University of Plymouth

School of Engineering, Computing, and Mathematics

COMP3000
Computing Project
2021/2022

CryptoShare

Khodadad (Adrian) Nouchin

10615420

BSc (Hons) Computing

Acknowledgments

There are a number of people I was fortunate enough to have around me that I would like to thank for their guidance and support throughout the duration of this project; firstly, my supervisor Shirley Atkinson for guiding the project and ensuring I was on the right path while focusing on the right aspects, and answering the countless questions I had regarding the minutiae of the project. I would also like to thank David Walker for his helpful ideas for the Natural Language Processing functionality of the chat bot, and Nathan Clarke for his help regarding the ability for users to change their passwords without having to re-encrypt all their data. Additionally, I would like to thank Leah, my girlfriend, for always being supportive, and my parents: Maryam and Simon.

Abstract

Research shows that investing in the stock and cryptocurrency market is becoming more commonplace; consequently, this trend has caused a significant demand for software that provides the ability to track investments and budget one's finances. In this report, the process of developing a web, desktop, and mobile application called "CryptoShare", that investors can use to manage their finances will be discussed and analyzed. The full software development life cycle of said undertaking will be outlined and explored, starting with an introduction to provide an overview of this document, and broadly explain what the project is. From there, the background of the project will be discussed, which will include statistics and relevant data from the literature surrounding the topic in order to justify the development of the application. Furthermore, market research will be carried out, and its result will be evaluated to ensure the software fills a realistic gap in the market, and provides value to its users. Once the background of the project has been covered in detail, the relevant legal, social, ethical, and professional aspects will be explored, with an emphasis on data security and user privacy due to the nature of the application. Subsequently, the user requirements of the project will be highlighted, which will be followed by the design process with the goal of producing wireframes and mockups of the user interface, along with UML diagrams to visualize how the user would interact with the system. By having a clear understanding of the requirements, and what the interface for each platform would look like, it then becomes possible to decide on a suitable tech stack to turn the concept into reality. Following the design process, the development approach and sprints will be broken down and discussed, along with the project management. The next two sections would involve the actual development of the software, and the quality assurance process, which ensures that the objectives and aims of the project have been reached. Finally, the report will conclude with an evaluation and summary of the work carried out, with discussions regarding its accomplishments, limitations, and the overall management of the project.

Table of Contents

Acknowledgments **1**

Abstract **1**

Table of Contents **2**

1. Introduction **5**
1.1. Aims and Objectives 6
1.2. Minimum Viable Product 6

2. Background **7**
2.1. Current Solutions 8
2.1.1. Functionality 8
2.1.2. Features 9
2.1.3. User Reviews 9
2.1.4. Existing Basis for the New Solution 10

3. Legal, Social, Ethical, and Professional Issues **10**
3.1. Legal 10
3.2. Social 12
3.3. Ethical and Professional 12

4. Design **13**
4.1. Requirements 13
4.1.1. User Persona 13
4.1.2. User Stories 13
4.1.3. Functional Requirements and Product Backlog 14
4.1.4. Non-Functional Requirements 15
4.2. Diagrams 15
4.2.1. Architecture Diagram 16
4.2.2. Activity Diagram 17
4.2.3. Sequence Diagrams 18
4.2.4. Entity Relationship Diagram 20
4.2.5. API Class Diagram 21
4.2.6. Mobile App Class Diagram 22
4.3. User Interface 23
4.3.1. Wireframes 23
4.3.2. Mockups 24

5. Development Approach **25**

5.1. Database 25
5.2. API 25
5.2.1. Node.js 25
5.2.2. TypeScript 25
5.2.3. GraphQL 26
5.2.4. Socket.IO and Natural Language Processing 26
5.3. Client-Side 26
5.3.1. Web App 26
5.3.2. Desktop App 27
5.3.3. Mobile App 27
5.4. Cryptography 28
5.4.1. Hashing with “bcrypt” 28
5.4.2. RSA 28
5.4.3. AES 28
5.5. Continuous Integration and Testing 28
5.6. Tools 29
5.7. Version Control System 29

6. Project Management 29
6.1. Risk Assessment 30
6.2. Gantt Chart 32
6.3. Kanban Board 32
6.4. GitHub Issues 33

7. Development 34
7.1. Initial Challenges 34
7.2. Sprints 35

8. Testing and Quality Assurance 38
8.1. Unit and Integration Testing 38
8.2. Usability Testing 39
8.3. Production Build Testing 39

9. Project Closure 40
9.1. Aims and Objectives Review 40
9.2. Project Post-Mortem 41
9.2.1. Tech Stack Evaluation 41
9.2.2. Project Management Evaluation 41
9.2.3. Personal Reflection 42
9.2.4. Areas of Improvement and Additional Features 43

9.3. Conclusions 43

References **44**

Appendices **47**

Appendix 1: Kanban Boards 47

Appendix 2: Software Engineering 50

Stock Market API Limitations 50

Changing Passwords Without Re-Encrypting Data 53

Desktop App Data Not Persisting 55

Incompatibility Between Cryptocurrency and Stock Market Data 56

Incompatibility Between Encryption Algorithms 57

Calculating Holdings Based on Trading Activity 57

Natural Language Processing Functionality 57

Appendix 3: Automated Test Results 60

Appendix 4: Usability Test Results 63

Session 1 - End of Sprint 4 63

Session 2 - End of Sprint 6 63

Session 3 - End of Sprint 8 64

Appendix 5: Production Build Test Results 65

Appendix 6: API Documentation 69

Express Routes 69

Socket.IO Event 70

GraphQL Resolvers 70

Appendix 7: Biweekly Stand-Up Meetings 75

Appendix 8: User Guide 76

Setting Up the API 76

Web App 78

Mobile App 86

Word Count: 10,420

Source Code: https://github.com/Xtrendence/CryptoShare

Demo Video: https://www.youtube.com/watch?v=nyzBvKHnYas

1. Introduction

Within this report, the full software development life cycle of a finance management application called “CryptoShare” will be discussed. The software itself is a cross-platform and open-source application, which includes a web, desktop, and mobile app. Additionally, it features a self-hostable GraphQL API, with a focus on security, data protection, and user privacy. The aim of the project is to provide users with a unified interface through which they can manage every facet of their finances, from budgeting, to tracking their cryptocurrency and stock investments.

First, the report will provide a detailed background for the project, with in-depth analysis and market research to identify key gaps the software can fill in the market, and aspects it can improve upon relative to its competition. The background research would serve as a foundation upon which the aims and objectives of the project will be based.

Once the background of the project has been discussed, the legal, social, ethical, and professional elements will be explored and highlighted. Considering the application involves the management of finances, it is incredibly important to thoroughly research the implications of its nature.

