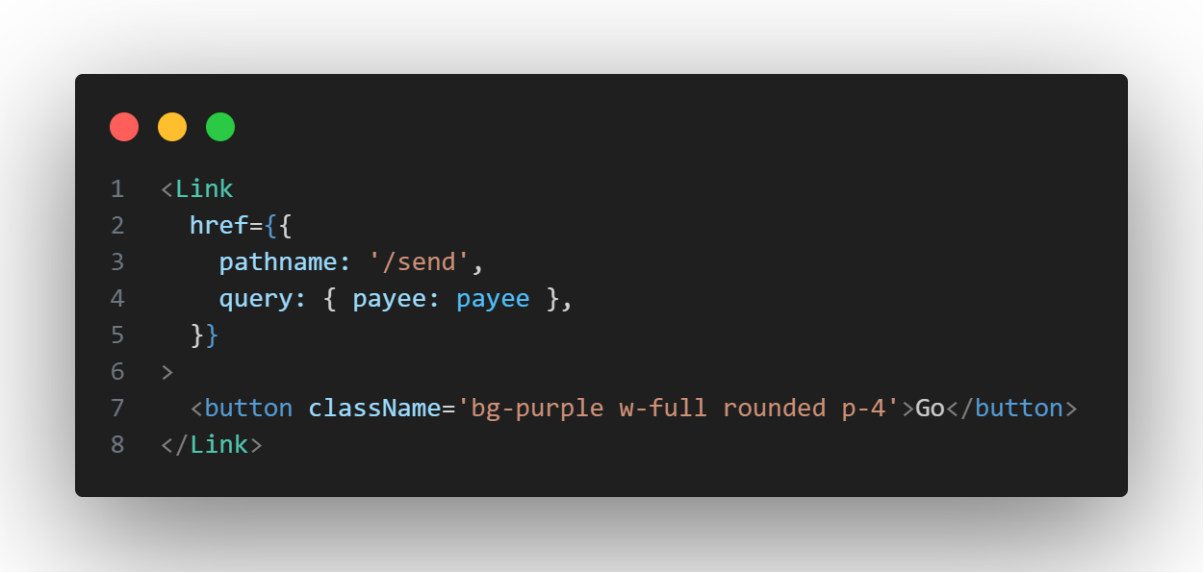


Figure 40 Search address input

Once the user has input an address to send to, they can then be routed to the send page, where the payee address is passed as a query parameter (Figure 41 Link to send page) and can be retrieved in the send page using Next Navigation.



```
<Link
  href={{
    pathname: '/send',
    query: { payee: payee },
  }}
>
  <button className='bg-purple w-full rounded p-4'>Go</button>
</Link>
```

Figure 41 Link to send page

QR Code Generator

To create a QR code in React, several packages were examined, such as ‘react-qr-code’, ‘react-qrcode-logo’, and ‘node-qrcode’. After some research and trying out different packages, ‘react-qrcode-logo’ (react-qrcode-logo, 2024) was found to be the most customisable and best fit for the application. Unlike the other packages, it allowed customisation of the background image, size, colour, and other styles of the QR code.

To implement the QR code, a custom QR component Figure 42 QR Code component was created to handle all the QR code-generating logic. The QR code is imported from the ‘react-qrcode-logo’ package. The imported component takes in several optional props, which are stated in its documentation. Some props I used were the qrStyle to choose the shape of the QR code, bgColor for the background, fgColor for the foreground colour, and the value, which is the user's address.

```
<QRCode
  ecLevel='H'
  quietZone={20}
  size={windowWidth}
  bgColor='#0c0e1800'
  fgColor='#a7a3f5'
  qrStyle='dots'
  eyeRadius={8}
  //pink, purple, yellow
  eyeColor={['#ec91d8', '#ec91d8b3', '#ec91d866 ']}
  value={`${address}`}
/>
```

Figure 42 QR Code component

One problem with the QR code component was the size prop. It only allowed a pixel value instead of a percentage value, so making the QR code responsive for every device was hard. If the size was hard coded at a size of 300 pixels, on devices smaller than 300px, parts of the QR code would be hidden off the screen. I implemented a function (Figure 43 QR Code Functionality) to calculate the device's screen width to fix this. When the component mounts, a ‘useEffect’ function called the ‘handleResize’ function is run, which uses the window object to get the screen's width. I then removed 10% of the screen width to add padding to the QR code for better UX. This function could be turned into a react hook if the screen width needs to be calculated later in the project.

```
export default function Qr() {
  const address: string = useSelector(
    (state: RootState) => state.address.value
  );

  const [windowWidth, setWindowWidth] = useState(0);

  useEffect(() => {
    function handleResize() {
      const screenWidth = window.innerWidth; // Get the screen width
      const tenPercentOfScreenWidth = screenWidth * 0.1; // Calculate 10% of the screen width
      const screenWidthMinusTenPercent = screenWidth - tenPercentOfScreenWidth; // Subtract 10% from the screen width
      setWindowWidth(screenWidthMinusTenPercent); // Set the state with the modified screen width
    }

    // Initial width on component mount
    handleResize();

    // Event listener for window resize
    window.addEventListener('resize', handleResize);

    // remove the event listener when the component unmounts
    return () => window.removeEventListener('resize', handleResize);
  }, []);
```

Figure 43 QR Code Functionality

A little later I then realised that there was a better way to implement this code, by using a React hook rather than using the window object. I created a React ref named ‘widthRef’ (Figure 44 Width ref), which refers to the parent div containing the QR code. Then any time the screen width changed, a use effect hook would run to recalculate the width of the div. The width of the div was then passed into the QR code to control its width.

```
const [windowWidth, setWindowWidth] = useState(0);

const widthRef = useRef<HTMLDivElement>(null);

useEffect(() => {
  if (widthRef.current) {
    const width = widthRef.current.offsetWidth;
    console.log('width', width);
    setWindowWidth(width);
  }
}, []);

return (
  <div ref={widthRef}>
```

Figure 44 Width ref

QR Code Scanner

The QR code scanner is implemented using a package named ‘yudiel/react-qr-scanner’ (react-qr-scanner, 2024). A package called ‘html5-qrcode’ was also researched, but its implementation did not work.

The QR code scanner was hard to implement as Apple IOS does not allow applications to access the device’s camera unless the web app is hosted on localhost or a secure HTTPS domain. As the app was being developed on a local LAN network, I couldn’t access the camera during development. So, every time the scanner was tested, the application had to be deployed to Vercel and reinstalled on the device. This slowed down testing as redeployment takes several minutes, and if there are any bugs, it can break the application's build process. During testing, an uncommon bug (mebjas, 2024) in iOS was found, where the camera doesn’t work, and the device has to be restarted. I researched this bug on StackOverflow, GitHub, and Reddit and found several other developers facing this issue. Unfortunately, this bug is an iOS issue; nothing can be done in the application to fix this issue.

The ‘QrScanner’ component (Figure 45 QR Scanner component) is imported from the package and takes in two props: an ‘onDecode’ function and an ‘onError’ function. When the camera scans a QR code, the ‘onDecode’ function is run. The result parameter is the address of the payee's scanned QR code. The result is then used to route the user to the send page, and the payee's address is input into the payment component.

```
<QrScanner
  onDecode={(result) => {
    setIsOpen(false);
    router.push(`/send?payee=${result}`);
  }}
  onError={(error) => console.log(error?.message)}
/>
```

Figure 45 QR Scanner component

Sprint 5

Goal

In this sprint, the goal was to set up push notifications to a user's device, change the login provider to use Privy instead of Web3Auth, change all previous functionality to use Privy and clean up the ‘Send’ page UI by adding a Keypad.

Push Notifications

Although push notifications were not required for the application to work, I knew it would be a good challenge, and I would learn more new technologies as I had never worked with notifications before. One prerequisite for notifications on iOS devices was that the Safari browser doesn’t support notifications by default. To enable notifications on a user’s device, the user must go into the Safari settings and enable notifications in the ‘experimental features’ section. This is not ideal for the UX of the app for iOS users. This step is not required for Android and PC devices.

To implement push notification, there are two main pieces of functionality: an API I developed that users can subscribe to notifications and a webhook that listens to users who subscribed to notifications and alerts my notification API when the user receives money.

I had a look at several providers for handling subscriptions and sending notifications. The first was a decentralised push notification service named ‘Push Protocol’. They provide a way to stream and subscribe to push notifications on several blockchains. Another service was ‘Pusher’, a JavaScript-based SDK for handling push notifications. However, I chose not to go with these services and developed my own as they provided unnecessary functionality for my application, which just needed basic receiving notifications functionality. Another consideration was how I would integrate these push notifications SDKs with the backend webhooks that listen for transactions.

I first set up the webhook using the ‘Alchemy Webhook SDK’. Alchemy is responsible for listening to users' transactions when they send and receive money on the blockchain. Alchemy only listens for transactions associated with your Alchemy project. I created a project with Alchemy and received my API key. I could then manually add blockchain addresses through their dashboard (Figure 46 Alchemy Webhook dashboard) or an API.

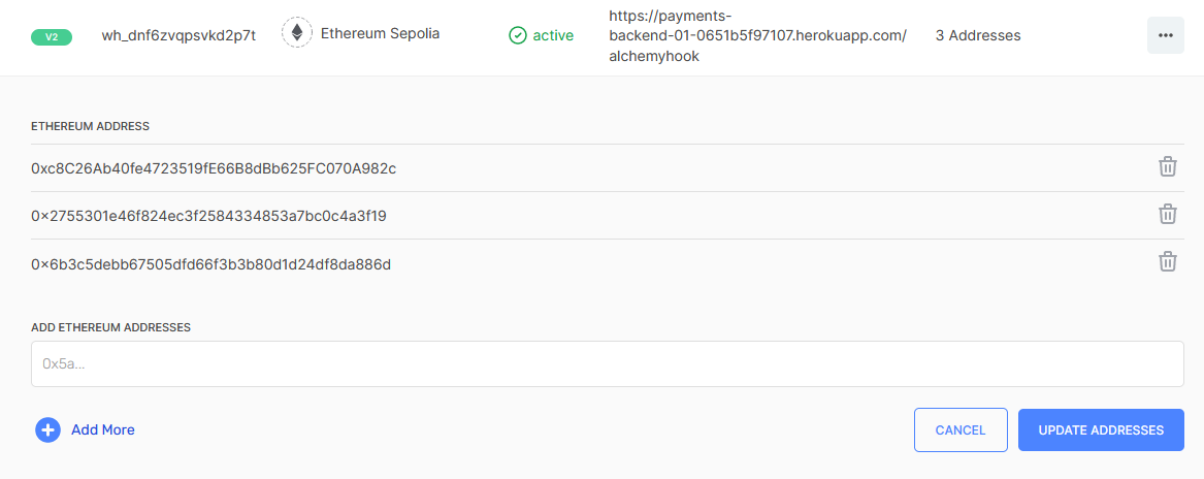
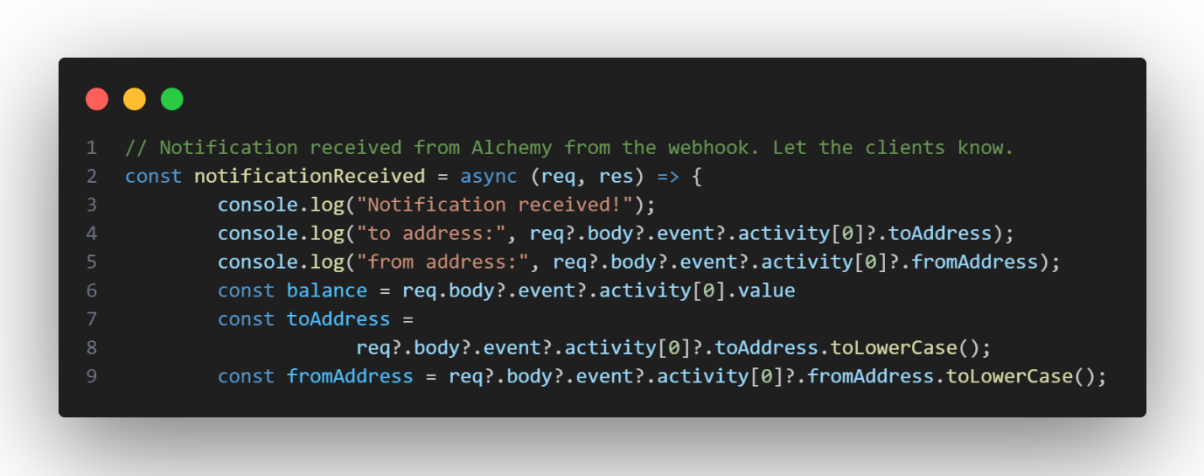


Figure 46 Alchemy Webhook dashboard

The Alchemy SDK requires a server to post a user's send and receive transaction events to. They recommended a Heroku hosted server and gave an example git repository to follow along with. To follow along, I cloned their example repository, replaced their API keys with mine, and then deployed the Express JS server to Heroku. I then manually added a blockchain address to their dashboard and sent some money to that address. After some debugging, the server started to pick up the webhooks sent from Alchemy to my server. So, when that user who was entered into the Alchemy dashboard sends or receives money, Alchemy will notify my server with the transaction details, such as the timestamp, value, toAddress, and fromAddress.

The 'notificationReceived' function (Figure 47 'notificationReceived' function) handles the post from the Alchemy Webhook. It is an async function that takes in two parameters: a request and a response. When Alchemy posts its webhook to my API, the function picks up the data as a request containing all the transaction details. To access the transaction value in the request, I created a variable named balance, which checks the 'req.body.event.activity.value'. This and the to and from address were needed to send notifications to a user.

This was hard to debug as I had to go through the Heroku logs, search for my 'console.log', and look for the transaction details. Also, I would have to connect to the blockchain and send some money each time I want to test if the transaction was being picked up by both Alchemy and my server. In development, debugging would be more accessible when running on localhost and my console; however, Alchemy required a hosted server to post the data to.



```
// Notification received from Alchemy from the webhook. Let the clients know.
const notificationReceived = async (req, res) => {
        console.log("Notification received!");
        console.log("to address:", req?.body?.event?.activity[0]?.toAddress);
        console.log("from address:", req?.body?.event?.activity[0]?.fromAddress);
        const balance = req.body?.event?.activity[0].value
        const toAddress =
                    req?.body?.event?.activity[0]?.toAddress.toLowerCase();
        const fromAddress = req?.body?.event?.activity[0]?.fromAddress.toLowerCase();
```

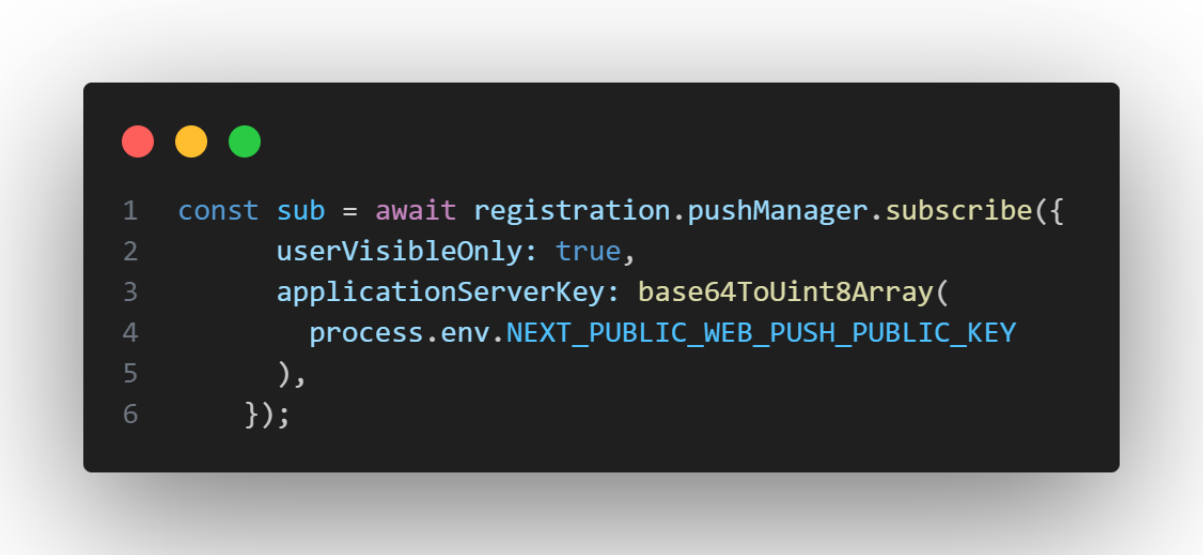
Figure 47 'notificationReceived' function

Next, I added an address using an API connecting to my Alchemy dashboard. This was necessary as the application wouldn't scale if I manually added each user to the dashboard. The function 'addAddress' (Figure 48 'addAddress' function) takes in a new address to add to the dashboard as a parameter. It then connects to the Alchemy API using the fetch method. My Alchemy API key is passed into the body or the request so they can verify which project to add the address to. Alchemy will provide a response in JSON to confirm the address has been added.

```
// Add an address to a notification in Alchemy
async function addAddress(new_address) {
    console.log("Adding address:", new_address);
    const body = {
        webhook_id: "wh_dnf6zvqpsvkd2p7t",
        addresses_to_add: [new_address],
        addresses_to_remove: [],
    };
    try {
        const response = await fetch(
            "https://dashboard.alchemyapi.io/api/update-webhook-addresses",
            {
                method: "PATCH",
                body: JSON.stringify(body),
                headers: {
                    "Content-Type": "application/json",
                    "X-Alchemy-Token": "2CxWNMpNMB3IeHKVJBROFv-19LQ3LKKQ",
                },
            }
        );
        const json = await response.json();
        console.log("Successfully added address:", json);
    } catch (err) {
        console.error("Error! Unable to add address:", err);
    }
}
```

Figure 48 'addAddress' function

Then, I used an SDK named 'Web Push' to implement the push notification in the application. Web Push was chosen as it is an industry standard and works on multiple devices, including iOS, Android, and several web browsers. Web Push allows users to subscribe to notifications. When a user subscribes, they are assigned an HTTPS endpoint to push the notification to. I created a subscribe function that returns a 'sub' variable. This sub variable (Figure 49 Subscription variable) is an object that has the user's endpoint.



```
const sub = await registration.pushManager.subscribe({
      userVisibleOnly: true,
      applicationServerKey: base64ToUint8Array(
        process.env.NEXT_PUBLIC_WEB_PUSH_PUBLIC_KEY
      ),
    });
```

Figure 49 Subscription variable

Once the user has subscribed, the ‘createUser’ function (Figure 50 ‘createUser’ function) is called to store the user's endpoint in a database. This function takes the user's sub variable and posts it to my ‘register’ endpoint, which will add the user's sub to the database.

```
const createUser = async (sub: any) => {
    try {
      console.log('going to create user');
      await fetch(
        'https://payments-backend-01-0651b5f97107.herokuapp.com/api/users/register',
        {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
          },
          body: JSON.stringify({
            address: address.toLowerCase(),
            subscription: sub,
          }),
        }
      );
    } catch (error) {
      console.error('Error:', error);
    }
  };
```

Figure 50 ‘createUser’ function

The ‘register’ function (Figure 51 ‘register’ function) is in my user controller in my Express JS server. This function handles the post request for creating a user. The function then checks to see if the user is already subscribed. If they are, it deletes their old subscription and creates a new one to always have the latest endpoint. If the user isn’t in the database, their address is added to the Alchemy dashboard, and then a new user is created in the database using the body of the post request.

```
const register = async (req, res) => {
    console.log("Registering user");
    console.log(req.body);

    try {
        // Check if a user with the same address already exists
        let existingUser = await User.findOne({ address: req.body.address });

        if (existingUser) {
            // If user with the same address already exists, delete them
            await User.findOneAndDelete({ address: req.body.address });
        }

        // Add the address to the Alchemy webhook for notifications
        addAddress(req.body.address);

        // Proceed with registering the new user
        const newUser = new User(req.body);
        await newUser.save();

        return res.status(201).json(newUser); // Return the newly registered user
    } catch (error) {
        console.error("Error registering user:", error);
        return res.status(500).json({ error: "Internal server error" });
    }
};
```

Figure 51 'register' function

Once the user has successfully subscribed, their address is in the Alchemy dashboard, and their subscription object is held in the database, they are nearly ready to receive notifications. When a user receives money, the Heroku Express server will pick up that the subscribed user received money and will then post a message to the user's device. In Next JS, I created a 'route' file that allows you to create custom request handlers. My server posts a request to this route, passing in the balance of the payment they just received and a custom message. Just before it posts to the client application, it gets the user's subscription endpoint from the database, as it's required to send notifications (Figure 52 Send notification API request).

```
try {
    const response = await axios.get(
                `https://payments-backend-01-0651b5f97107.herokuapp.com/api/users/${toAddress}`,
                {
                    headers: {
                        "Content-Type": "application/json",
                    },
                }
            );

    console.log("Response:", response.data);
    await fetch("https://payments-lyart.vercel.app/notification", {
        method: "POST",
        headers: {
            "Content-type": "application/json",
        },
        body: JSON.stringify({
            subscription: response.data.subscription,
            message: `Hey you just received ${balance}!`,
        }),
    });
    console.log("Notification sent!");


} catch (error) {
    console.error("Error:", error.message);
}
```

Figure 52 Send notification API request

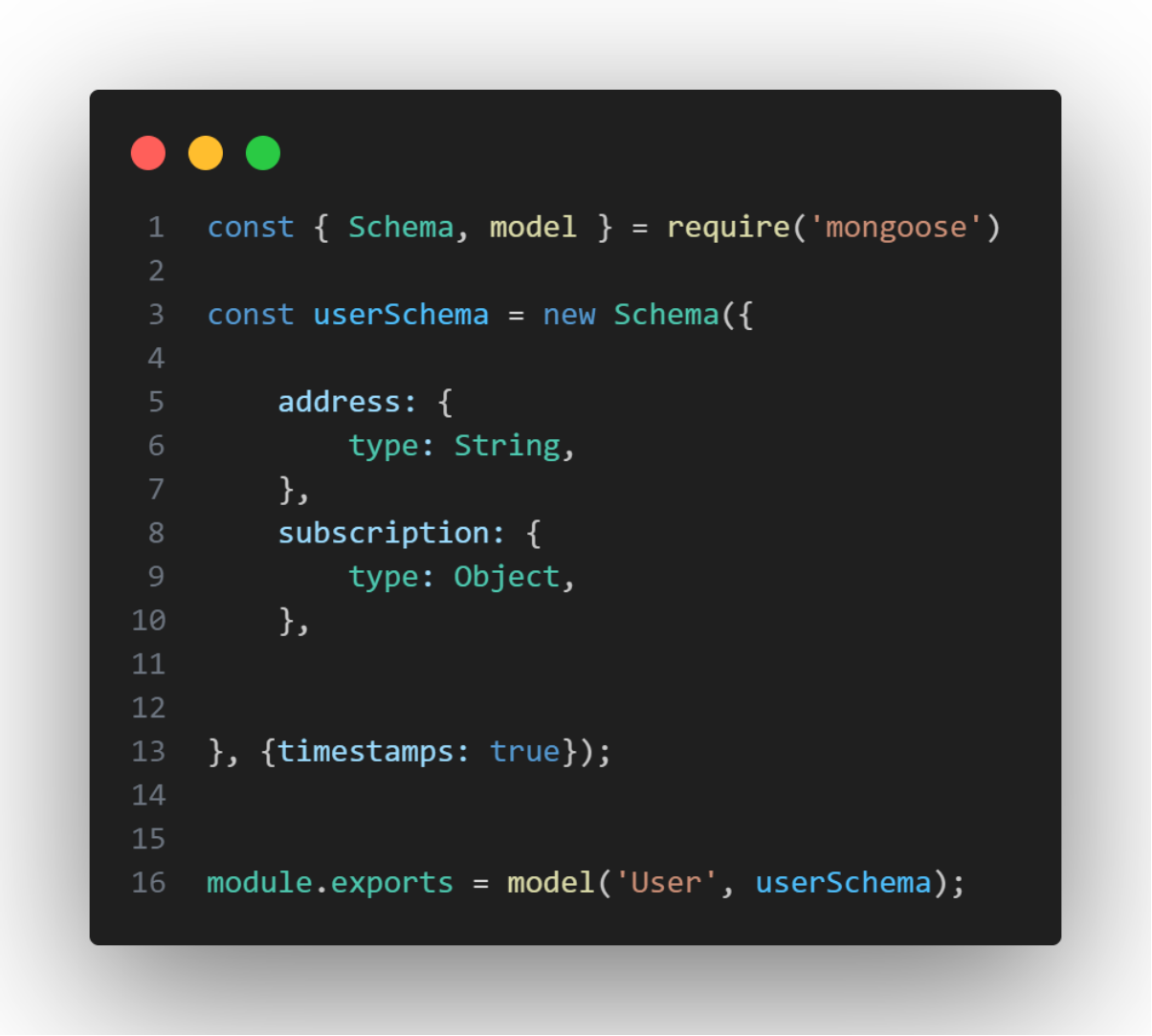
The ‘route’ (Figure 53 Next JS route file) file in the Next JS application then receives this post request with the transfer balance and the subscription object. It deconstructs the message and the subscription from the post request. Web Push is then set up using the setVapidDetails method. Once Web Push is set up, the ‘sendNotification’ method on the ‘webPush’ variable can be called. The user's subscription and the message are passed into the method. Then, once the user receives a payment, they get a notification on their device.

```ts
export const POST = async (req: NextRequest) => {
  if (
    !process.env.NEXT_PUBLIC_WEB_PUSH_PUBLIC_KEY ||
    !process.env.NEXT_PUBLIC_WEB_PUSH_EMAIL ||
    !process.env.NEXT_PUBLIC_WEB_PUSH_PRIVATE_KEY
  ) {
    throw new Error('Environment variables supplied not sufficient.');
  }
  const { subscription, message } = (await req.json()) as {
    subscription: webPush.PushSubscription;
    message: string;
  };
  console.log(subscription);
  try {
    webPush.setVapidDetails(
      `mailto:${process.env.NEXT_PUBLIC_WEB_PUSH_EMAIL}`,
      process.env.NEXT_PUBLIC_WEB_PUSH_PUBLIC_KEY,
      process.env.NEXT_PUBLIC_WEB_PUSH_PRIVATE_KEY
    );
    const response = await webPush.sendNotification(
      subscription,
      JSON.stringify({
        title: `PWA Payments`,
        message: `${message}`,
      })
    );
    return new NextResponse(response.body, {
      status: response.statusCode,
      headers: response.headers,
    });
  } catch (err) {
    if (err instanceof webPush.WebPushError) {
      return new NextResponse(err.body, {
        status: err.statusCode,
        headers: err.headers,
      });
    }
    console.log(err);
    return new NextResponse('Internal Server Error', {
      status: 500,
    });
  }
};
```

Figure 53 Next JS route file

In the future, the Web Push functionality can be moved from Next JS to the Heroku Express server to consolidate and organise the backend.

For the database, I chose a MongoDB database as it is fast and reliable and relatively easy to set up and host a small database, compared to Laravel, which can have a lot of project bloat and requires an SQL database to be hosted. In the same codebase as the Heroku backend, I set up a model file that defines the schema for a user (Figure 54 User schema model). A user can have a blockchain address to identify the user and a subscription, which is required for Web Push. The database was then hosted on MongoDB.



```
const { Schema, model } = require('mongoose')

const userSchema = new Schema({

    address: {
        type: String,
    },
    subscription: {
        type: Object,
    },


}, {timestamps: true});


module.exports = model('User', userSchema);
```

Figure 54 User schema model

To test the notifications, I created a JavaScript file to test sending a push notification to a device. I hoped to get the script to send a notification to a test endpoint, for example, my phone or laptop. The script ‘sendPushNotification.js’ script (Figure 55 ‘sendPushNotification’ script) could be run with node and would declare a set of public and private keys, and an endpoint to push a notification to. Once this functionality was working after a lot of debugging, I could implement it on my Heroku Express server.

```
//import { NextRequest, NextResponse } from 'next/server';
const webPush = require('web-push');

const sendPushNotification = async () => {

  const publicKey =
    'BKIvOJTEdSWEOc3P_-BI2yIGHOJHYrZaNv_hGamAKOX_VaMaEYILZetpjEva9PDd1S9HsP6afe7WATCHcpJmcoOKbSkEaE_zG2E8zlbIs';
  const privateKey = 'MJWB6JVXESFaBuXTBFcJw';

  const pushSubscription = {
    endpoint:
      'https://fcm.googleapis.com/fcm/send/d91W5xF9fDY:APA91bH4-ImywZYwrqaUCdP2mgUQVvmryYmeYi4lxsll568jcP0JdkrVUbcOvjsAi91g_cV3_idplV_F198LeOm1iclff8Ili2xWorI-mLkFVAsPe6gWttXCEMxndqYnpvWEsOxiT-iC',
    expirationTime: Math.floor(Date.now() / 1000) + 10,

    keys: {
      auth: privateKey,
      p256dh: publicKey,
    },
  };

  console.log(pushSubscription);
  try {
    await fetch('https://payments-lyart.vercel.app/notification', {
      method: 'POST',
      headers: {
        'Content-type': 'application/json',
      },
      body: JSON.stringify({
        subscription: pushSubscription,
        message: 'Hello from ste',
      }),
      signal: AbortSignal.timeout(10000),
    });
  } catch (error) {
    console.log(error);
  }
};

sendPushNotification();

```

Figure 55 'sendPushNotification' script

In the future, I plan to add authorisation to the database so users can't access other users' endpoints. I also want to add encryption so that in case of a database leak, users' endpoints won't be affected and spammed by a malicious actor. Also, it would be nice to support multiple subscriptions so a user can receive notifications on multiple devices. Multiple subscription endpoints would be easy to implement in the future, by storing each the subscription object in an array on subscriptions. Then on my Express backend, notifications could be sent to each endpoint for each user.

Privy

During my interim presentation, my supervisor advised me to reach out to a blockchain developer or developer community to see if my application was secure. I then went to Reddit to the page "r/ethdev", a development community for the Ethereum blockchain, to ask them if my login implementation from sprint 1 was secure. After receiving feedback about my implementation, one user advised me to switch to using 'Privy', a JavaScript library that handles the authentication flow in your app. Privy handles the creation of blockchain wallets, key management, and authorisation, and it allows for several social login methods, such as Google, Apple, and email. Privy also integrated with ZeroDev to help developers implement account abstraction wallets while handling the authentication flow.

As these two libraries are designed to work together, the documentation process was easy to follow. As I was familiar with the ZeroDev documentation from my previous implementation, I could go straight to Privy and start converting my old implementation to a new one. After reading the Privy documentation, I cloned their example 'create-next-app' (Privy, Privy, 2024) from GitHub and analysed how they implemented the login process. This example, however, was for a regular wallet and not an account abstraction wallet. So I then went to another GitHub example, "zerodev-example" (Privy, Privy, 2024), and again analysed how they implemented an account abstraction wallet and how I could combine this with a Next JS app. Finally, I researched how Privy can be implemented in a PWA, as small changes often must be made.