Subsequently, the design process will follow, with the gathering of user requirements, and the production of wireframes and mockups to ensure the interface is consistent across different platforms, and accessible to users. HCI and usability will therefore be discussed, with an emphasis on developing an application that is suitable for as many people as possible. Diagrams would also be used to visualize how the user would interact with the interface.

Utilizing the gathered user requirements, it would then be possible to decide on a suitable tech stack, as well as an adequate development approach with the appropriate tools and technologies.

Having decided on a tech stack and approach, it would then be possible to better estimate the schedule of the project, and plan it. The project management will therefore be discussed, along with the sprints as part of the agile methodology.

The development, testing, and quality assurance of the software will then be analyzed and discussed, with the use of diagrams to show the architecture of the software.

Finally, a summative and conclusive overview along with an evaluation will be provided.

1.1. Aims and Objectives

The aims and objectives of CryptoShare were primarily identified through the market research that is outlined in section 2. The salient aim of the application is to provide its users with a free, self-hostable, open-source, and cross-platform software, with a responsive web app, desktop app, and mobile app that allow individuals to budget their income, track their investments (whether stock or cryptocurrency assets), record their trading activity and everyday transactions, and keep up-to-date on the market as a whole. This would all be available to the user in a unified and accessible application with user experience as its top priority, with an emphasis on HCI and user customizability.

The main objectives of this project are to…

- Research the market and feedback obtained from CryptoShare's predecessor to help identify user requirements.
- Research the legal, social, ethical and professional requirements.
- Develop a self-hosted and performant API that can be easily deployed on any system without much set up necessary for server-side functionality.
- Develop a production-ready, free, and open-source unified web, desktop, and mobile application for client-side functionality.
- Produce a report that thoroughly examines the software development life cycle of the project, the research carried out before and while developing it, and the challenges faced throughout its duration along with lessons learned from creating a substantial and complex piece of software alone.

1.2. Minimum Viable Product

Normally, an MVP is released to a number of initial adopters that essentially test the system while it only has some basic features. For this project, a slightly altered process will be followed. Despite usability tests being done throughout the development of the project, a longer and more detailed one will be carried out once the main features of the application have been developed. In order to qualify as an MVP, the following features were deemed to be required at minimum:

- Check cryptocurrency and stock market data (prices, market cap, volume etc.)
- Add crypto or stock assets as holdings.
- View portfolio performance (profit, loss, and performance chart).
- Record monetary transactions, whether it is income, stocks, mortgage payments, cryptos etc.

2. Background

It is estimated that, in the UK, 33% of the population owns shares, with over ⅔ of the population stating that they plan on buying shares in the future. In 2018, this figure was 22%, which means a 50% increase in individuals who invest (Stelzer, 2020). The cryptocurrency market has had an even greater increase in popularity and growth, with its overall market capitalization increasing from ~$200B at the start of 2020, to almost ~$800B at the start of 2021. Currently, it sits at above $2T, a tenfold increase since 2020 (CoinMarketCap, 2022). Comparatively, looking at data regarding the S&P 500 index fund, it is evident that it has almost doubled in value. In other words, the top 500 companies in the U.S. are worth double (in market capitalization) what they were 5 years ago (Yahoo Finance, 2022). Analyzing Google Trends data over the past 5 years, it is also possible to see that interest in cryptocurrency has increased by a factor of 13, with stocks also increasing by a factor of 2 (Google Trends, 2022).

Considering the aforementioned figures, one can identify a clear and undeniable uptrend in the popularity of both stock and cryptocurrency investments. Furthermore, until recently, cryptocurrencies were considered a fad, with the most popular one, Bitcoin, being deemed “dead” countless times over the years (“Bitcoin Is Dead”, 2022). In contrast, nowadays, cryptocurrencies are being adopted and promoted by financial firms and institutions that are decades old, including Barclays and Goldman Sachs, with the latter having “digitalization” and the quote “from cryptocurrencies to the metaverse, explore the megatrends that are reshaping economies” on its landing page (Goldman Sachs, 2022). Aside from corporations, even entire governments are investing and utilizing cryptocurrencies, with El Salvador recognizing Bitcoin as legal tender, and holding large amounts of it in their treasury, which has actually increased tourism in the country by 30% since it started legally recognizing Bitcoin (Quiroz-Gutierrez, 2022). Governments such as Africa have other plans involving cryptocurrencies, such as using its immutable nature to digitally store and verify identities and ownership (e.g. an individual proving they own their house through a signed transaction on the blockchain). Developing countries especially benefit from this as corruption and lack of documentation and proof of ownership are significant issues (Cardano, 2022).

Combining the rising interest in investing with the adoption and credibility that cryptocurrencies are achieving, it is inevitable that software and tools that facilitate the tracking and inclusion of such assets in one’s budget become commonplace, which increases the demand for them. As such, CryptoShare aims to fill gaps in the market that other apps do not, and to provide its users with features and functionality that its competitors lack.

2.1. Current Solutions

When developing a piece of software, it is imperative to research the current market, and analyze what current solutions offer, as well as what they lack. In doing so, it is possible to envision a solution that tackles the shortcomings of other ones, and offers a realistic alternative while filling a gap in the market that others have failed to fill. Below, six different solutions will be analyzed in order to determine what functionalities they provide. The applications below were chosen to be analyzed due to their popularity and due to the fact that they are commonly used to track investments and keep up-to-date on the market, thereby having a similar goal to CryptoShare.

2.1.1. Functionality

First and foremost, users care about what they can do with an application. As such, the first step is to see what each app provides in terms of functionality.

	Application					
	Delta	Hodler	CoinGecko	Blockfolio	CoinTracker	CoinMarketCap
Functionality						
Crypto Market Data	✓	✓	✓	✓	✓	✓
Stock Market Data	✓	✕	✕	✓	✕	✕
Recording Trading Activity	✓	✕	✓	✓	✓	✓
Searching Trading Activity	✓	✕	✕	✕	✕	✕
Budgeting System	✕	✕	✕	✕	✕	✕
Manual Holding Editing*	✕	✓	✕	✕	✕	✕
Holdings Chart	✓	✓	✕	✓	✓	✓
Market Chart	✓	✓	✓	✓	✓	✓
Data Import/Export**	✕	✕	✕	✕	✓	✕

* Manual Holding Editing refers to the ability to set your asset holdings without having to create a transaction for the app to base your holdings off of.
** Data Import/Export entails providing a way for users to export all their data into, for example, a CSV file, which they can later import back into the application. This allows them to back up their data easily, and also access it offline and in any way they wish. This does not refer to cloud sync functionality.

2.1.2. Features

Having analyzed the functionalities provided by each app, it is relevant to consider the availability and accessibility of the apps themselves.