To implement Privy, I first created an account with the Privy dashboard and got my ‘App Id’ key. Next, I went to the root of my application, the Next JS layout file, to wrap the whole application in a ‘PrivyProvider’ (Figure 56 'PrivyProvider' context). This provider allows my whole application access to the Privy context. The PrivyProvider needed to be wrapped inside the ZeroDev provider from my first implementation, as Privy needs access to the ZeroDev context to create the account abstraction wallet. For the PrivyProvider context to work, it takes my app ID and config as props. The privy login modal can be changed in the config option, including the theme, login methods, blockchains, and other features. One change that had to be implemented in the config was “createOnLogin” had to be set to true. This was necessary for Privy to work with progressive web apps.

```tsx
<ZeroDevProvider projectId={'f6375b6f-2205-4fc7-bc87-f03218789b86'}>
  <PrivyProvider
    onSuccess={() => {
      router.push('/home');
    }}
    appId={process.env.NEXT_PUBLIC_APP_ID as string}
    config={{
      appearance: {
        theme: 'dark',
      },
      defaultChain: sepolia,
      loginMethods: ['apple', 'google', 'email'],
      embeddedWallets: {
        createOnLogin: 'users-without-wallets',
        noPromptOnSignature: true,
      },
    }}
  >
    <Providers>
      <PersistGate loading={null} persistor={persistor}>
        <WagmiProvider config={config!}>
          <QueryClientProvider client={queryClient}>
            <AnimatePresence mode='wait' initial={false}>
              <LayoutGroup>

                  <div>{auth}</div>
                  <div>{drawer}</div>

                  <main
                    vaul-drawer-wrapper=''
                    className='h-[100vh] text-gray-300'
                  >
                    {children}
                  </main>
                  <div className='w-full overflow-hidden'>
                    <BottomNavbar />
                  </div>

              </LayoutGroup>
            </AnimatePresence>
          </QueryClientProvider>
        </WagmiProvider>
      </PersistGate>
    </Providers>
  </PrivyProvider>
```

Figure 56 'PrivyProvider' context

Then, I used Privy's hook 'usePrivySmartAccount' (Figure 57 'usePrivySmartAccount' hook) to allow a user to log in. This hook has a method called login, which, when invoked, will show a popup on the client's screen, prompting them with the log-in options configured in the PrivyProvider. Once the login is successful, it invokes a callback function named "onSuccess" that is passed into the PrivyProvider. In my PrivyProvider, I passed a function to route the user to the homepage.

```
// privy
const { ready, authenticated, login, zeroDevReady, user } =
  usePrivySmartAccount();
```

Figure 57 'usePrivySmartAccount' hook

A button (Figure 58 Login button) is responsible for calling the login method.

```
<Button className='mt-4' disabled={!ready || authenticated} onClick={login}>
  Login
</Button>
```

Figure 58 Login button

When the login is successful, I have a 'useEffect' (Figure 59 'useEffect' function to check if the user is authenticated) React hook that watches for changes in the 'authentication' and 'zeroDevReady' variables. Once these variables turn true, it means that the user was authenticated successfully and the ZeroDev account abstraction wallet is ready to use. Once they are ready, I get the user's wallet address from the Privy user object and set it in my Redux store using the dispatch hook.

```
useEffect(() => {
  if (zeroDevReady && authenticated ) {
    // set user address
    dispatch(setAddress(user?.wallet?.address));

    // route home
    router.push('/home');
  }
}, [authenticated, zeroDevReady]);
```

Figure 59 'useEffect' function to check if the user is authenticated

Privy generally improved the login flow, as I didn't have to store the user's private key anywhere in my application. It also provided useful hooks for logging in, getting the user's account information, and for authorisation.

Sending USDC

Because I changed how the account is created and changed to use Privy, this affected the functionality of how a user would use their blockchain wallet. This, in particular, affected how the user would send money, as the functions required to call the smart contract changed. I changed the application to no longer use the 'sendTx' function to send USDC.

As part of the design process, I created a confirmation page where users would be prompted with the amount of money and the payee they were going to send it to. This is where I moved the sending money functionality to.

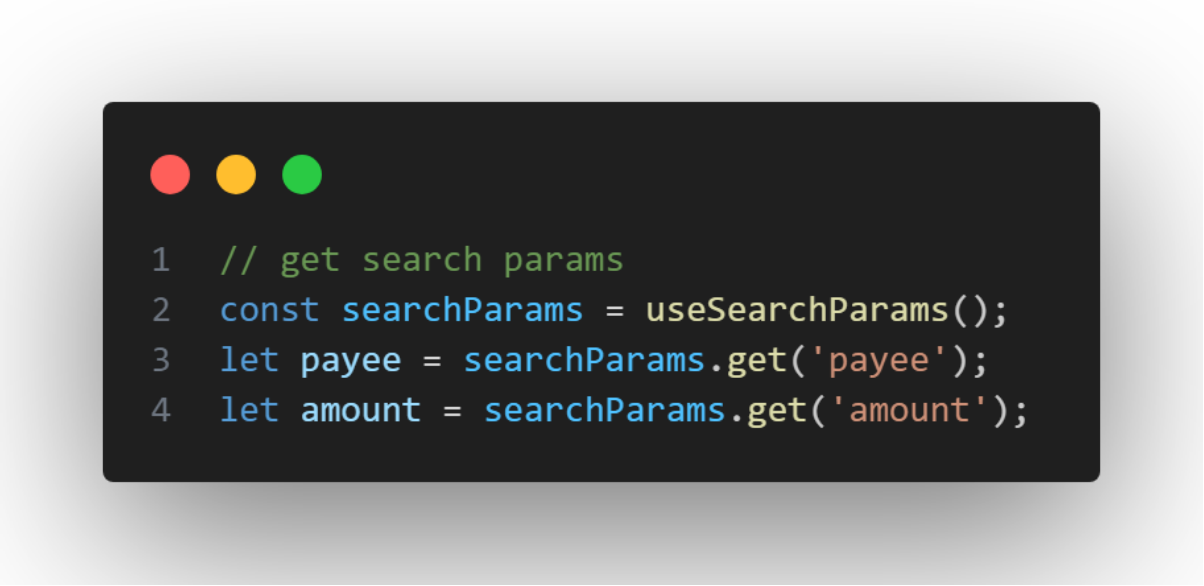
While reprogramming the sending functionality, I took the opportunity to change the 'sendTx' function into a custom React hook named 'useSendUsdc' (Figure 60 Send USDC hook). This would separate the code to make it cleaner to read and more reliable. This hook exported several methods and variables, including a function 'sendUsdc' to send the USDC, a 'transactionStatus' variable to make sure the transaction was confirmed, a loading status variable to make sure a loading spinner can be shown when the transaction is waiting and finally, a transactionHash variable which would be used to show the user their transaction details.

The 'sendUsdc' function in the hook first checks to see if the wallet is ready by checking the 'zeroDevReady' variable, which should return a boolean true variable. It then checks to ensure the amount the user wants to send is greater than 0. Then, the rest of the sending, including the 'encoding' and the 'sendTransaction' functionality, is the same as the old implementation.

```
// react
import { useState} from 'react';
// viem
import { encodeFunctionData, parseUnits, erc20Abi } from 'viem';
// privy
import { usePrivySmartAccount } from '@zerodev/privy';

const useSendUsdc = () => {
  const { zeroDevReady, sendTransaction } = usePrivySmartAccount();
  const [transactionStatus, setTransactionStatus] = useState<boolean>(false);
  const [loading, setLoading] = useState<boolean>(false);
  const [transactionHash, setTransactionHash] = useState<string>('');

  const sendUsdc = async (amount: string, payee: string) => {
    try {
      if (!zeroDevReady || amount == '' || amount == '0') {
        return;
      }
      console.log('ready to send');
      const encoded = encodeFunctionData({
        abi: erc20Abi,
        functionName: 'transfer',
        args: [payee as `0x${string}`, parseUnits(amount, 6)],
      });

      setLoading(true);
      const txnHash = await sendTransaction({
        to: '0x94a9D9AC8a22534E3FaCa9F4e7F2E2cf85d5E4C8', // USDC contract address
        value: BigInt(0),
        data: encoded,
      });

      if (txnHash) {
        setLoading(false);
        setTransactionStatus(true);

        setTransactionHash(txnHash);

        console.log('Transaction Hash:', txnHash);
      }
    } catch (error) {
      console.log(error);
    }
  };

  return { sendUsdc, transactionStatus, loading, transactionHash };
};

export default useSendUsdc;

```

Figure 60 Send USDC hook

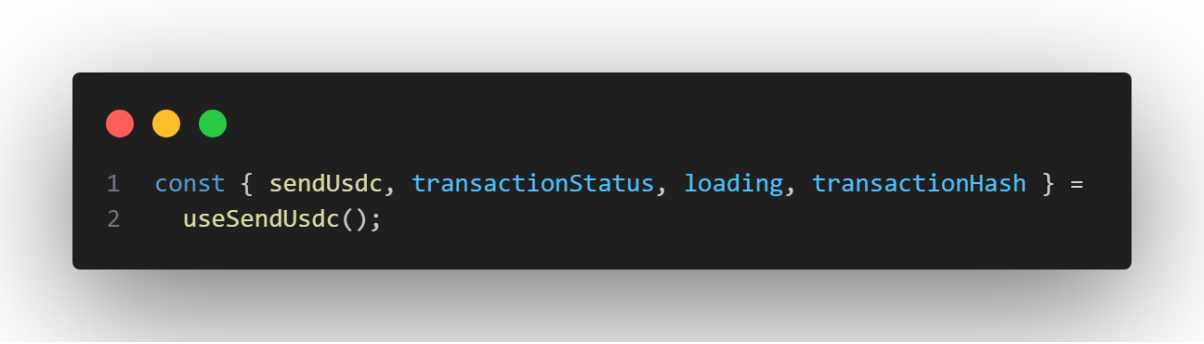
Next, on the confirmation page, the payee's address and the amount to send are retrieved from the search params (Figure 61 Search params), which are set in the send page.



```
// get search params
const searchParams = useSearchParams();
let payee = searchParams.get('payee');
let amount = searchParams.get('amount');
```

Figure 61 Search params

Then, the methods and variables are imported from the ‘useSendUsdc’ (Figure 62 'useSendUSdc' hook methods and variables).



```
const { sendUsdc, transactionStatus, loading, transactionHash } =
  useSendUsdc();
```

Figure 62 'useSendUSdc' hook methods and variables

Then, a button is used to invoke the ‘handleSend’ method (Figure 63 Send Button).

```
<Button
  onClick={handleSend}
  className='text-xl'
  size={'lg'}
  variant={'default'}
>
  <div className='flex content-center items-center'>
    <div className='text-xl'>
      <div>Send</div>
    </div>
  </div>
</Button>
```

Figure 63 Send Button

The ‘handleSend’ function (Figure 64 'handleSend' function) then checks to ensure that the user inputs an amount and an address to send it to. If the user did, the sendUsdc method is called with the amount and payee passed in as parameters.

```
const handleSend = () => {
  if (!payee || !amount) return;
  console.log('calling hook');
  sendUsdc(amount, payee);
};
```

Figure 64 'handleSend' function

While the transaction is being sent, the loading variable is set to true to give the user feedback that the payment is being sent. Once the transaction is successful, the transactionStatus variable is set to true, and loading is set to false. Then, the user is redirected to a page where they can see the transaction receipt, which is retrieved using the transactionHash variable from the hook.

Keypad

Instead of using the iPhone default input keypad, I created a custom keypad. This was also influenced by looking at similar applications and how they let users input amounts to send. After some research on how to implement a keypad, I found an implementation of the IOS calculator app in CodeSandBox by Sam Selikoff (Selikoff, 2024). I then forked the code and changed the keypad to suit my own design. In particular, I added a delete and full stop button.

The keypad (Figure 65 Keypad component functionality) works by having a button for each number, and the button's value is assigned to the element. When a button is clicked, that number is added to an array of numbers, which will be the value the user wants to send to their payee. When the back button is clicked, it will remove the last number in the array.

The keypad component receives a callback function through its props, responsible for passing the value back to its parent component. This is needed so the value can be then passed on to the confirm page through the search params where the ‘useSendUsdc’ hook will be called.

```
<div className='mt-9 flex flex-wrap justify-between gap-4'>
  <Button onClick={() => handleClick('7')}>7</Button>
  <Button onClick={() => handleClick('8')}>8</Button>
  <Button onClick={() => handleClick('9')}>9</Button>
  <Button onClick={() => handleClick('4')}>4</Button>
  <Button onClick={() => handleClick('5')}>5</Button>
  <Button onClick={() => handleClick('6')}>6</Button>
  <Button onClick={() => handleClick('1')}>1</Button>
  <Button onClick={() => handleClick('2')}>2</Button>
  <Button onClick={() => handleClick('3')}>3</Button>
  <Button onClick={() => handleClick('.')}>.</Button>
  <Button onClick={() => handleClick('0')}>0</Button>
  <Button
    onClick={() => {
      setNums(nums.slice(0, -1));
    }}
  >
    <ArrowLeft size={40} className='m-auto text-center' />
  </Button>
</div>

```

Figure 65 Keypad component functionality

Sprint 6

Goal

In this sprint, the goal was to implement authenticated routes, add a bottom navigation component, add some nice UI animations and motion design, change the design system from only using Tailwind to using a component library and implement a contacts feature to the app.

Authenticated routes

Once Privy had been setup in the application, it was then possible to make authenticated routes. The ‘usePrivySmartAccount’ hook allows access to an authenticated variable. I can use this variable to check if the user is logged in or not, and control what routes they can go to.

To do this, I created an ‘AuthPage’ component (Figure 66 Authorized route component). This component would wrap around any page that I wanted to make an authorized only route. For

example if I didn’t want a user to access the homepage without first logging in I could wrap the homepage in this component. If the user tries to access the homepage they will be rerouted to the login page if they are not yet authenticated.

```
import { useRouter } from "next/navigation";
import { useEffect } from "react"

// privy
import { usePrivySmartAccount } from "@zerodev/privy";


export default function AuthPage ({children} : {children: React.ReactNode}) {

    const router = useRouter();

    const { authenticated } = usePrivySmartAccount();

     useEffect(() => {

        if (!authenticated) {
          router.push('/login');
        }

    }, [authenticated]);
    return (
        <>{children}</>
    )
}
```

Figure 66 Authorized route component

Bottom Navbar

To implement the ‘Navbar’ component, I followed the recommendations in the Next JS documentation. They recommend creating a navbar component, and then rendering that component in the layout file, so it renders over all pages that are children of the layout.

As the Navbar renders over all routes in the app, I wanted to make sure the user couldn’t see it on the login page. The navbar has a ‘useEffect’ function (Figure 67 Logic to show the bottom navbar on specific routes) that checks when the application's path changes. If the path is to the login page, the navbar which is conditionally rendered using a ‘useState’ variable, will be removed from the DOM.

```
const pathname = usePathname();

const [showNav, setShowNav] = useState(true);

 useEffect(() => {
  if (pathname == '/' || pathname == '/login') {
    setShowNav(false);
  } else {
    setShowNav(true);
  }
}, [pathname])
```

Figure 67 Logic to show the bottom navbar on specific routes

Framer motion Layout ids

Several animations are present in the UI. To implement them, I used the Framer Motion library. For Framer Motion to work, it must be installed using npm. The application is wrapped in an 'AnimatePresence' component, which will listen to see if an element is being removed from the DOM and animate it out.

For an element to be animated, it must be a motion component. This is done by adding 'motion' to the start of the element. So a normal HTML 'div' becomes a 'motion.div' (Figure 68 Framer Motion div).

Figure 68 Framer Motion div

For exit animations to work, each element must have a key that acts as an ID so the ‘AnimatePresence’ component (Figure 69 'AnimatePresence' component) can check to see if it has been removed.

In this example, the ‘motion.div’ is wrapped in an ‘AnimatePresence’. Then, for exit animations to work, the child ‘motion.div’ must be conditionally rendered using a boolean variable, and the div must have a key. Then, the ‘motion.div’ can take in props to control the animation. In this example, I control the element's opacity and the animation's transition. By passing an initial object as a prop, the element's initial state can be controlled. By setting the opacity to zero, the ‘motion.div’ will not be visible. Then, an animate prop can be passed in to animate the ‘motion.div’ back to full opacity. Then, similar to the initial, the exit prop will set the opacity back to zero. The animation can be controlled with a transition prop, which can control the duration, and easing of the animation.

```jsx
<AnimatePresence>
  {payee !== '' && (
    <motion.div
      key='clear'
      initial={{ opacity: 0 }}
      animate={{
        opacity: 1,
        transition: { duration: 0.4, ease: 'easeInOut' },
      }}
      exit={{ opacity: 0 }}
      className='ml-auto grid content-center justify-end'
    >

    </motion.div>
  )}
</AnimatePresence>
```

Figure 69 'AnimatePresence' component

Background Gradient

To implement the background gradient (Figure 70 'Aceternity' background gradient animation) on the homepage and login page, I installed the component from 'Aceternity', a component library for React, built with Framer Motion and Tailwind. I added the component to the components folder and made some small changes to the colours and animation length. I could then render the background gradient inside an 'absolute' placed div, with a 'zIndex' of '-50' to place the gradient behind the rest of the divs.

Figure 70 'Aceternity' background gradient animation

https://ui.aceternity.com/components/background-gradient-animation

ShadCN

As the application grew, it became more work to rewrite the same Tailwind styles for each component throughout the app. When working on the design sprint, I decided to switch from designing all my own components in Tailwind to building a component library.

I chose a new framework named 'ShadCN', which describes itself as not a component library but a collection of reusable components that can be used to build your own library. ShadCN is built with Radix UI and Tailwind CSS. ShadCN was chosen as components can be added individually rather than adding every component from the library. This reduces your project size as there are no extra unused components. As ShadCN is built using tailwind, it is very easy to customise components as needed by overwriting the CSS classes in the component. ShadCN comes with various useful UI components such as buttons, dropdowns, sheets, inputs, and many more.

To install ShadCN, Tailwind must be set up in the project. Then, ShadCN can be added using the 'npx shadcn-ui@latest init' command. Once it's set up, any component can be added again using the 'npx' command and the component's name.

Ios specific CSS problems

I noticed some CSS problems when designing the app due to IOS and how it interprets CSS. I wanted to make sure that a user couldn't zoom in to make the app feel more like a native app rather than a website. To do this, I applied a CSS class (Figure 71 Cusom CSS class to disable zoom) to all elements in my app, turning off the zoom by targeting the touch action property.

Figure 71 Cusom CSS class to disable zoom

Another problem with IOS is that any time a user interacts with an input field where the text size is less than 16 pixels, the device automatically zooms in on the input field. This often breaks the UI as elements are then off the screen, which can be annoying for users. To fix this, all inputs were made to have a text size greater than 16 pixels.

Contacts

I added a contacts feature to improve the app's UX. This would make the app feel better, as currently, each payee in the app is displayed by their unique blockchain address. This address is quite long and confusing for many users. To replace this blockchain address, I first added a package named ‘truncate-eth-address,’ which contained a function to shorten a blockchain address to 10 characters, making it easier to read (Figure 72 truncate-eth-address’ shortened blockchain address).

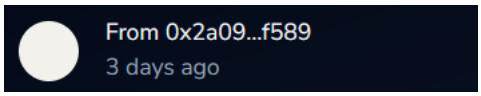


Figure 72 truncate-eth-address’ shortened blockchain address

However, this still was confusing for people, so I added a contacts feature where a user could assign a username to an address on their device. I had two options for this approach: either store a user's contacts in a MongoDB database or store the contacts in the device's local storage. Both were possible to set up as I already had a MongoDB collection for all the app’s users from the push notifications from a previous sprint. So, I could just add another entry in the user's document under their push notification subscription, which would be an

array of contact objects and could be served by the same API and backend hosted on Heroku.

However, I decided to store the contacts in the device's local storage using my Redux store setup from previous sprints, as this would then work offline and wouldn't have to be encrypted. Also, at this stage of the project, I had to weigh up which features would take more time to implement and figured it would be more time-efficient to save them to local storage. Also, going forward, a contacts feature could be added to the MongoDB, and when the app starts up, it could fetch all the user's contacts and store them in local storage, building on what I've already done.

First, I created a redux slice (Figure 73 Contacts Redux slice) that declares the contacts as an array and exports the 'setContacts' function, which updates the contacts array when called.

```
'use client';

import { createSlice } from '@reduxjs/toolkit';

export const contactsSlice = createSlice({
  name: 'contacts',
  initialState: {
    value: [],
  },
  reducers: {
    setContacts: (state, action) => {
      state.value = action.payload;
    },
  },
});
export const { setContacts } = contactsSlice.actions;
export default contactsSlice.reducer;

```

Figure 73 Contacts Redux slice

Then, I created a react component responsible for adding a contact functionality. When adding a contact, first, the 'handleAddContact' function (Figure 74 'addAContact' function) is called. This function then checks if the contact is available in the contacts array so that there are no duplicate entries. Then, if the contact is available, the 'useDispath' redux hook is used to update the contacts array in local storage. The 'setContacts' function is passed into the 'useDispatch' hook to update the contacts. The spread operator is also passed into the

function to access the previous state of the contacts array, to make sure that all the previous state of contacts are kept, and the new contact is added to the end of the array. If the spread operator wasn't used, then all the previous contacts would be deleted.

```
const handleAddContact = () => {
  if (!isContactAvailable(newContactName)) {
    dispatch(
      setContacts([
        ...contactsState,
        { name: newContactName, address: payee },
      ])
    );
    setShowAddContact(false);
    setNewContactName('');
  } else {
    alert('Contact already exists!');
  }
};
```

Figure 74 'addAContact' function

The 'isContactAvailable' function (Figure 75 'isContactAvailable' function) is invoked in the 'addAContact' function. The function takes in the new contact's address and checks if it is already in the contacts array using the JavaScript 'some' method. It then returns a boolean value indicating whether they are in the array or not.

```
const isContactAvailable = (name: string): boolean => {
    return contactsState.some(
      (contact: Contact) => contact.name === name || contact.address === payee
    );
  };
```

Figure 75 'isContactAvailable' function

Another check is done in the UI before adding the address to ensure that it is a valid blockchain address and to prevent application errors. The ‘handleAddAddress’ (Figure 76 ‘handleAddAddress’ function) function uses the ‘isAddress’ function imported from the ‘Viem’ package, which returns a boolean value based on whether the address is valid or not.

```
const handleAddAddress = (e: React.ChangeEvent<HTMLInputElement>) => {
  const isAddressValid = isAddress(e.target.value);
  setNewContactAddress(e.target.value);
  !isAddressValid && setShowErrorMessage(true);
};
```

Figure 76 ‘handleAddAddress’ function

If all the checks pass, when the user inputs the new contact name into the input (Figure 77 Add an address input field), the address is then added to the contacts array.

```
<Input
  onChange={(e) => setNewContactName(e.target.value)}
  type='text'
  id='name'
  name='name'
  style={{ width: '100%' }}
  value={newContactName}
  placeholder='Name'
  className='w-full text-base'
/>
```

Figure 77 Add an address input field

A contact can be added to the contacts page or the payee page. To improve the user experience, I decided to check if a recent payee was in the contacts array and, if not, prompt the user with a button to add them. I use the ‘isInContacts’ function (Figure 78 ‘isInContacts’ function) to check if the payee is in the contacts array.

```
const isInContacts = contactsState.some(
  (contact: any) => contact.address == payeeAddress
);

```

Figure 78 'isInContacts' function

Then, in the payee page, if the payee is not in the contacts array, a conditional block of JSX code (Figure 79 Conditional JSX code to show add a contact button) is rendered. This code contains a button to save the payee to their contacts.

```
{!isInContacts && (
   <div className='ml-auto'>
     <div
       onClick={handleAddUser}
       className='text-muted-foreground flex space-x-2 text-base font-light'
     >
       <UserPlus
         strokeWidth={2}
         className='fill-muted-foreground stroke-muted-foreground'
       />
       <p>Save</p>
     </div>{' '}
   </div>
 )}
```

Figure 79 Conditional JSX code to show add a contact button

If the payee is not in the contacts array, the user's blockchain address is shown using the 'truncateEthAddress' function to shorten it (Figure 80 UI where the contact is not in the array of contacts), and the save button is displayed beside the address.

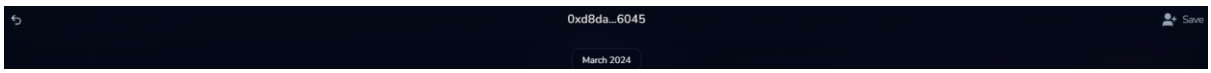


Figure 80 UI where the contact is not in the array of contacts

When the save contact button is clicked, a modal (Figure 81 Add a contact modal) is shown to the user to add the contact.

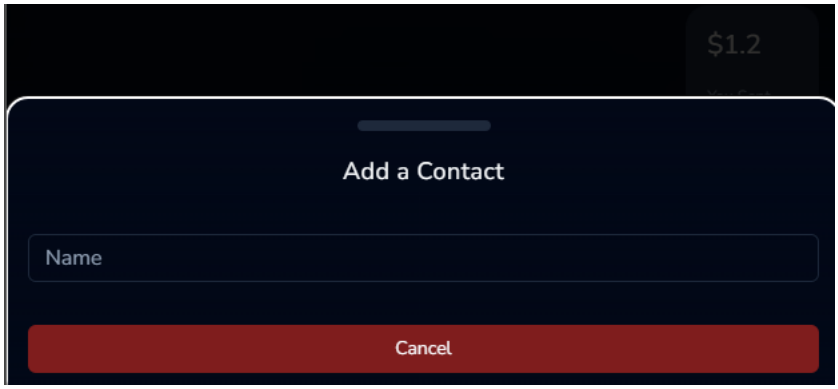


Figure 81 Add a contact modal

Then, when the user is added, their name is displayed throughout the app. This can also be confirmed in the contacts page.

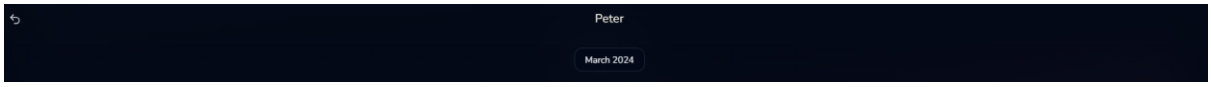


Figure 82 Payee is added to contacts in the Payee page

In the contacts page (Figure 83 All contacts in the contacts page), all the user's contacts can be viewed, and when a contact is clicked, the user is navigated to the send money page to pay the contact. A new contact can also be added from this page.

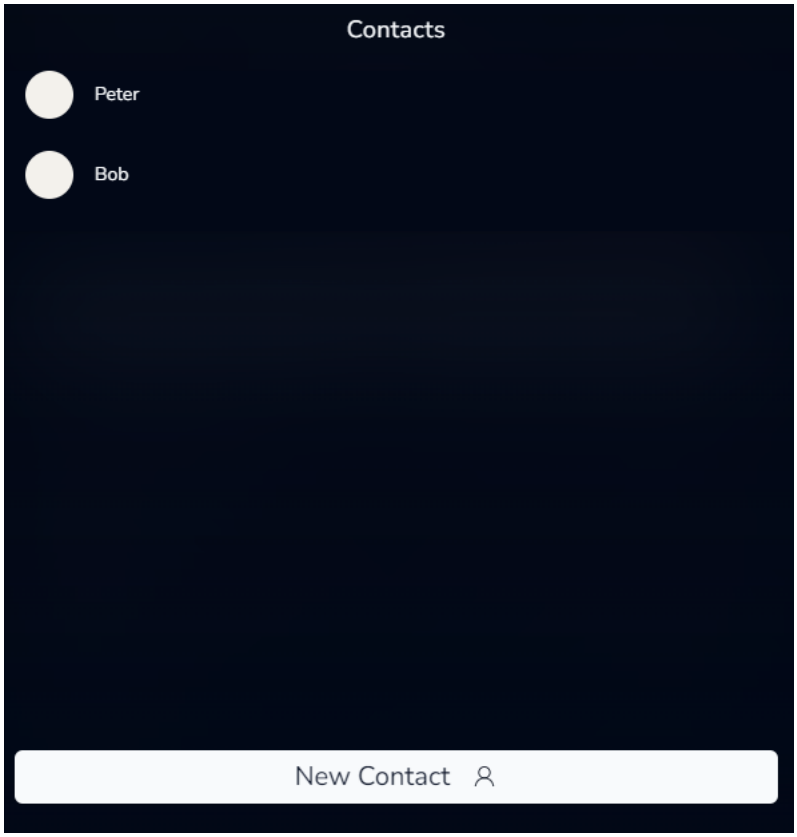


Figure 83 All contacts in the contacts page

A react hook was created to display the contact name in the UI. The ‘useFindPayeeName’ hook (Figure 84 'useFindPayeeName') is used to search the contacts array for a blockchain address, find the contact name associated with that address, and return the contact name. It

returns a shortened version of the blockchain address if it does not find contact with that name. The JavaScript find method is used to search the contacts array for the blockchain address passed into the hook.

```
import { useSelector } from 'react-redux';
import { RootState } from '@/GlobalRedux/store';
import { Contact } from '@/app/types/types';
import truncateEthAddress from 'truncate-eth-address';

const useFindPayeeName = (payeeAddress: string, contactsState : any) => {

  const findPayeeName = (payeeAddress: string): string | null => {
    if (!contactsState || contactsState.length === 0) {
      return truncateEthAddress(payeeAddress);
    }

    // Ensure to lower case both sides to match
    const contact = contactsState.find(
      (element: Contact) => element.address?.toLocaleLowerCase() === payeeAddress.toLocaleLowerCase()
    );

    return contact ? contact.name : truncateEthAddress(payeeAddress);
  };

  return findPayeeName(payeeAddress);
};

export default useFindPayeeName;

```

Figure 84 'useFindPayeeName'

The hook can then be used (Figure 85 Code to show the Payee's name) in the app by calling it with a payee’s blockchain address passed in as a parameter.

```
  You've got no transfers with{' '}
   {payeeAddress && useFindPayeeName(payeeAddress, contactsState)}
</p>
```

Figure 85 Code to show the Payee's name

If I had more time, I would like to have used ENS (ENS, 2024), a decentralised identity solution, where each user can hold their own ENS username in their wallet as an NFT. A blockchain-based app can then use this username as it belongs to the user, providing more interoperability between applications. It also means that applications do not have to store user information but is rather written to the blockchain. Recently, ENS released a new feature called Sub Domains. This allows an app to have a domain like ‘payments.eth’ and allows users to pick a subdomain for their wallet under the app’s subdomain. A user could

choose a subdomain such as ‘user1.payments.eth’ and receive payments to this username, which maps directly to their blockchain address.

By adding ENS domains to an application, users no longer have to interpret the long blockchain addresses, and applications don’t have to store usernames. ENS users can also set a profile picture, providing better UX in the app, as users could navigate contacts based on pictures.

Sprint 7

Goal

In this sprint, the aim was to conduct user testing for the app. Several types of testing will be carried out, such as user, unit, and manual functional testing.

Unit testing

Several parts of the application, both front end and back end, were subject to unit testing, which is described in the testing chapter.

User testing

User testing was carried out, which is described in the testing chapter.

Sprint 8

Goal

In this sprint, the goal was to implement the recommended feedback gained from user testing and fix the recorded bugs that came up during testing. Another aim was to redeploy the app onto the ‘Base’ blockchain, a faster and cheaper blockchain.

Changing APIs

After some user testing, I realised there were some problems with the third-party APIs I was using in the app. The API to get a user's recent transactions was not getting the complete list of transactions. The list of transactions contained all transactions the user had initiated; for example, if the user sent money to another user, it was in the list; however, if the user received money from another user, it was not in the list. This was a problem as users noticed that some of their transactions were not in the UI during user testing.

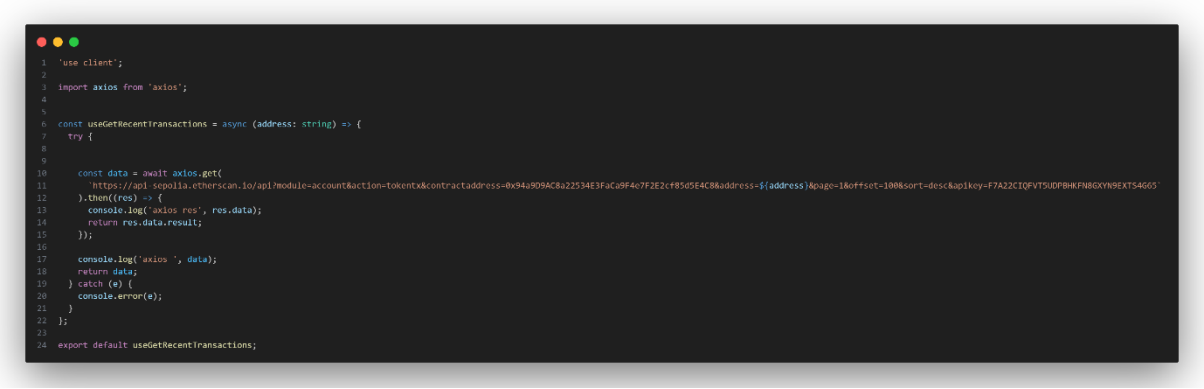
This led me to change from implementing the ‘Alchemy SDK’ from a previous sprint to using an API provided by ‘Etherscan’, another blockchain API company. Before using Etherscan, I also looked at using another blockchain API service by Moralis. However, the setup process was not very clear in their documentation. Etherscan was chosen after some testing with their API endpoint in insomnia. I tested the endpoint with a test blockchain address used in

development. Checking the response from Etherscan (Figure 86 Etherscan Recent Transactions API response), I confirmed that all the users' recent transactions were in the array and that the endpoint could be used in the app.

```
{
  "status": "1",
  "message": "OK",
  "result": [
    {
      "blockNumber": "5451942",
      "timeStamp": "1710021312",
      "hash": "0x065c60f24e663498b3012f3766e9efaf61de5c959dbecc0dc70041622b5bc1d4",
      "nonce": "1409",
      "blockHash": "0x4ec2762efdd92c88ff1a2d985776d884a6dad692f0f3794809d59ee57f58e162",
      "from": "0x6b3c5debb67505dfd66f3b3b80d1d24df8da886d",
      "contractAddress": "0x94a9d9ac8a22534e3faca9f4e7f2e2cf85d5e4c8",
      "to": "0x2a09f3d902d8151340dc6098707da60ad5a1f589",
      "value": "15000000",
      "tokenName": "USDC",
      "tokenSymbol": "USDC",
      "tokenDecimal": "6",
      "transactionIndex": "59",
      "gas": "238573",
      "gasPrice": "1110251748",
      "gasUsed": "140824",
      "cumulativeGasUsed": "10511039",
      "input": "deprecated",
      "confirmations": "121560"
    },
    {
      "blockNumber": "5451958",
      "timeStamp": "1710021504",
      "hash": "0x701133e586c8d2c4d3c3aa60164fd286c7d366221f4c928f7fda264af0bf5a0d",
      "nonce": "2402",
      "blockHash": "0xfebf4cd5affaa11de70200299f67e13314447f9de406fe11d6a5c5d5b746cd7d",
      "from": "0x2a09f3d902d8151340dc6098707da60ad5a1f589",
      "contractAddress": "0x94a9d9ac8a22534e3faca9f4e7f2e2cf85d5e4c8",
      "to": "0xa9b3191b07a317744d6eb9b65eec291c51c90f13",
      "value": "2000000",
      "tokenName": "USDC",
      "tokenSymbol": "USDC",
      "tokenDecimal": "6",
      "transactionIndex": "43",
      "gas": "567442",
```