	Application					
	Delta	Hodler	CoinGecko	Blockfolio	CoinTracker	CoinMarketCap
Feature						
Self-Hosted*	✕	✕	✕	✕	✕	✕
Cross-Platform**	✕	✕	✓	✕	✓	✓
Open-Source	✕	✕	✕	✕	✕	✕

* Self-Hosted apps would allow users to host their own instance of the app instead of relying on third-party servers. For long-term applications that could potentially be storing years of transaction data for users, it is important to provide the user with the ability to keep using the service in the event the service provider no longer exists. When self-hosting, users can also rest assured that their private financial data is not being sent to third-party and unknown servers.
** Cross-Platform apps allow the user to access their financial data on any platform, whether it is on the web, their desktop or their mobile phone. Throughout the report, cross-platform will refer to applications that are available on the aforementioned three platforms, not only the different operating systems for one platform (i.e. Android and iOS for mobile, or Windows, macOS and Linux for desktop).

2.1.3. User Reviews

Despite having listed the functionality and features that current solutions provide, as a developer, it is oftentimes difficult to objectively judge and determine what the user wants. Consequently, an incredibly useful way to obtain valuable market research data is through reading the reviews that users have left on existing applications. By doing so, one can get a better understanding of what users looking for similar apps value. In order to compete with existing solutions, the new solution must offer an incentive for users to switch over. User reviews are perfect for identifying areas to improve upon. After analyzing reviews on all six applications, it was found that the most common complaints involved the apps forcing users to create transactions in order to set their holdings. Other reviews focused on issues regarding the UI of the apps, with some mentioning the lack of any backup functionality, and the unavailability of servers.

2.1.4. Existing Basis for the New Solution

While studying the competition is invaluable, it is noteworthy to consider that CryptoShare is based on an existing application that the developer aims to rebuild from scratch while adding a host of new features and requested functionality. The existing software is called Cryptofolio, and has close to 200,000 downloads (based on GitHub Release downloads and Docker Hub pulls), with hundreds of requests and suggestions from its users. By combining the feedback from Cryptofolio, with the user reviews of existing solutions, while also developing additional features with an improved and more performant tech stack, CryptoShare can provide excellent value to its users.

3. Legal, Social, Ethical, and Professional Issues

When developing software, it is imperative to consider the legal, social, ethical, and professional issues that it entails. Considering this project involves financial data, it is especially important to research the relevant laws and ensure that, as a “FinTech” (Financial Technology) app, it complies with all rules and regulations.

3.1. Legal

Since the application is being developed and is intended primarily for UK citizens, there are a number of British laws it needs to follow. First of all, it is important that the software is accessible to all people, regardless of their disabilities (Disability Discrimination Act, 1995). As such, the application needs to be tested to ensure it adheres with the Web Accessibility Initiative standards, and that the user interface is accessible to users who may be colorblind, have motor disabilities etc.

Taking into account the sensitive nature of financial data, it is of paramount importance to not only protect user data once stored (Data Protection Act, 2018), but also during transit. The relevant requirements of the Data Protection Act, in the case of CryptoShare, include protecting user data against unauthorized access. Furthermore, the regulation requires that any data collected be absolutely necessary, and that such collection be transparent to the user. Furthermore, users must have the ability to delete their data in accordance with the “right to be forgotten”, as well as request it or download it as part of the “right to data portability” (GDPR, 2016).

Normally, applications providing financial services are required to carry out certain procedures, such as KYC (Know Your Customer), which entails requesting and storing user identity documents such as passports and a recent bank statement (Baus, 2022).

Since CryptoShare does not actually provide any financial services, and does not interact with or handle users' money, financial laws do not apply.

In order to meet the aforementioned data protection requirements, the application does the following:

- Allow the open-source API to be self-hosted, allowing for full transparency and control over one's data.
- Collect the minimum amount of information from the user (usernames and passwords only).
- When registering, users are shown a message written in simple and plain English that explains exactly where their data is stored and how market data is retrieved.
- Data stored in the SQLite database is encrypted with AES-256 on the client-side before being sent to the server (so the server does not have access to the underlying data at any point).
- User data is encrypted using a key generated on the client-side, which is also encrypted using the user's password. Even if the database's content were stolen, all user data would be safe.
- Login credentials are encrypted with RSA on the client-side before being sent to the server, so even without SSL/HTTPS, usernames and passwords are always secure during transit.
- Passwords are hashed using "bcrypt".
- Tokens are used when making requests to the API (after logging in), so user credentials are not constantly being sent back and forth.
- Users can export their data as JSON and CSV files, making it easy to move around. The data can then be imported again should the user wish to do so.
- Users can easily delete all their data, or their entire account.

It is worth noting that since CryptoShare is intended to be hosted by the user of the application, some of the laws concerned with the storage of data do not strictly apply, as the user is not "providing" their data to any other person(s) or company. At all times, they are in full possession of their data, and none of it is ever transmitted to any other server or third-party. However, since the application has a registration feature, it is appropriate and ethical to provide functionalities such as the ones listed above, due to the fact that users may decide to host a public instance, and allow others to sign up; though in that case, the legal responsibility would fall on the host rather than the original developer.

3.2. Social

There are a number of social issues that arise from the creation and usage of a financial app such as CryptoShare. When looking at the social impact of investing, research shows that it can induce stress in people, and that the recommended way to handle the stress is to stop looking at the charts (When the Stock Market Gets You Down, 2022). It is reasonable then to be wary of the impact the software could have on people's mental health. Fortunately, there are steps one can take to both reduce the amount of stress investing can cause, and ensure that they are protecting their assets. The most important rule is to only invest what one can afford to lose (Kennon, 2022). CryptoShare aims to offset the negative social ramifications of investing by providing the means to invest responsibly; it does this by offering budgeting tools the user can utilize to ensure they are diligent with their finances, and aware of how much they are spending and earning at all times. Furthermore, one of the key ideas that CryptoShare (and other applications like it) promotes is the act of retaining one's trading history, and always keeping records on their transactions and activities. Doing so encourages responsible financial management, and ensures individuals can safely and accurately calculate their Capital Gains Tax, and avoid complications in the future.

3.3. Ethical and Professional

In terms of the ethical and professional issues that concern this project, the majority of them relate to the usability tests that will be carried out to prevent accessibility and usability issues. In order to remain ethical and professional, a great deal of focus will be given towards maintaining the anonymity and confidentiality of participants, while respecting their rights by thoroughly explaining each session's activities, and making sure they fully consent to participating.

Furthermore, as far as anonymity and confidentiality are concerned, the same approach will be taken when dealing with potential suggestions, questions, and bug reports from users, where they will be kept anonymous.

Additionally, another factor to consider would be the usage of copyrighted work during development, and the proper attribution of third-party services and/or libraries. To address this, all the libraries and external resources used throughout the software or otherwise will be credited in the "README" file of the version control repository.

4. Design

One of the key phases of the software development life cycle is the “design” phase, as it begins the process of turning a concept or idea into a tangible product. The process starts with the identification of user requirements. Properly understanding user requirements is an absolute necessity, and is critical to the success of any piece of software (Maguire and Bevan, 2002). User personas are also created, which are then used to derive the user stories. Once the functional and non-functional requirements have been identified, diagrams are used to visualize the components of the software and how they interact with one another. Finally, wireframes and mockups are created based on the requirements.