Figure 86 Etherscan Recent Transactions API response

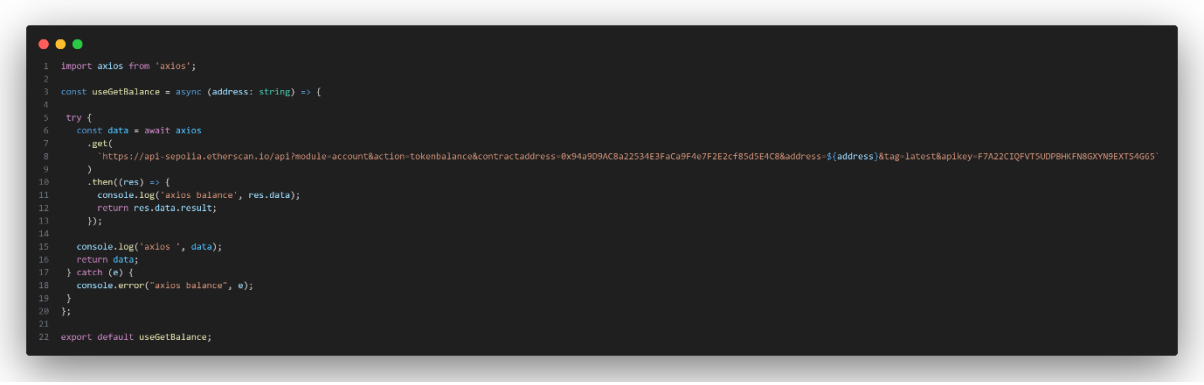
The Etherscan endpoint could then be used in my 'useGetRecentTransactions' hook (Figure 87 'useGetRecentTransactions' hook using Etherscan), replacing the previous code using 'Alchemy'. Using Axios, a package that helps a client consume APIs, I could then get all the user's recent transactions every time the hook is called.



```
'use client';

import axios from 'axios';


const useGetRecentTransactions = async (address: string) => {
  try {


    const data = await axios.get(
      `https://api-sepolia.etherscan.io/api?module=account&action=tokentx&contractaddress=0x94a9D9AC8a22534E3FaCa9F4e7F2E2cf85d5E4C8&address=${address}&page=1&offset=100&sort=desc&apikey=F7A22CIQFVT5UDPBHKFN8GXYN9EXTS4GG5`
    ).then((res) => {
      console.log('axios res', res.data);
      return res.data.result;
    });

    console.log('axios ', data);
    return data;
  } catch (e) {
    console.error(e);
  }
};

export default useGetRecentTransactions;
```

Figure 87 'useGetRecentTransactions' hook using Etherscan

Another API that did not work as intended after user testing was the API responsible for getting a user's balance (Figure 88 'useGetBalance' hook using Etherscan). This API used the 'Wagmi' library in my 'useGetBalance' hook from a previous sprint. I realised when testing that the user's balance was slow to update, often taking several minutes. This was because the Wagmi API was quite slow in fetching the data. The slow API annoyed users, so I switched from Wagmi to Etherscan. Etherscan was chosen again to replace the current API to get a user's balance, as I was happy with the endpoint that got the user's recent transactions. I followed a similar process, first testing the API in Insomnia. Then I went to the app, sent a money transfer, and called the API to see if it was fast to notice the user's updated balance. After confirming the API was fast enough, I replaced the code in my 'useGetBalance' hook again using Axios and Etherscan's endpoint.



```
import axios from 'axios';

const useGetBalance = async (address: string) => {

 try {
   const data = await axios
     .get(
       `https://api-sepolia.etherscan.io/api?module=account&action=tokenbalance&contractaddress=0x94a9D9AC8a22534E3FaCa9F4e7F2E2cf85d5E4C8&address=${address}&tag=latest&apikey=F7A22CIQFVT5UDPBHKFN8GXYN9EXTS4GG5`
     )
     .then((res) => {
       console.log('axios balance', res.data);
       return res.data.result;
     });

   console.log('axios ', data);
   return data;
 } catch (e) {
   console.error("axios balance", e);
 }
};

export default useGetBalance;
```

Figure 88 'useGetBalance' hook using Etherscan

Changing blockchain

The app was initially built and deployed on the Sepolia blockchain, a test network for the Ethereum blockchain. Sepolia was chosen due to its widespread support across the blockchain ecosystem, great infrastructure, and community backing.

However, the Ethereum blockchain is very slow in practice, with an average of only 13 tps (transaction per second), according to 'L2Beat' (L2Beat, 2024), an analytics company. If the

app were to be deployed to the 'Ethereum' blockchain after being tested on 'Sepolia', the throughput of the user's transactions would be very slow.

After some consideration, the 'Base' blockchain was chosen for the app to be redeployed. At the time of writing, 'Base' can do a throughput of 35 tps, with the second closest being 'Arbitrum', a competitor doing 16 tps according to L2Beat. Base may also sponsor all blockchain transaction fees for the app's users as the app is built using ZeroDev SDK who has a partnership with 'Base'. By switching to using the 'Base' blockchain in a live environment, users' transactions would be faster and cheaper, making a better UX.

https://twitter.com/zerodev_app/status/1761038219632857558

To change which blockchain the app is deployed to, I went into my code and changed all the SDKs and APIs configuration to target 'Base' instead of 'Sepolia'. The change was straight forward to implement as it is common for blockchain based apps to support multiple blockchains. I then got some test USDC from a faucet run by the 'Base' developers so I could test the app.

To ensure everything was working correctly, I reinstalled the app and signed in. I then checked that transfers were being sent and that all data was being retrieved from 'Base' instead of 'Sepolia'. When sending transfers on the 'Base' test network, transactions were noticeably faster, usually taking 20 seconds to transfer on 'Sepolia' and now 10 seconds on 'Base'. In production, transfers on 'Base' should feel instant to users. The transfer details could then be confirmed on the Base blockchain using the Base blockchain explorer (Figure 89 A test transfer done on the 'Base' blockchain).

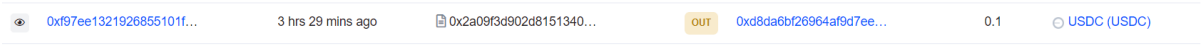


Figure 89 A test transfer done on the 'Base' blockchain

Conclusion

This chapter discusses the steps taken to implement and develop the payments app. It describes the requirements gathering, design, implementation, and testing sprints. The SDKs, frameworks, and technologies used in the app are described in detail, explaining why they are used and how they were implemented. Each feature and component and its functionality are described, as are the iterations of these components.

Testing

Introduction

This chapter describes the testing that took place of the app. Several tests will be conducted, including Unit testing to ensure all functions are working as expected in isolation, User testing to improve the UX, API endpoint testing for the backend, TypeScript testing, authentication testing for specific routes, manual functional testing, and multiple device testing. All test results will be recorded for examination with the intent of improving the application, checking if everything is working as intended, and finding any uncaught bugs.

Reddit question

During my interim presentation, my supervisor recommended that I reach out to blockchain developers to see if there are any potential security vulnerabilities in the way my application is set up.

Taking this advice, I joined the Reddit page ‘r/ethdev’, which is a community dedicated to Ethereum blockchain development. I asked the question ‘Question about storing encrypted private keys in local storage’ (Figure 90 Reddit question). This question was asked to figure out if the method I had implemented was secure. Every time the user wanted to sign in or send money, a signature from the private key was necessary. My implementation encrypted and stored the user's private key in the device's local storage. I wanted to check if there were issues with this implementation.

I received comments from several users. Some were not relevant to the question as they didn’t quite understand my implementation. One unuseful comment was from a user who suggested storing it in a database, which would not be a good idea as if the database was compromised, all users' accounts could be compromised.
I received two useful pieces of feedback. The first user said, “The key is derived from properties of the browser which are not secret, and a user key that gets baked into the built react app. Anyone with access to the session storage can easily decrypt the data”. The user criticised the implementation of encryption that I was using. I had implemented the encryption using a library ‘react-secure-storage’. The user described security vulnerabilities with this package, including problems with where the keys are stored, reliance on the browser, and the encryption key, which is an environment variable in the Next JS app. This feedback was useful as these are vulnerabilities that I was not aware of. I then asked the user for further suggestions, and they suggested implementing a password-based login system in which a unique encryption key could be derived from the user's password.

The second useful piece of advice was from another user who suggested moving away from my implementation completely and starting again using a login provider which they suggested ‘Privy’. I had originally planned to use privy when starting this application. However, it wasn’t possible as the library was in private development, and developers had to request access to use the library. However, since this user recommended using it, I had a second look to see if the access had become public. Since Privy became public, I started to implement the library in replacement of my previous implementation using Web3Auth. Ideally, I could have used Privy from the beginning, which would have saved time and provided a better login flow for the app.

Figure 90 Reddit question

Unit Testing

As part of the unit testing, the application was tested in several different ways. It was important to test each feature of the application in isolation to find bugs, and ensure all features are working correctly.

Jest

To achieve unit testing in my application, I integrated a testing library named ‘Jest’. As I am using Next JS as my frontend framework, it was easy to set up, as Next JS has great support for Jest. In the Next JS documentation, Jest is recommended as the testing library to use. To implement Jest, it is first installed as a package using ‘npm’. In my ‘package.json’ file, I then added a script named ‘test’ (Figure 91 Jest test script in the package.json file).

Figure 91 Jest test script in the package.json file

Next, a file named ‘jest.config.js’ was created and edited to set up my Jest configuration. The configuration was copied from the Next JS testing documentation and set up using their recommendations. Then, a test folder was created (Figure 92 Jest tests folder), where all the pages to test would be created. The naming convention for a test file is the filename followed by ‘test.jsx’.

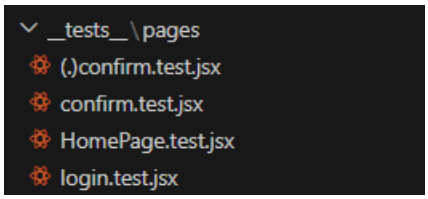


Figure 92 Jest tests folder

This test below (Figure 93 Jest HomePage test) will check to see if the ‘HomePage’ component is rendered. It can then be run using the command ‘npm run test’. If the page is rendered, the test will return a success beside the ‘HomePage’ test in the terminal.

```
1  import { render } from '@testing-library/react';
2
3  import Page from '@/app/home/page';
4
5  describe('Homepage', () => {
6    it('renders the Components', () => {
7      render(<Page />);
8
9    });
10 });
11
```

Figure 93 Jest HomePage test

This process should be repeated for every page in the application to check that they are all working as intended. Jest is also used to test UI components, such as buttons, headings, dialogs, or functions that are used in the application. To set up these tests, the same process is followed for testing the application's pages.

Insomnia API Testing

When testing the API, I used Insomnia to ensure that all routes worked as intended and that the JSON data was being served in the correct format. I had two routes to test in insomnia: a

‘GET’ request and a ‘POST’ request (Figure 94 Insomnia requests). I first entered the endpoint in Insomnia, followed by the path and the user's blockchain address.

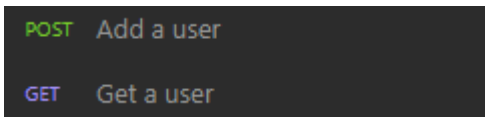


Figure 94 Insomnia requests

When the GET request is performed (Figure 95 Insomnia GET request), the expected status of the response should be ‘200’. If the response is 200, then JSON data containing the user's data (Figure 96 Insomnia GET request response), including their address and subscription object, should also be sent from the server to the client.

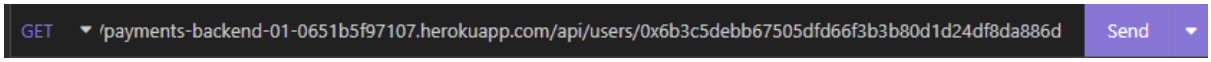


Figure 95 Insomnia GET request

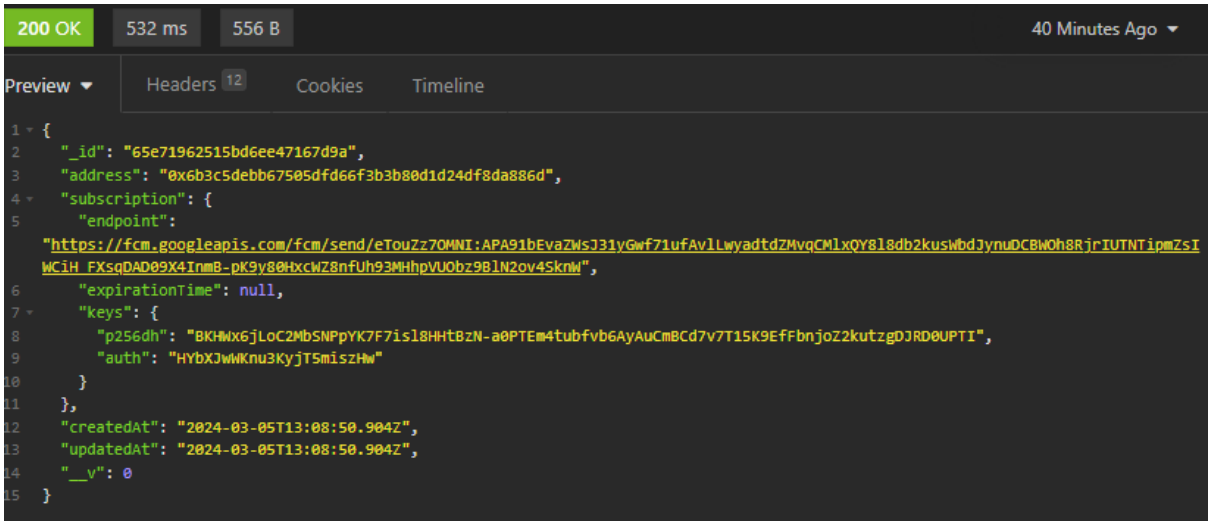


Figure 96 Insomnia GET request response

For the POST request to be sent, the user's address and subscription object is required. If the POST request was successful, a status code of ‘201’ and the new user's JSON data should be returned.

If the API POST request was successful, the Mongo DB database can be checked to see if the user's data has been entered (Figure 97 Mongo DB post request data).

Figure 97 Mongo DB post request data

Also, in the Alchemy dashboard, the admin can check to see if the user's address was added to the address dashboard (Figure 98 Alchemy address dashboard).

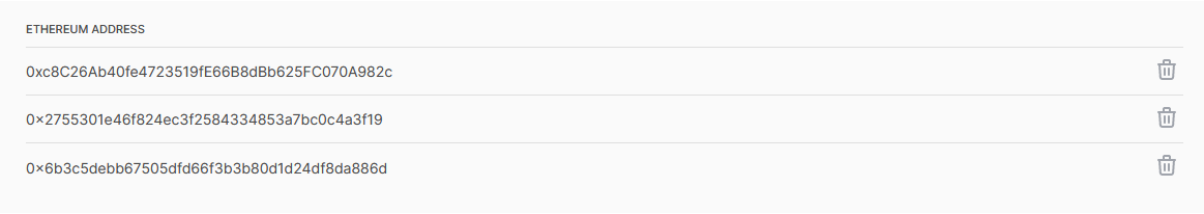


Figure 98 Alchemy address dashboard

The routes were tested and added results were recorded in a table (Table 4 API route testing).