4.1. Requirements

The user requirements of the product are based on three foundations:

- The sizable amount of feedback for CryptoShare’s predecessor (Cryptofolio).
- The common shortcomings and complaints regarding similar applications.
- A user persona that shows the needs of most cryptocurrency investors (who are often very conscious about their privacy, and are highly protective of their data).

4.1.1. User Persona

Maria is a financially responsible retail investor that trades cryptocurrencies and stocks. As a fairly tech-savvy but also privacy-conscious person, she is aware of how her privacy can be invaded by using proprietary applications that store her data on their servers. She requires a solution that lets her have control of her data to know it is protected, but also lets her manage her income and finances.

4.1.2. User Stories

As a privacy-conscious individual, Maria wants a solution that…

- Keeps her data private and away from the cloud/unknown servers.
- Encrypts and protects her financial data.
- Requires a secure login system.
- Is open-source and self-hosted.
- She can use and access on any platform.

As a cryptocurrency investor, she wants to be able to…

- Check cryptocurrency market data.
- Manage her cryptocurrency holdings/portfolio.
- Manage her cryptocurrency transactions.
- Calculate her staking and/or mining rewards.

As a stock investor, she wants to be able to…

- Check stock market data.
- Manage her stock holdings/portfolio.
- Manage her stock transactions.
- Calculate her dividend earnings.

As a financially responsible individual, she wants to be able to…

- Track her income.
- Budget her income.
- Calculate her taxes.
- Calculate her insurance.
- Calculate her mortgage.

4.1.3. Functional Requirements and Product Backlog

Having identified the user requirements. It is then possible to create a product backlog, which lists the functional requirements (derived from the user stories) broken up into several sprints. Each sprint has a start date, and an end date, with a duration of 2 weeks. However, the dates are adjusted to consider 8 hour work days, 40 hour weeks, and 20 day months.

Sprint	Start	End	Tasks
Sprint 0	01/10/2021	20/10/2021	• UI Design Concept Inspiration • High-Level Planning • User Stories & Requirements • Setting Up Environment
Sprint 1	21/10/2021	09/11/2021	• Entity Relationship Diagrams • Tables • Views
Sprint 2	10/11/2021	29/11/2021	• API UML Class Diagram • API Helper Functions / Classes • API Endpoints

Sprint 3	30/11/2021	17/12/2021	● Bot Setup ● Bot WebSocket ● Bot Responses ● Bot Functions ● Bot NLP Functionality
Sprint 4	18/12/2021	05/01/2022	● App UML Class Diagram ● Navigation & Common UI Elements ● Login ● Encryption ● Settings
Sprint 5	06/01/2022	25/01/2022	● Cryptocurrency Market Data ● Cryptocurrency Holdings ● Cryptocurrency Transactions ● Staking Calculator ● Mining Calculator
Sprint 6	26/01/2022	14/02/2022	● Stock Market Data ● Stock Holdings ● Stock Transactions ● Dividend Calculator
Sprint 7	15/02/2022	04/03/2022	● Chat Bot ● Income Tracker ● Budgeting System ● Tax Calculator ● Insurance Calculator ● Mortgage Calculator
Sprint 8	05/03/2022	23/03/2022	● Testing In Production

4.1.4. Non-Functional Requirements

Due to the fact that the software is self-hosted, most of the non-functional requirements of the API, such as speed, availability, capacity etc. are entirely dependent on the device hosting it. On the client-side, however, speed and usability are a top priority.

4.2. Diagrams

Following the identification of the user requirements, the solution needs to be planned in detail. Using diagrams, it is possible to visualize exactly how the different parts of the system interact with each other, and what the structure of the codebase looks like.

4.2.1. Architecture Diagram

CryptoShare would have a client-server architecture, as clients would not interact with one another, and would interact with a GraphQL RESTful API for CRUD functionality. Its architecture would be classed as a 3-Tier Architecture, with the UI and such being stored on the client's device, and the API and database being stored on the server. GraphQL would act as middleware between the client and server, handling clients' specific needs and controlling the data logic to help with performance and flexibility. It was decided that this would be the ideal architecture as the software aims to provide multi-user functionality, with users being able to register accounts to use the application. The host/admin, of course, would be able to disable this if they only want to use CryptoShare for themselves.

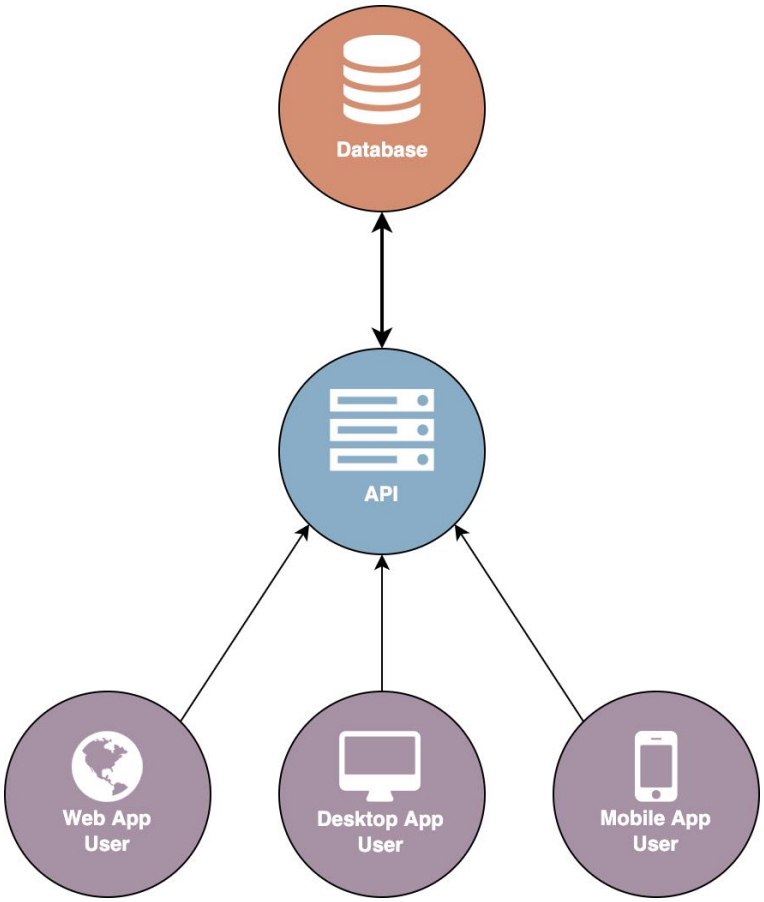


Figure 1 - Architecture Diagram

The image above shows the architecture of the application. Users would be able to use any of the 3 client-side platforms to interact with CryptoShare's API, which reads and writes from and to a database.