Table 4 API route testing

Test ID	API Route Testing		**Priority**		HIGH
Test Description	To test if the API routes work as intended				
Pre-Requisites	Insomnia		**Post-Requisite**		
	Request Method	**Endpoint**	**Expected Output**	**Actual Output**	**Result**	**Comments**
1	GET	/users	Returns the user data in JSON format	Returns the user data in JSON format	Pass	
2	POST	/users	Users' data is entered into the database	Users' data is entered into the database	Pass	

With more time to develop the API, I would add Swagger documentation (Swagger, 2024) to test the API endpoints better. Swagger provides a UI that describes each endpoint, provides example inputs and outputs for each endpoint, and allows developers to test their endpoints through the UI.

Typescript testing

My application was written with the intent to use TypeScript so testing can be done to ensure that every type is working as intended. However, as the project grew, support for TypeScript lagged in the workflow, and some functions and variables did not have the correct type declarations.

When building a Next JS application and preparing it for deployment, Next JS checks that all variables and functions have been assigned a type. This reduces unexpected errors in the production environment. Developers often love TypeScript for this reason, as it can reduce bugs in the code. However, a lot more work has to be done to create types for every variable and function in the project.

In my project configuration, I turned on ‘ignoreBuildErrors’ for the TypeScript setting, which would allow for a production build without having types implemented.

In the future, it would be nice to have types written in my project, but unfortunately, I didn’t have time. If I did keep up writing in TypeScript, I would test it using the Next JS build test and the command ‘npm run typecheck’ (Figure 99 npm typecheck script in package.json file), which will also perform a check on all types in your application.

Figure 99 npm typecheck script in package.json file

Testing the app on multiple devices

The application was mainly designed and implemented for iPhone and Chrome browsers. Ideally, it works on many supported devices so that testing can be carried out on multiple browsers and devices. As the application was mainly designed and implemented for iPhone and Chrome PC, it was important to test it on other devices, such as Android phones.

To test the app on Android devices, Android Studio was installed on a computer. Within Android Studio, Android emulators are available to test the application on many Android devices. Once the emulator was set up, the app was installed on the phone through the Chrome browser. A Google Pixel 7 was chosen for the emulator.

A set of tests was lined up to check if the app's core functionality worked as intended on the emulator and recorded in a table (Table 5 Android Testing). First, it was important to test if every route was rendered as intended. On the emulator, I navigated to every route in the app to test if the page and its children's components were rendered as expected (Figure 100 Routes working as intended on an Android Emulator). While testing, I realised there were some design bugs where colours and certain CSS-positioned elements were not being displayed as intended. This was because the Android device uses Chrome, so the CSS was interpreted differently from how I expected it compared to IOS devices.

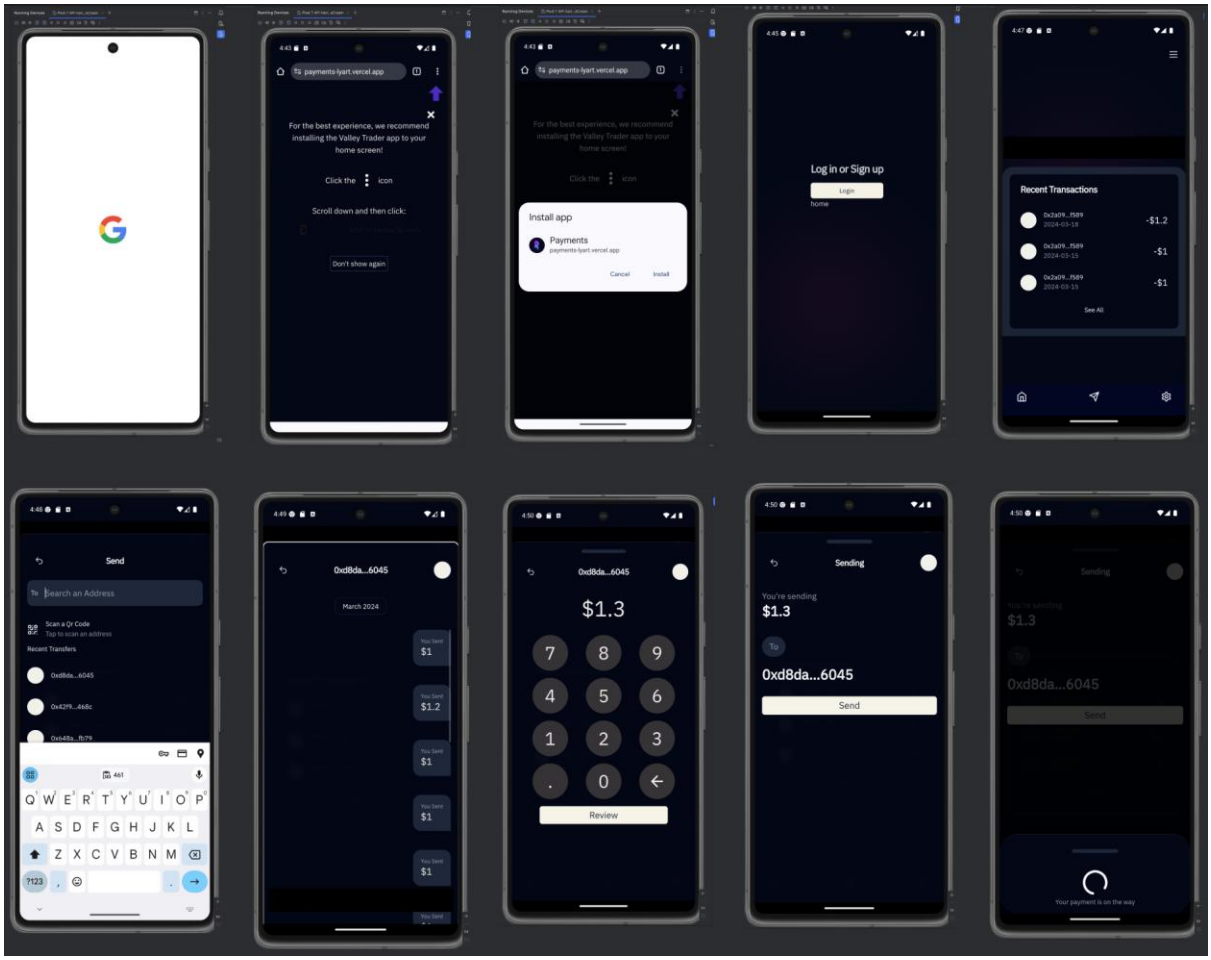


Figure 100 Routes working as intended on an Android Emulator

It was also important to check that the QR code was generated and could be decoded on Android. I navigated to the route containing the QR code, and the code was generated successfully. I then checked on another device's camera to see if the QR code could be decoded with the user's address as the value. The test was successful.

I then tested to see if the Android camera could decode a QR code on another device.

Notifications were also tested on Android. When logged into the Android devices, I subscribed to push notifications. Then, I sent money to the account and waited to receive the push notification on the device. The notification was received successfully.

Next, it was important to test the app's core feature, sending money. An amount was inputted through the UI, and a payee was selected. The amount was successfully sent and could be confirmed by checking the blockchain explorer ‘Etherscan’ with the transaction hash (Figure 101 Confirmed transaction hash on Etherscan).

Txn Hash	Method	Block	Age	From		To	Quantity
0x7c242c03ffe...	Handle Ops	5539072	2 days ago	0x2a09F3d9...ad5a1F589	OUT	0xd8dA6BF2...37aA96045	1.3

Figure 101 Confirmed transaction hash on Etherscan

Table 5 Android Testing

Test ID	Android Testing	**Priority**	HIGH
Test Description	Test if the app can be installed and used to send and receive money.		
Pre-Requisites	Android mobile device or emulator	**Post-Requisite**	

	Action	**Inputs**	**Expected Output**	**Actual Output**	**Result**	**Comments**
1	App can be installed	Add to home screen	App is installed	App is installed	Pass	
2	Navigate to each route	Navigate to each route through the UI	Each page and component render successfully	Each page and component render successfully	Pass	
3	QR code can be decoded	QR code is generated	Generate QR code	Generate QR Code	Pass	
4	Send	Input amount and payee	Input amount is sent to payee.	Input amount is sent to payee.	Pass	
5	QR code scanner opens	Camera is opened through the QR code scanner	Camera opens and scans	Camera opens and scans	Pass	

Authentication Testing

To test authentication, I checked all routes that were wrapped in an authentication component and visited those routes in the app. If the route passed or failed, it was recorded in a table (Table 6 Authentication testing table).

Table 6 Authentication testing table

Test ID	Route authentication testing	**Priority**	HIGH
Test Description	To test if a user can access authenticated routes		
Pre-Requisites	App installed on phone	**Post-Requisite**	

	Route	**Authentication Required**	**Authentication Successful**		**Result**	**Comments**
1	Home	Yes	Yes		Pass	
2	Transactions	Yes	Yes		Pass	
3	Search	Yes	Yes		Pass	

4	Send	Yes	Yes		Pass	
5	Tx	Yes	Yes		Pass	
6	Transactions	Yes	Yes		Pass	
7	Menu	Yes	Yes		Pass	
8	Scanner	Yes	Yes		Pass	
9	Receive	Yes	Yes		Pass	
10	Payee	Yes	Yes		Pass	
11	Login	No	Yes		Pass	Login is not an authenticated route

Manual Functional Testing

The app's functionality was tested through the app front end using an iPhone. An iPhone was required for installing and running the app, and a Google account was required for logging in. The test tested a user's ability to log in, receive money, and send money to another user. The test results were recorded in a table (Table 7 Manual Functional Testing table).

Table 7 Manual Functional Testing table

Test ID	Sign in and send	**Priority**	HIGH
Test Description	Test if a user can sign in and send money to another user.		
Pre-Requisites	iPhone, Google account	**Post-Requisite**	

	Action	**Inputs**	**Expected Output**	**Actual Output**	**Result**	**Comments**
1	User navigates to the website	https://payments-lyart.vercel.app/	User is prompted to install the app	User is prompted to install the app	Pass	

2	Install app	Click Add to Home Screen	App is installed on the user's iPhone	App is installed on the user's iPhone	Pass	
3	Click 'Log in' button	Click	Google auth login page is displayed	Google auth login page is displayed	Pass	
4	Click 'Receive' button	Click	User's QR code is displayed	User's QR code is displayed	Pass	
5	Navigate back to 'home page'	Click	User navigates back to 'home page'	User navigates back to 'home page'	Pass	
6	Check balance	View	User views their updated balance	User views their updated balance	Pass	
7	Search for a payee	Click	User clicks on 'Send' button	User clicks on 'Send' button	Pass	
8	Scan a QR code	Scan	User scans a QR code with their camera in the app	User scans a QR code with their camera in the app	Pass	
9	Input an amount to send	Input	User inputs a value to send to a payee	User inputs a value to send to a payee	Pass	
10	Confirm payment	Click	User confirms the payment amount	User confirms the payment amount	Pass	
11	View the completed payment	View	User views the completed payment receipt	User views the completed payment receipt	Pass	

User Testing

Each task the user was presented with was based on the manual functional testing tests. The goal of testing with this user was to see if the features worked as intended on a new device with a live user. The developer also wanted to see if all routes and UI components worked as intended. As the test was carried out, the developer took some notes to record the user's results.

User 1

Description

The first user was chosen as they were a frequent user of Revolut. They were asked why they use Revolut and what purpose they use it for. They said, “Because everyone uses it”, further explaining that they use Revolut as all their friends and family use it, suggesting that Revolut has had a lot of network effects. They said that they mainly use Revolut to transfer money to friends and family, and they never use it for other features such as investing, saving, and lifestyle.

Tasks

The user was successfully able to download and log in using their Google account. The user was initially confused by their blockchain address displayed on their homepage, beside their balance and username. They didn’t understand what it was. The user was asked to navigate to settings to turn on push notifications. However, the subscribe to push notifications button did not work. The user was then asked to receive money by clicking on the receive button. After being sent money by the developer, the user received the money. However, their updated balance was not shown in the UI. The user was confused as they had to close the app and reopen it for their balance to be updated. Then, the user was asked to perform a transfer. The user went to open the send page and was asked to open the camera QR code scanner. The user was successfully able to open the camera and said the process was smoother than the previous tasks. Then, the user was asked to scan the developer's QR code to send money to them. The QR code successfully routed the user to the send page. The user then became confused about what they were sending, whether they were sending real money from their bank account or cryptocurrency, and where they were sending it from. The user was able to input an amount to send successfully using the keypad. Then, the user was able to review the payment that was about to be sent. The user liked this feature as they could confirm the amount and payee they wanted to send to. The user, however, got confused as the payee was a long blockchain address rather than a username. The user then pressed send, and the transfer was successful, and the money was sent. However, after the payment was sent, the app crashed due to a bug known to the developer in the UI which confused the user as they thought they had done something wrong in the task. After restarting the app to fix the bug temporarily, the user was then asked to view the recent payment they just made. When viewing the payment history with the payee, another bug known to developers was present where the user’s transaction history was not being displayed correctly, and some transactions were missing from the history.

Analysis

Installation

The user found the app easy to install, but the process was unfamiliar compared to installing apps through the official Apple App Store. Unfortunately, as the application is a progressive web app, nothing can be done to list it on the App Store, as Apple does not support this. However, some changes can be added to the installation to provide more clear instructions on why the app had to be downloaded as a progressive web app.

Confusing blockchain address

As the blockchain address confused the user, perhaps some UI feedback explaining what the address is would be appropriate. After the login process, a tutorial screen that explains how to use the app and the relevant terminology could be shown to the user to make them more comfortable with the app. A tutorial screen is often quite common after the installation of an app, as seen in both Revolut and Uniswap Wallet.

Fixing the user balance not updating

The user's balance didn't update when receiving money, as this feature had not been added yet. Two common approaches can be taken to fix this. An API can be developed to push a request to the user's client to update the balance, or a drag-down feature to refresh the screen could be implemented. This drag-down feature is often seen in social media apps to get the latest posts or in payment apps to update the user's device with the latest data.

Confusing transfer

The user was confused about what currency they were sending and whether it was live or a test, which could have been explained in the tutorial screen mentioned previously. It is possible that the users didn't understand the transfer because they were not explained enough before using the app. The user could also have been explained more about the app and its functionality before the test began to make sure the user understood the payment process.

Fixing known bugs

Several of the bugs encountered by the user in the app were already known to the developer and were in the backlog of features to be fixed. Bugs such as the balance not updating, seeing the wrong recent transactions with the payee, and seeing their own account in recent payments were recorded and checked to see if they were in the backlog of features. The push notifications bug was not known to the developer and was recorded for debugging. It seems that IOS was blocking popups on iPhones. The 'Popups Enabled' setting in Safari was required for the subscribe to notifications functionality to work.

User 2

Description

After fixing the known bugs by from the previous test, a second test was carried out with another frequent Revolut user. Since the last test, many UI features were pushed to the app, such as making it easier to add a contact, which removes the confusing long Payee name. The bug that made the app crash after sending a transaction was also fixed. The bug where a user saw the wrong recent payments with a user was also fixed.

The user was briefed more than the first test user about what a progressive web app is and why it has to be installed through their Safari browser. They were told that the money was just a test cryptocurrency and was meant to mimic dollars in a bank account. Also, they were told that the blockchain address on their homepage was their unique ID, which they could use to receive money, which they said made sense to them.

Tasks

The user followed the same tasks as the first user and successfully completed them. After completing the tasks, the user was asked about their experience. The user said that after sending a payment, they would like to be shown a confirmation “receipt page” of their transaction. This feature was already built into the app but required a small line of code to redirect the user to the receipt page.

User 3

Description

The third test user was someone who had never used Revolut or similar fintech payment apps before but had used Irish banking apps such as Bank of Ireland or AIB. As they were not users of Revolut or similar apps, extra briefing took place. The briefing included an explanation of how to use the app and what tasks they would be asked to complete to make sure the user understood what a payments app was meant for. This user required more guidance through the app than the previous users.

Tasks

The user installed the app with no issues and thought the Google login process was simple. They were then asked to repeat the same tasks as the previous two users, such as receiving payment via QR code, checking their balance, adding a contact, sending money to their new contact, and checking the recent transfer to the contact. The user successfully completed each task assigned to them and didn’t encounter any UI bugs during the test.