4.2.2. Activity Diagram

An activity diagram has been provided below to visualize how the user would interact with the chat bot.

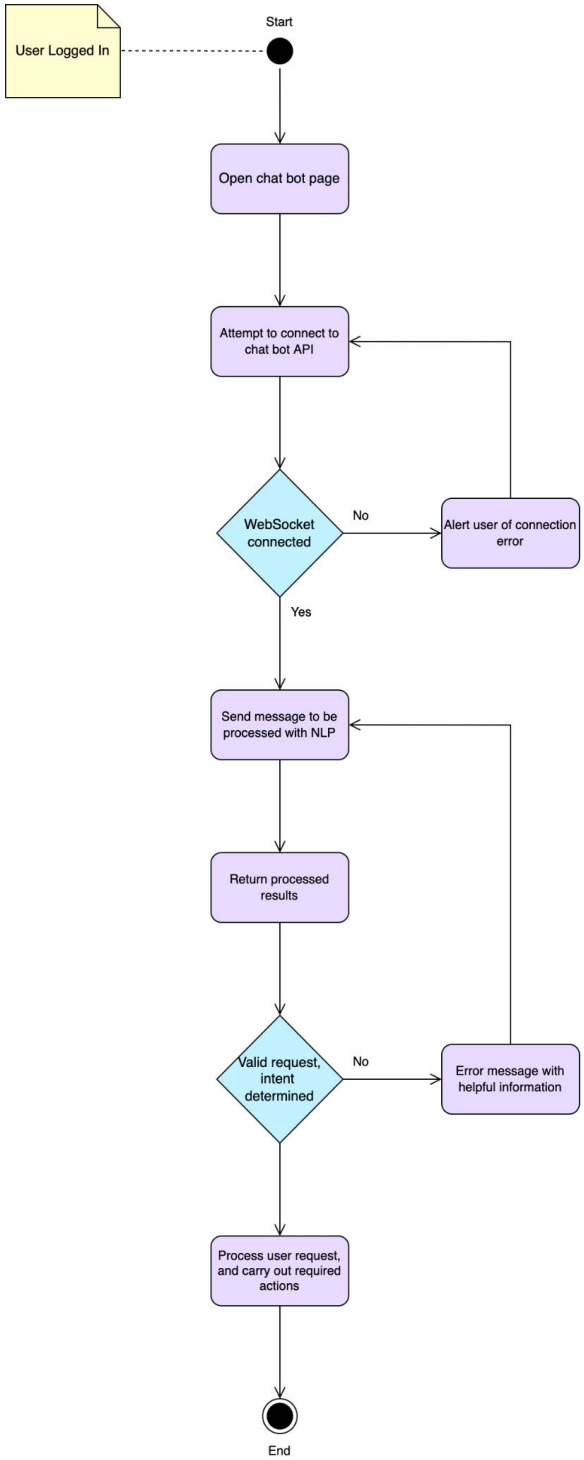


Figure 2 - Activity Diagram

4.2.3. Sequence Diagrams

The sequence diagram below showcases exactly how the client-side software interacts with the API during a standard login process. The API then interacts with the database to store and retrieve the relevant data. Please note that login details are encrypted on the client-side before being sent to the API.

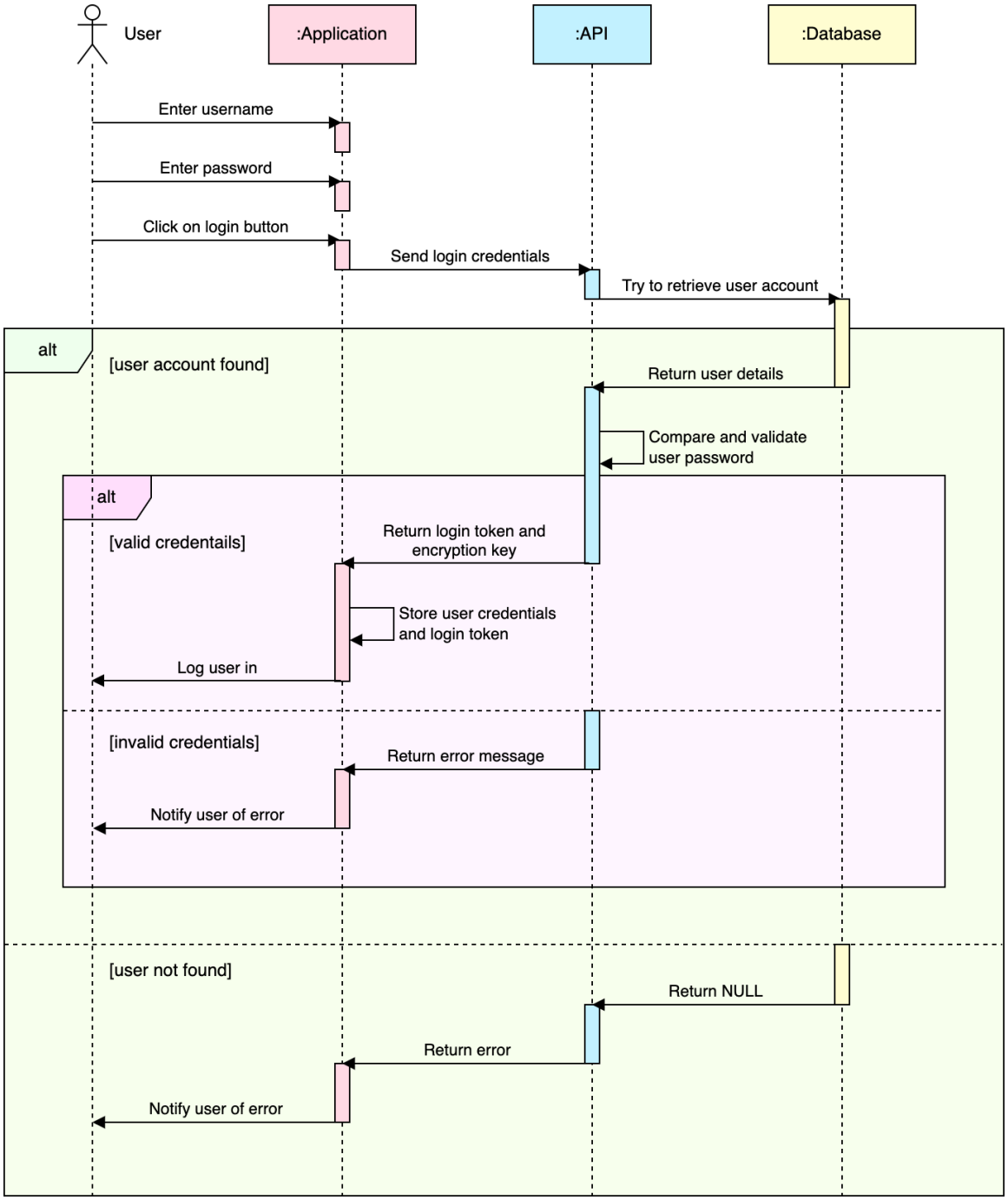


Figure 3 - Login Sequence Diagram

Once the user has logged in, all other requests require the user to have a valid login token. The process slightly differs from logging in, so a different sequence diagram has been designed to visualize how the user interacts with the system once logged in. In this diagram, the user is updating their holding data. Please note that the data is encrypted on the client-side before being sent to the API.

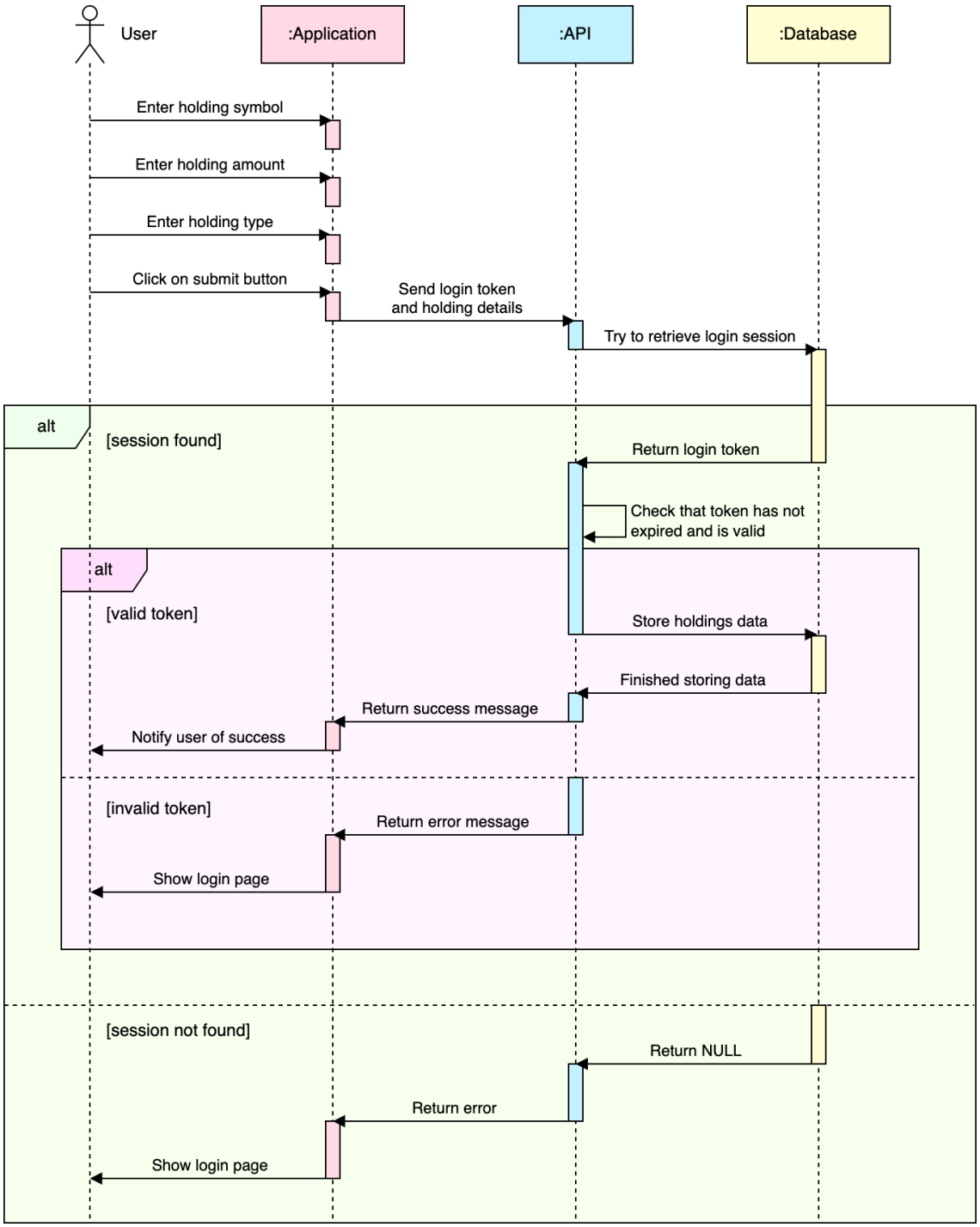


Figure 4 - Update Holding Data Sequence Diagram

4.2.4. Entity Relationship Diagram

In order to present the structure of the database, an ERD has been designed. Please note that the actual schema uses the “BLOB” data type for most columns, as all user data is encrypted and should not be stored under regular data types, such as “VARCHAR” or “INT”; therefore, the diagram below shows the underlying plaintext data:

Figure 5 - Entity Relationship Diagram

As seen above, some attributes such as the “password” and “key” in the “User” table do still need to have “BLOB” as their data type due to the fact that passwords are hashed, and encryption keys are themselves encrypted using the user’s password.

It is worth noting that the “Stock” and “Coin” tables do not have a relationship between any other tables, as they are used solely for data caching as part of a solution that will be expanded upon in the development section of this report.

4.2.5. API Class Diagram

Since TypeScript was used rather than, for example, C#, the class diagram had to be adapted to a more functional programming style. As per the ES6 documentation, modules are imported by other classes and files that use them (JavaScript modules, 2022). To make it easier to understand, files whose contents are simply read have an arrow label that says “uses” to imply the content is used rather than imported.