The user was asked how they thought they would have found the app without being briefed by the developer, replying that they would have been confused by its purpose at first. The user was asked how useful they thought a tutorial screen would be, to which they responded, “Very useful”.

Conclusion

Several types of tests were conducted and described in this chapter. First, a question was asked to a blockchain developer community to test the application implementation and security practices. The application backend API endpoints were tested to see if the correct JSON data was being served. TypeScript testing was outlined and why it was important for the app. Authentication testing was carried out to make sure that several routes implemented authentication correctly. Unit testing was carried out by using the Jest framework and Next JS to test if several of the application's components and pages were rendered correctly. Both manual functional and multiple device testing were carried out by the developer, and results were recorded. User testing was conducted with several users, with the intent to find and fix any potential bugs missed by the developer and to improve the UX in the app. During all testing phases, bugs were recorded and added to a backlog of features to be fixed and developed.

Project Management

Introduction

This chapter describes the process of how the project was managed, the steps taken to develop the project, and the tools and technologies used during the development process.

Project Phases

This section describes the phases of the project and what took place within each phase.

Proposal

During the project proposal, an initial outline for the application was created. The purpose of the app was defined, and similar applications were presented, along with where the idea for the project came from.

Some desired features were picked, and several tools to build the app were listed. Some brief research was done to collect a list of technologies that would potentially be used to build the app, such as React Native, Expo, Account Abstraction, and Ethers JS. Several account abstraction SDKs, such as Biconomy, ZeroDev, and Alchemy, were examined. Several approaches for building an app were examined, such as React Native, React Native with Expo, and a Progressive Web App. Several problems with similar applications were stated, including security and UX features. Then, several points were listed regarding how the proposed app could fix the problems.

Requirements

In the requirements phase, several similar applications were examined to gather the required features needed to build a payments app. Two similar applications, Uniswap Wallet, a blockchain-based wallet, and Revolut, a fintech payments app, were examined for their functionality, UI, and usability. Several key advantages and disadvantages of each app were recorded.

Two fictional personas were created based on the examined apps to better understand what a user wants from an application and why the user would use it.

Several interviews were conducted to study the usability of these apps, including task analysis to find out how users interacted and navigated within the apps and usability questions, where interviewees were asked about the usability of the apps.

A use case diagram was developed to understand how users would use the app for both sending and receiving money. Finally, a feasibility study was conducted to examine what technologies would be used to create the payments app.

Design

In the design phase of the project, the technologies being used for the project were listed, and their use case within the project was described. The structure of the app was outlined, and why this structure was chosen was explained. The applications architecture and database design ERD were designed to understand better how the application will work. A range of wireframes from low fidelity to high fidelity was designed to have a reference when implementing the design. A user flow diagram was designed to understand better how a user will navigate through the app and what screens would need to be shown to the user. Several design choices were explained, such as font family, colour scheme and typography scale.

Initially, several React Native example repositories from companies such as Alchemy, Biconomy, and ZeroDev, along with others, were examined to see if a React Native app was feasible. After having no success with React Native examples, I chose to develop a PWA instead. Several examples of PWA GitHub repositories were examined, along with how blockchain technology could be implemented in a PWA. Frontend frameworks such as React, Next JS, and Redux were researched to understand better the structure of a PWA and how a PWA blockchain app could be built. Several backend libraries were examined, such as Viem, ZeroDev and Wagmi React Hooks, to understand how the users's account creation would happen, and how they would interact with the blockchain to send money. Research was taken to make sure that the backend libraries were all compatible with each other.

Several wireframes were drawn by hand to get a basic idea of how the app would look. These wireframes were based on the requirements gathered from similar applications studied. The low-fidelity hand-drawn wireframes were then redesigned in Figma with the intent of building on them to develop a high-fidelity prototype design. In Figma, several frames were developed to store each stage of the wireframe. In Figma, some basic prototyping was created to test how a user would navigate from screen to screen to understand the flow of the app better.

Based on the wireframes and Figma flow prototype, a user flow diagram was designed. This diagram would be used when developing the app to remember where each screen should flow to next in the app.

Next, I chose colours, typography, and a font family and applied them to my Figma wireframes to continue building a high-fidelity design. These colours, typography, and font family were based on Tailwind, a CSS framework. As Tailwind would be used in the app's implementation, it made sense to design the app with the same system.

Implementation

The application was implemented using scrum methodology in two weeklong sprints focusing on design and implementation. First, the design sprint, followed by the implementation sprint, followed by another design and another implementation sprint. Each sprint had several development goals to implement. For example, in sprint one, the login functionality was implemented. The sprint goals were designed around features in the app that were ranked from high to low priority to be implemented. First creating the required screens, then login functionality, payment functionality, and getting the user's data logic. Then, once the high-priority features were implemented, other features could be added, such as push notifications, adding a contact, and another UI component.

During the implementation phase, as the developer became aware of better ways to implement certain features, several technologies were dropped from the project, and others were added to the project in replacement. When developing the login functionality, initially, it was created with ‘Web3Auth’ but was later switched to use ’Privy’ instead for enhanced UX and security. Switching to Privy required redesigning and changing the payment and login logic, but in the long run, it made the code more reliable and easier to understand. During later implementation sprints, the npm package used to implement the PWA technology was changed to allow features such as push notifications. This required additional research to find a suitable package for implementing push notifications. Towards the end of the project, after some user testing, the developer was made aware that certain APIs were not working as expected and had to be changed to work with the app in the intended way.

Several design choices were also made during the design sprints. Rather than developing a component library from scratch using Tailwind, ShadCN and Aceternity were added to the project to develop a component library and speed up the design process.

During implementation, a list of bugs to fix was kept to track what had to be fixed.

Testing

Several tests were carried out in the project's testing phase. First, the project supervisors recommended reaching out to blockchain developers and asking them about the app's security, reliability, and potential flaws. This test was very useful, as several vulnerabilities were pointed out and fixed after reaching out to a blockchain development community on Reddit.

The next API testing was carried out with Insomnia to ensure that all the developed endpoints were working as expected.

Typescript testing was carried out but was expected to fail as the TypeScript had not been kept up throughout the project.

Testing was carried out on multiple devices to check that the app worked and the UI looked as intended on devices such as Chrome browsers, Android phones, and iOS devices. During testing, Android Studio was set up to emulate an Android device. The test went through several individual tests on the Android device to check that the app's features worked as intended. Each test result was recorded in a table.

Next, basic Unit testing was carried out using Jest, a JavaScript testing framework. Although the Jest tests worked as intended, they were not as thorough, as more time was needed to check every function, page, and component.

The developer carried out manual functional testing. During the testing, several tests, such as logging in, transferring money, and others, were carried out, and the test results were recorded in a table for examination. These test results were important to find any unexpected bugs in the software.

Usability testing was carried out on several users. Users were asked to complete several tasks while the developer watched and recorded the results. These tests were very useful as the developer was able to catch bugs that the users ran into and record them to be fixed. Also, the developer was able to receive feedback on features the users desired in the app.

SCRUM Methodology

In total, there were eight sprints, each lasting two weeks. Each sprint had specific goals to be completed, all with the intention of improving the app. At times, some goals were done in parallel. While working on the login logic, I was also working partially on the sending logic, or at least trying to code in a way that the functionality and logic or design system could be reused in later sprints.

These sprints worked well as key objectives, goals, and priorities were set before adding a new feature.

If a goal was not completed or a bug interrupted it, it was often recorded, and if not easily fixed, another goal was started so as not to get hung up on difficult-to-fix features for long periods. A backlog of features added was recorded after each sprint.

Project Management Tools

Trello

A Trello board (Figure 102 Trello board) was maintained during the sprints to keep track of what features had been added and what week they had been added. This was helpful in documenting the process of developing the app. I created a Kanban board to record the app's development process. A list was created for each development week, and a to-do list was created to record features to be added. As a feature was complete, it was removed from the to-do list and added to the list with the week it was complete, along with any relevant screenshots of code or designs. Another list was created to record bugs in the application that needed fixing.

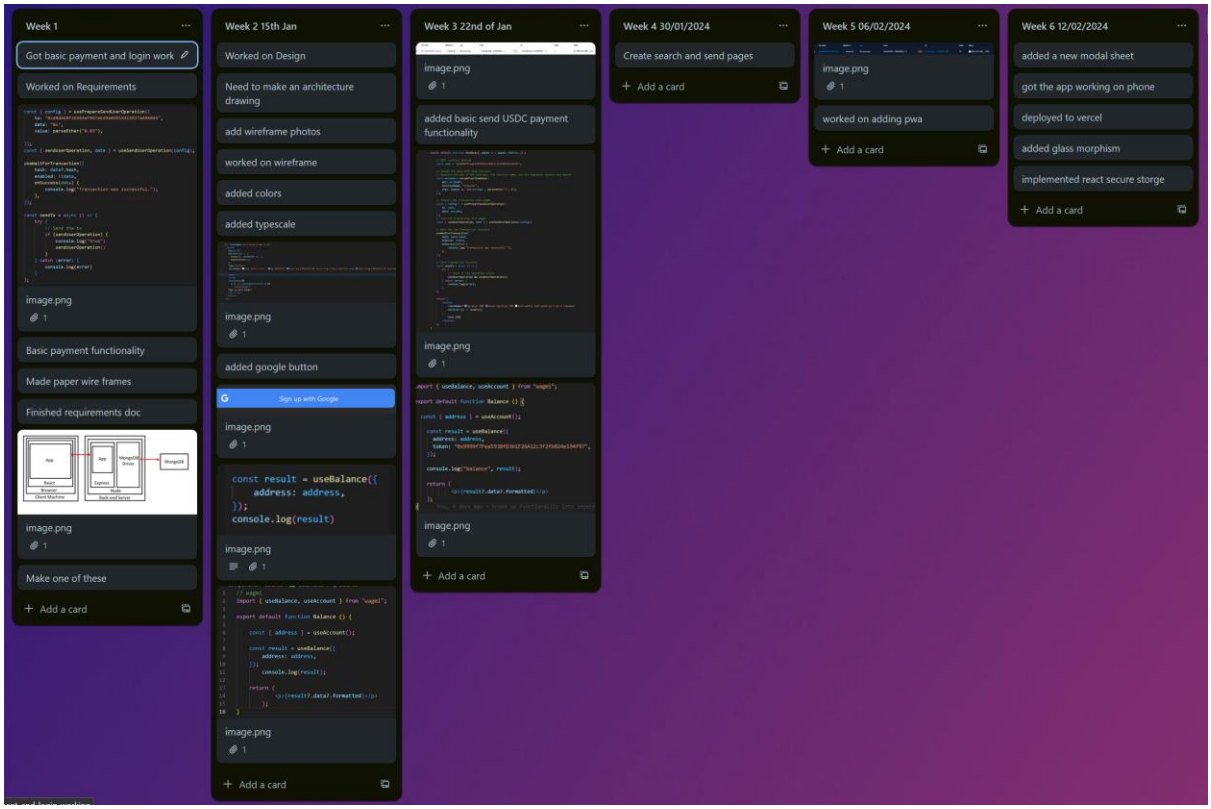


Figure 102 Trello board

GitHub

GitHub was used to store the code in a remote repository. Additionally, GitHub provided a useful UI (Figure 103 GitHub UI displaying the project's commits) to visualise recent commits, merges, and branches. Git was to manage the app and version control. While working on the app, as features were added, removed, or completed, they were committed to the repository with a message to record what the changes made were. If a big feature was being added to the app, a new branch was created so as not to break the current working code. The new branch would be worked on, and then when everything was completed and working as expected, the branch was merged into the main branch and deployed. Two git repositories were created for the app, one for the client-side user interface and another for the backend API, which stores all the code for sending notifications and listening for users' blockchain transactions.

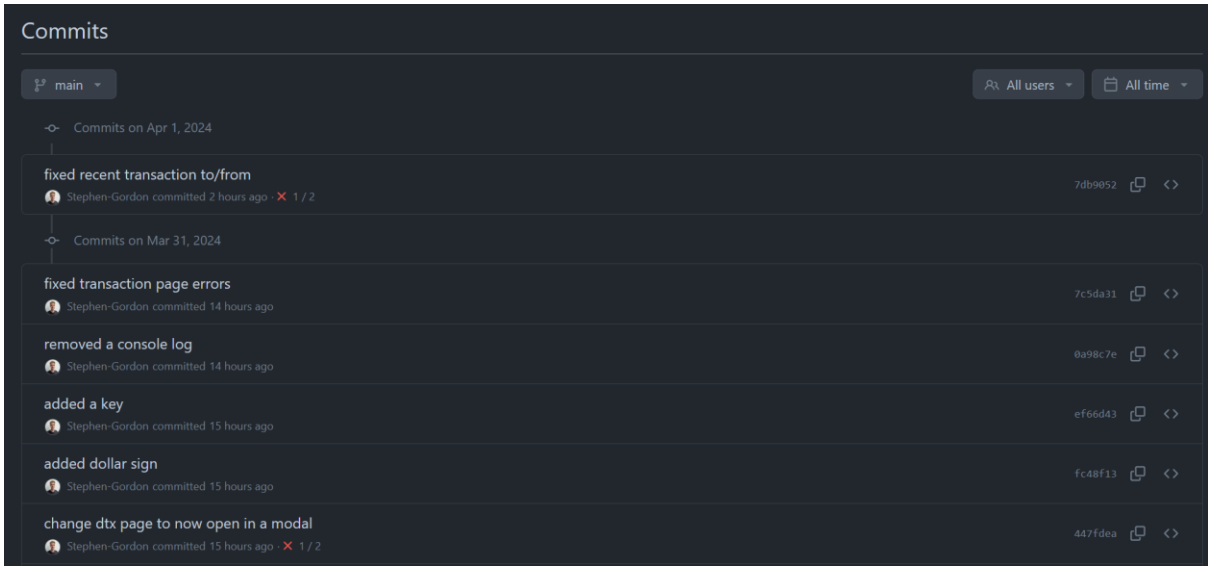


Figure 103 GitHub UI displaying the project's commits

Vercel

Vercel was used to manage the app's deployment. Vercel provides a UI (Figure 104 Vercel UI showing the app's recent deployments) to deploy your app straight from your repository in GitHub by connecting your GitHub account. As the app was built with Next JS, which is compatible with Vercel, this streamlined the deployment process. When specific git branches were created to test specific features in the app, Vercel created a hosted deployment of that branch and provided development and testing tools while keeping that deployed branch separate from the production-ready branch. In the future, this would be useful for testing features live without affecting app users. Vercel also showed any build or TypeScript errors in the app in their logs. Vercel provides useful web analytics for the app, as well as recent deployments. It also provides a useful UI to configure the project environment variables so as not to expose them on the client side of the app.

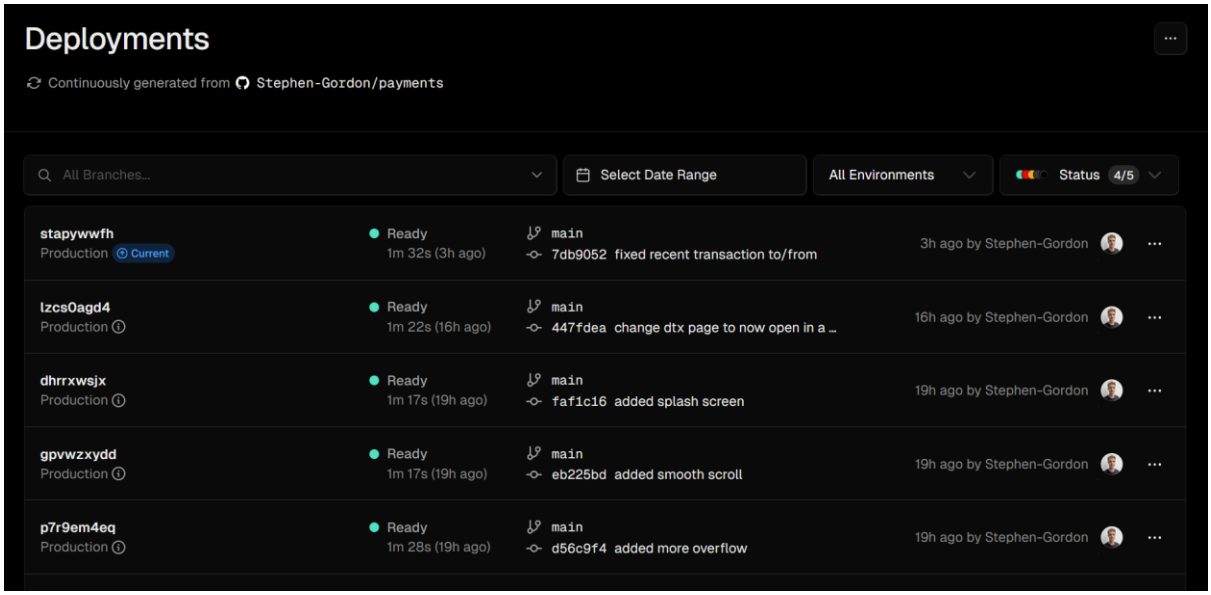


Figure 104 Vercel UI showing the app's recent deployments

Privy dashboard

Although it was not useful in development, the Privy dashboard (Figure 105 Privy dashboard) for this project could be useful going forward. It records useful data such as the number of users your app has and the login methods they used. Other login methods can be added to the application through this dashboard.