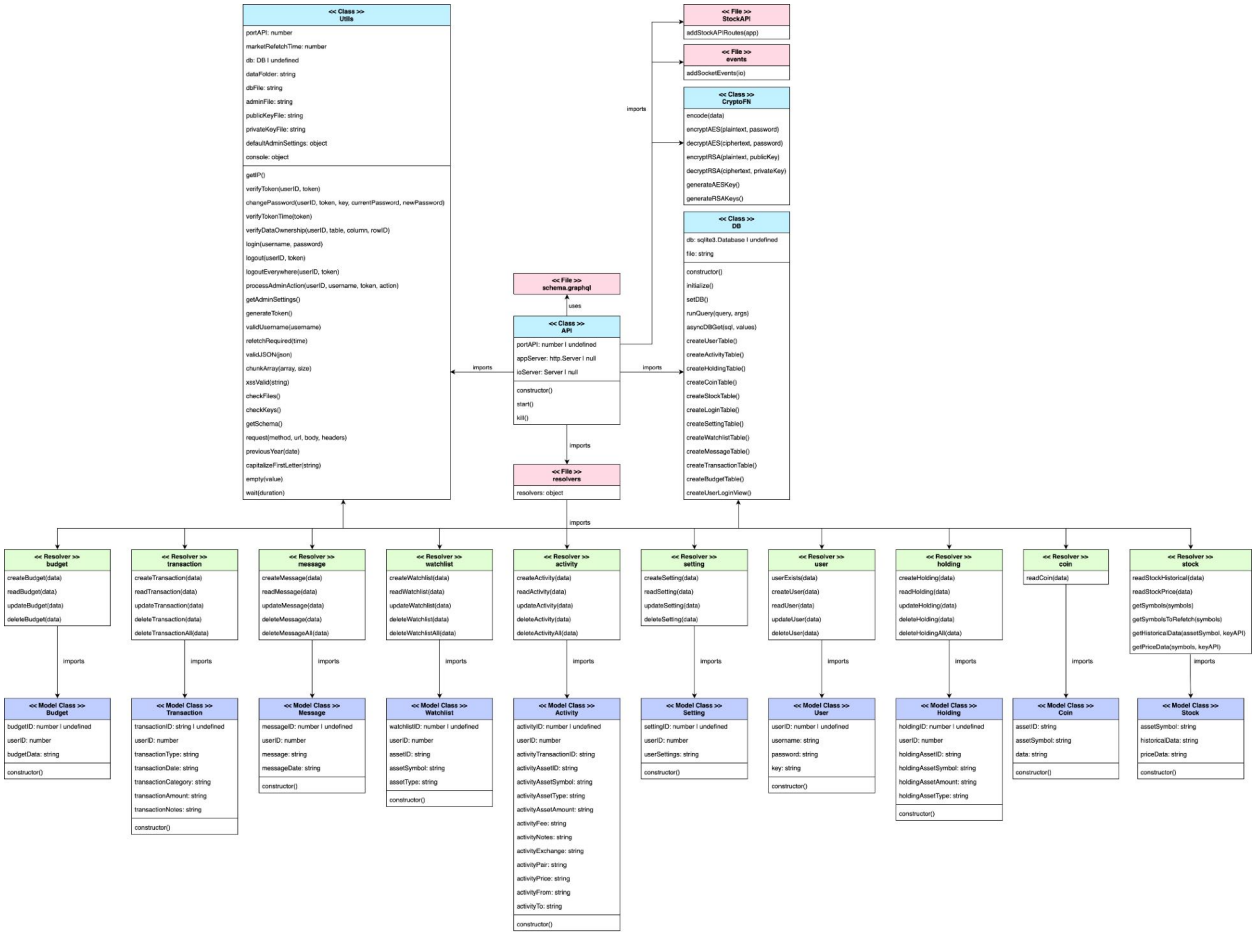


Figure 6 - API Class Diagram

High-Quality Image: https://i.imgur.com/zxVWcbW.png

4.2.6. Mobile App Class Diagram

Similarly to the API's UML class diagram, React Native also uses functions that are imported in the same way as other ES6 modules. In this case, functions that return a React component have the label "<< Component >>" to make the diagram easier to understand. Components that return a page that can be navigated to are labeled as "<< Screen >>". Some helper classes and functions have been omitted to save space.

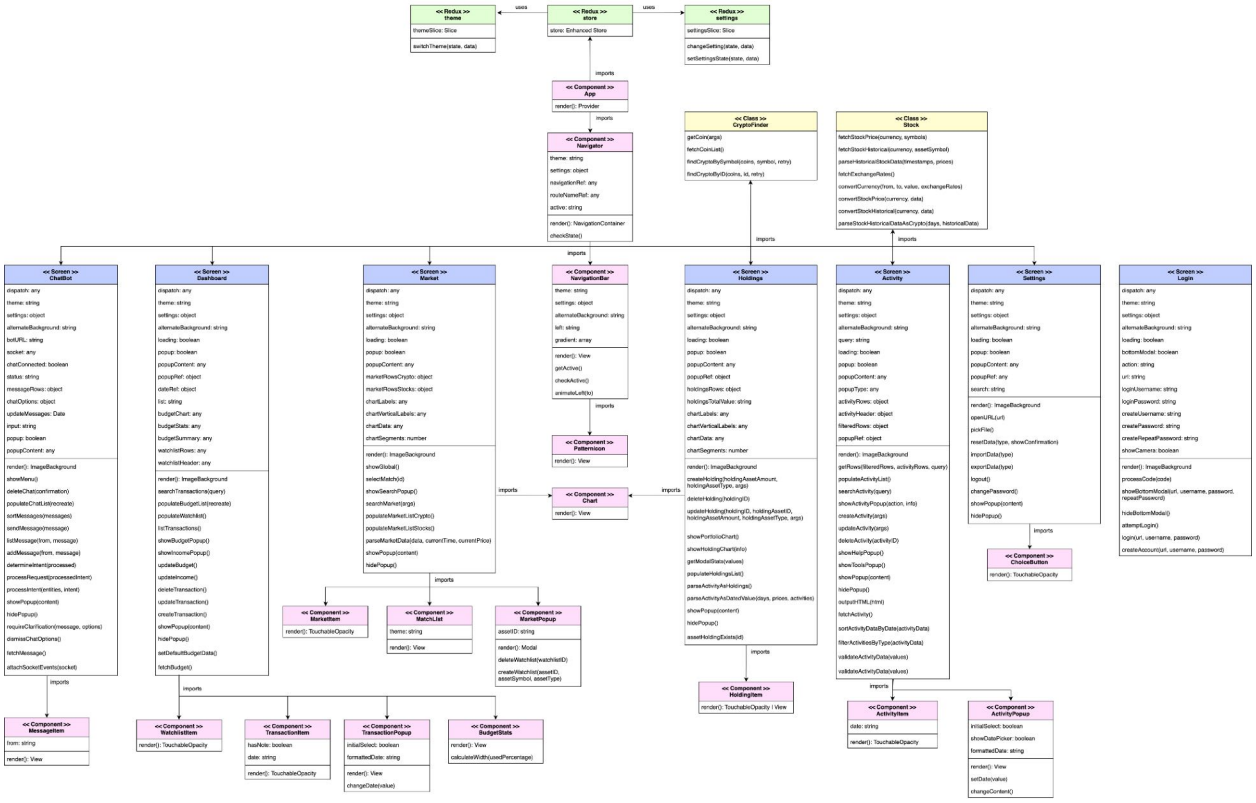


Figure 7 - Mobile App Class Diagram

High-Quality Image: https://i.imgur.com/KvKUeXz.png

4.3. User Interface

Having identified the functionalities that the software would provide, it becomes relevant to design a user interface that lets users access the features listed in section 4.1.3.

4.3.1. Wireframes

Before designing detailed mockups of the user interface, wireframes were created. The first one below is the “Market” page, and the second is the “Dashboard” page.