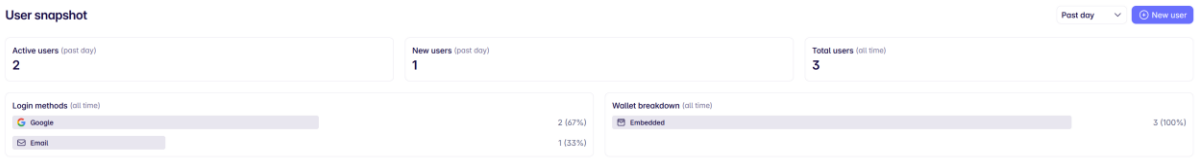


Figure 105 Privy dashboard

GitHub Copilot

GitHub co-pilot was useful when generating boilerplate code, such as basic Next JS pages. As copilot was not trained on the newest data and had limited knowledge about the packages I was using in my project such as ZeroDev Account Abstraction, Viem, Privy, and the PWA functionality, it was not helpful and often generated the wrong code. However, it was useful in coding well-known libraries such as Redux, creating type interfaces for TypeScript, and using basic JavaScript methods such as map and filter. For some specific use cases, Copilot helped speed up the development process, allowing the developer to spend more time on more complex features.

Conclusion

The aim of this project was to design, develop, and record the steps to build a payment app using progressive web app technology. The purpose of the app was to allow users to send and receive payments of cryptocurrency stablecoins such as USDC, by using a blockchain.

Summary

During the project proposal stage, I outlined a plan to build a payments app built with blockchain technology and how I got the idea. I outlined several desired features that would be nice to see in the app. Along with several examples of similar applications and a brief description of the technologies that would be used to build the app. Several problems with blockchain UX were outlined, and how new technologies such as account abstraction could fix these problems.

The research chapter reviews blockchain technology, blockchain wallets, and account abstraction to understand better the blockchain technology the app is built with. The research explores how blockchains work, the different types of blockchain wallets and how they are secured. It also investigates some common usability problems with blockchain wallets.

The requirements gathering chapter outlines the steps taken to find the required functionality for the app. Several similar applications were examined, functional and non-functional requirements were outlined, and several user personas were created to examine how potential users might use the app.

Using the gathered functional and non-functional requirements, the design process aimed to use this information to build wireframes, a prototype, a design system, and an application architecture diagram. An application architecture diagram, a user flow diagram, and an ERD were designed to help with the development of the app. Several technologies and SDKs that would later be used in the implementation sprints were examined.

The implementation phase consisted of several sprints, each sprint having a goal of features to implement into the app. During each feature, the process of implementing the feature and the reasoning behind implementing that way is described. During the implementation phase, several features were reimplemented to increase UX, security and functionality. Trello was used during this phase to keep a log of features created, bugs to fix, and new features to implement.

Both unit and user testing were conducted. The developer conducted unit testing to ensure that each function, component, route, and API endpoint worked as intended. Results were recorded in tables to keep track of features that were working and not working as expected. User testing took place to test the usability of the app. It was important to try the app with users to receive feedback on features not working that were not picked up by the developer. It was also useful to receive suggestions on how to improve the app's UX.

Business Opportunities

There are two possible business opportunities from this project.

The payment app could be further developed, and several business models could be added, such as transfer fees and an investing section where fees could also be implemented based

on the investment. A trading cryptocurrencies feature could be added, which could take a percentage fee for every trade. Similar revenue streams are seen in Revoult and Uniswap Wallet.

The second business opportunity is related to progressive web apps. Currently, there is a lack of consolidated information on PWAs; it is spread across YouTube videos, blog posts, SDK documentation, and StackOverFlow posts. A business could be setup that develops tooling and a workflow for PWAs. Two similar businesses are Vercel, the company that develops Next JS, which provides a framework for React to streamline the development process, and Expo, a company that creates tooling and packages and helps streamline app development for native apps. I think a similar company could also be created that would help with implementing PWAs on multiple devices, creating icons and splash screens, creating extensive documentation for implementation, and consolidating packages for accessing the native APIs such as the camera and notifications. This business could setup a similar business model and revenue stream to Vercel and Expo.

Project achievements and learning outcomes

The app is now fully functional and works on Android and iOS. The project successfully implemented PWA features and Account Abstraction to create an app that lets users send and receive money on a blockchain. The app is complete with login, authentication, transfer, contacts, camera, and notification features.

Along the way, I learnt several new technologies, and advanced technologies I was already comfortable with. When starting the project, I was comfortable developing with React and creating components. However, I built on top of my React knowledge by learning how to create custom hooks and use state management tools like Redux and frameworks like Next JS, all of which are industry standards. I further developed my experience with building apps by learning how to build a PWA. I now know what packages to use to implement a PWA and what APIs and features are available on the device. I learnt new blockchain technologies, SDKs and tools such as account abstraction, ZeroDev SDK, Alchemy, Viem, Wagmi, and Etherscan, some of which are widely used or similar to widely used tools in the blockchain developer community.

Further work

From the start, I would have liked to have built the app with React Native rather than building a PWA. When I first started developing the app, I looked at using the Privy SDK to handle the login and account creation. However, it was still in private beta, and I could not get access to it. Having access to Privy from the beginning would also have saved a lot of time and allowed me to spend time adding other features. However, I did learn a lot by using the Web3auth SDK and migrating to Privy. Then, later, when Privy was available, I was able to implement their SDK in my app. Since the start of April, Privy has now released an SDK for React Native for public use. If this were available in January, I would have liked to use it. However, I am still not sure if the Privy React Native SDK is compatible with ZeroDev and Account Abstraction; further research would have to be conducted to check the compatibility.

Another Privy SDK that could be implemented is an 'onramp' SDK. This SDK aims to help users deposit money from their bank account to their blockchain wallet, via Visa debit card, MasterCard, GPay, or other common payment methods. This would improve UX greatly.

When studying similar applications, this was a feature both Uniswap Wallet and Revolut had, with Revolut having many onramp options.

I would have liked to have spent more time implementing blockchain features that only account abstraction can enable, such as session keys, account recovery, passkeys and more. However, these features are nearly projects in themselves and would take a lot more research to implement them.

I want to use some form of decentralised identity solution in the app. Several of these are ENS, Polygon ID, and Ethereum Attestation Service. These services could be used to allow a user to have account nicknames, profile pictures, and much more customisability of the user's profile in the app while allowing the user to have self-custody of their information using their wallet rather than storing it in a centralised database.

I am happy with how I implemented push notifications, as it was one of the most difficult parts. However, I would like to change the fact that on iPhones, a user has to enable experimental features for Safari to receive notifications. During the development of the app, I found iOS to be very unfriendly to developers. Often, features such as accessing the camera would not work at all, randomly stop working, or only work in a production environment. Their documentation for PWAs needs to be improved; a lot of the documentation needs to be updated, with Apple recommending to use meta tags that are not even supported anymore in Safari. In general, Apple as a company seems very against PWAs, even threatening to ban their functionality from all Apple devices (Techcrunch, 2024). When working with CSS in Safari it often felt like a battle. Certain styles didn't work as intended and required extra work around because Safari did not support certain standards and the way they bundle CSS. During development, I found this iOS PWA compatibility list very useful to find what features were available (Firtman, 2024).

If I had more time, I would also like to spend more time on the UI, adding more live feedback, such as 'react-hot-toast' (react-hot-toast, 2024), to alert users of when an event was triggered. Also, adding more error handling, a welcome screen, an app tutorial, and more contact features, such as removing and editing a contact, could greatly improve the UX. For the backend, it would be nice going forward to consolidate it to make the code more reliable. Also, I need to encrypt the user's data, storing their notification subscriptions; this could be done using 'bcrypt' (bcrypt, 2024) a JavaScript encryption library. To improve the client-side code, TypeScript needs to be improved upon for a more secure app, along with Next JS server components, to provide a faster UI.

Support for multiple cryptocurrencies, along with multiple blockchains going forward, would provide better UX and increase the functionality of the app. While USDC is considered a safe stablecoin, it is not decentralized, with more research I would like to find out if user’s would use a payment app built with a decentralized stablecoin such as DAI by Maker Dao (Dao, 2024) or LUSD by Liquity (Liquity, 2024).

References

4337Mafia. (2024, 09 04). *4337Mafia*. Retrieved from 4337Mafia: https://github.com/4337Mafia/awesome-account-abstraction

Adobe. (2024, 09 04). *React Aria*. Retrieved from React Aria: https://react-spectrum.adobe.com/react-aria/

Airbnb. (2024, 09 04). *Airbnb*. Retrieved from Airbnb: https://www.figma.com/community/file/1206705782258966386

bcrypt. (2024, 04 10). *GitHub*. Retrieved from GitHub: https://github.com/dcodeIO/bcrypt.js

Biernacki, K., & Plechawska-Wójcik, M. (2021). A comparative analysis of cryptocurrency wallet management tools. *Journal of Computer Sciences Institute*.

Buterin, V. (2014, December 22). *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Retrieved from Ethereum: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf

Buterin, V. (2022). *Proof of Stake: The Making of Ethereum and the Philosophy of Blockchains*. Seven Stories Press.

Buterin, V. (2024, January 05). *Vitalik*. Retrieved from Vitalik: https://vitalik.eth.limo/general/2021/01/11/recovery.html

Chainlink. (2023, December 22). *Chainlink Smart Contracts*. Retrieved from Chainlink: https://chain.link/education/smart-contracts

Chohan, U. W. (2017). The Double Spending Problem and Cryptocurrencies. *Social Science Research Network*.

Coinmarketcap. (2024, January 05). *Coinmarketcap*. Retrieved from Coinmarketcap: https://coinmarketcap.com/

Dao, M. (2024, 04 13). *Maker Dao*. Retrieved from Maker Dao: https://makerdao.com/en/

Diagrams. (2024, 09 04). *Diagrams*. Retrieved from Diagrams: https://app.diagrams.net/

ENS. (2024, 04 09). *ENS*. Retrieved from ENS: https://ens.domains/

erc4337. (2024, 09 04). *erc4337*. Retrieved from erc4337: https://www.erc4337.io/

Ethereum.org. (2023, August 27). *Account abstraction*. Retrieved from Ethereum: https://ethereum.org/en/roadmap/account-abstraction/

Firtman, M. (2024, 04 10). *Notes*. Retrieved from Notes: https://firt.dev/notes/pwa-ios/

HeadlessUI. (2024, 09 04). *HeadlessUI*. Retrieved from HeadlessUI: https://headlessui.com/

Horne, L. (2024, 01 22). *stablecoins*. Retrieved from liamhorne: https://liamhorne.com/stablecoins

Houy, S., Schmid, P., & Bartel, A. (2023). Security Aspects of Cryptocurrency Wallets—A Systematic. *ACM*.

L2Beat. (2024, 04 09). *L2Beat*. Retrieved from L2Beat: https://l2beat.com/scaling/activity

Liquity. (2024, 04 13). *Liquity*. Retrieved from Liquity: https://www.liquity.org/

Masoud , M., Wiese, O., Beznosov, K., Roth, V., & Voskobojnikov, A. (2021). User, The U in Crypto Stands for Usable: An Empirical Study of. *Association for Computing Machinery*.

mebjas. (2024, 09 04). *GitHub*. Retrieved from GitHub: https://github.com/mebjas/html5-qrcode/issues/713

Mozilla. (2024, 09 04). *Dialog*. Retrieved from Dialog: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog

Nakamoto, S. (2008, December 12). *Bitcoin*. Retrieved from Bitcoin: https://bitcoin.org/bitcoin.pdf

Next. (2024, 04 09). *Next*. Retrieved from Next: https://nextjs.org/

Palatinus, M. (2013). *Bitcoin: Github*. Retrieved from Github: https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt

Privy. (2024, 04 09). *Privy*. Retrieved from Privy: https://github.com/privy-io/create-next-app

Privy. (2024, 04 09). *Privy*. Retrieved from Privy: https://github.com/privy-io/zerodev-example

Privy. (n.d.). *Privy*. Retrieved from Privy: https://github.com/privy-io/zerodev-example

React. (2024, 04 09). Retrieved from React: https://react.dev/

react-hot-toast. (2024, 04 10). *react-hot-toast*. Retrieved from react-hot-toast: https://react-hot-toast.com/

react-qrcode-logo. (2024, 09 04). *react-qrcode-logo*. Retrieved from react-qrcode-logo: https://github.com/gcoro/react-qrcode-logo

react-qr-scanner. (2024, 09 04). *react-qr-scanner*. Retrieved from react-qr-scanner: https://github.com/yudielcurbelo/react-qr-scanner

Redux. (2024, 04 09). *Redux*. Retrieved from Redux: https://redux.js.org/

Revolut. (2024, 04 09). *Revolut*. Retrieved from Revolut: https://www.revolut.com/

Safe. (2024, January 05). *Safe*. Retrieved from Safe: https://safe.global/

Sarmah, S. S. (2018). Understanding Blockchain Technology. *Scientific & Academic Publishing*.

Selikoff, S. (2024, 04 09). *CodeSandBox*. Retrieved from CodeSandBox: https://codesandbox.io/s/github/samselikoff/2022-07-01-ios-calculator-clone

Serwist. (2024, 09 04). *Serwist*. Retrieved from Serwist: https://serwist.pages.dev/docs/next

Shadowwalker. (2024, 09 04). *GitHub*. Retrieved from GitHub: https://github.com/shadowwalker/next-pwa

Swagger. (2024, 04 09). *Swagger*. Retrieved from Swagger: https://swagger.io/

Szabo, N. (2023, December 22). *uva.nl*. Retrieved from University of Amsterdam: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

Tailwind. (2024, 09 04). *Tailwind*. Retrieved from Tailwind: https://tailwindcss.com/

Techcrunch. (2024, 04 10). *Techcrunch*. Retrieved from Techcrunch: https://techcrunch.com/2024/03/01/apple-reverses-decision-about-blocking-web-apps-on-iphones-in-the-eu/?guccounter=1

Temzasse. (2024, 09 04). *Github*. Retrieved from Github: https://temzasse.github.io/react-modal-sheet/

theodorusclarence. (2024, 09 04). *theodorusclarence*. Retrieved from theodorusclarence: https://github.com/theodorusclarence/ts-nextjs-tailwind-starter

Uber. (2024, 09 04). *Uber*. Retrieved from Uber: https://www.figma.com/community/file/805195278314519508

Uniswap. (2024, 04 09). *Uniswap*. Retrieved from Uniswap: https://wallet.uniswap.org/

Wagmi. (2024, 04 09). *Wagmi*. Retrieved from Wagmi: https://wagmi.sh/

Wang, Q., & Chen, S. (2023). Account Abstraction, Analysed. *arXiv* .

ZeroDev. (2024, 04 09). *ZeroDev*. Retrieved from ZeroDev: https://zerodev.app/

Zimwara, T. (2020, September 19). *https://news.bitcoin.com/analyst-1500-bitcoins-lost-every-day-less-than-14-million-coins-will-ever-circulate/*. Retrieved from bitcoin.com: https://news.bitcoin.com/analyst-1500-bitcoins-lost-every-day-less-than-14-million-coins-will-ever-circulate/

Appendix

Appendix A – App code Repository

A GitHub repository containing the frontend of the application.

https://github.com/Stephen-Gordon/payments

Appendix B – Backend code Repository

A GitHub repository containing the backend of the application

https://github.com/Stephen-Gordon/payments-noti-backend

Appendix C – Trello Board

A Trello board used for project management.

https://trello.com/invite/b/hBEhof3F/ATTI9a01ff6d847d19850a45952d07df18a437EECEC2/major-project