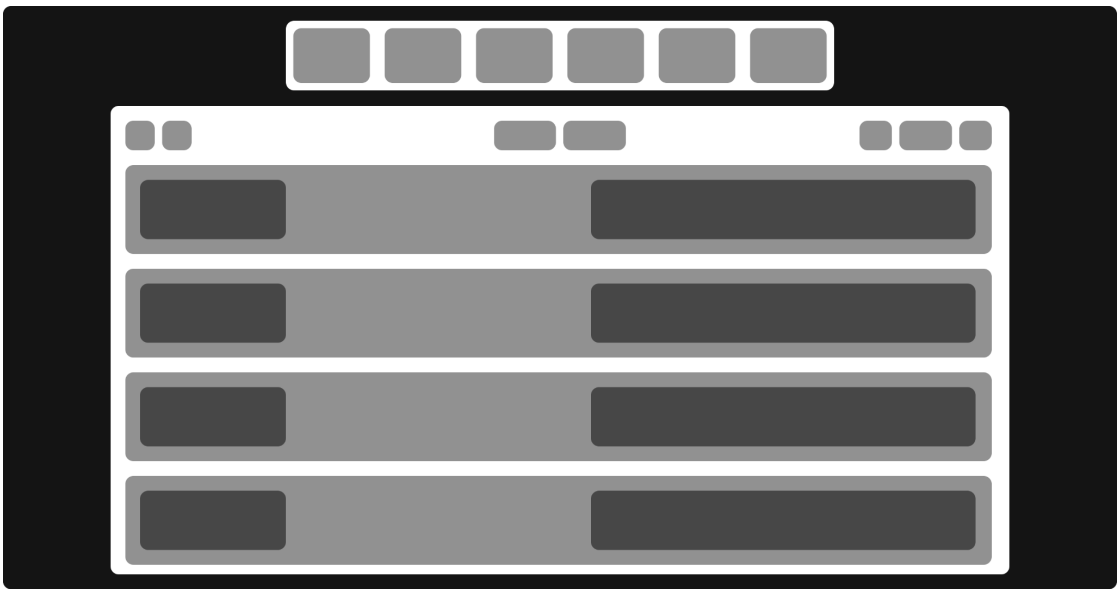


Figure 8 - Market Page Wireframe

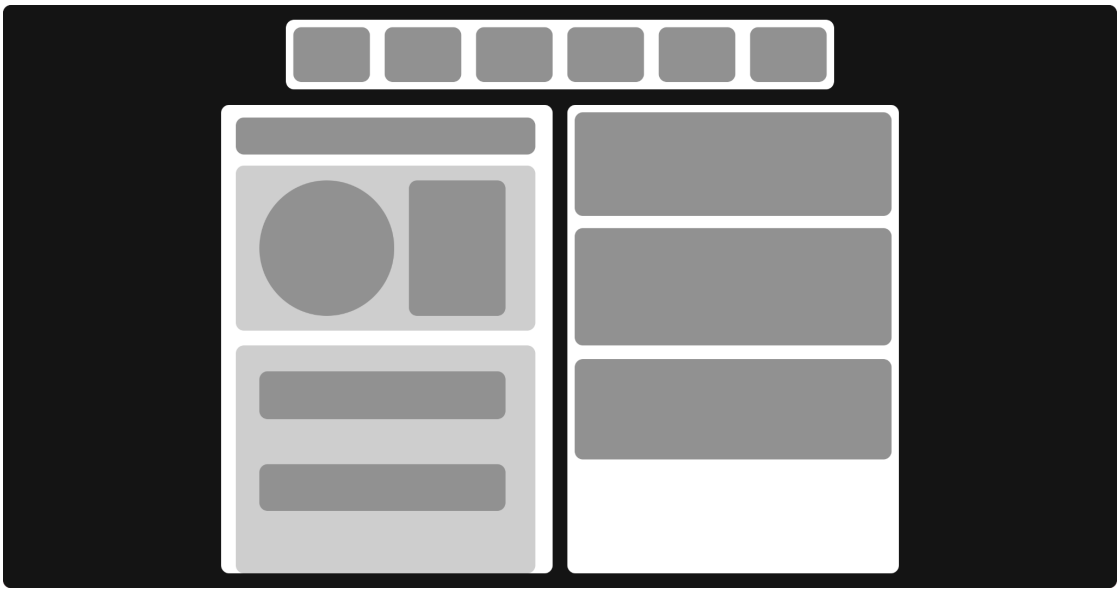


Figure 9 - Dashboard Page Wireframe

4.3.2. Mockups

Using the wireframes as a foundation, detailed mockups were designed using Photoshop. The eventual user interface of the application was based on a pixel-perfect recreation of the mockups. Below, the “Activity” page and “Holdings” page are shown.

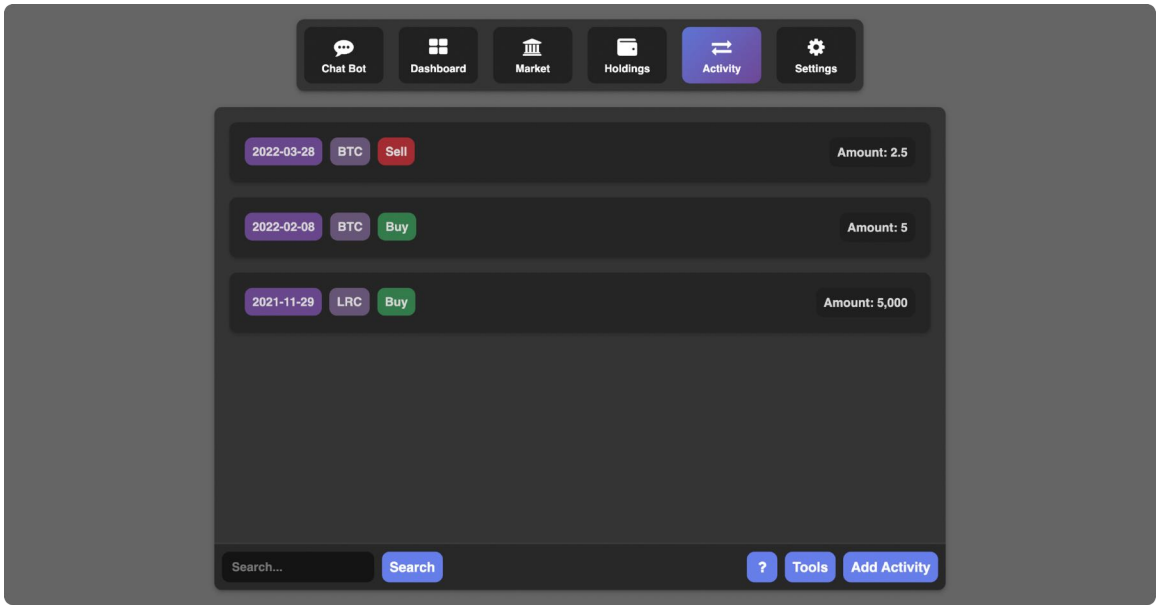


Figure 10 - Activity Page Mockup

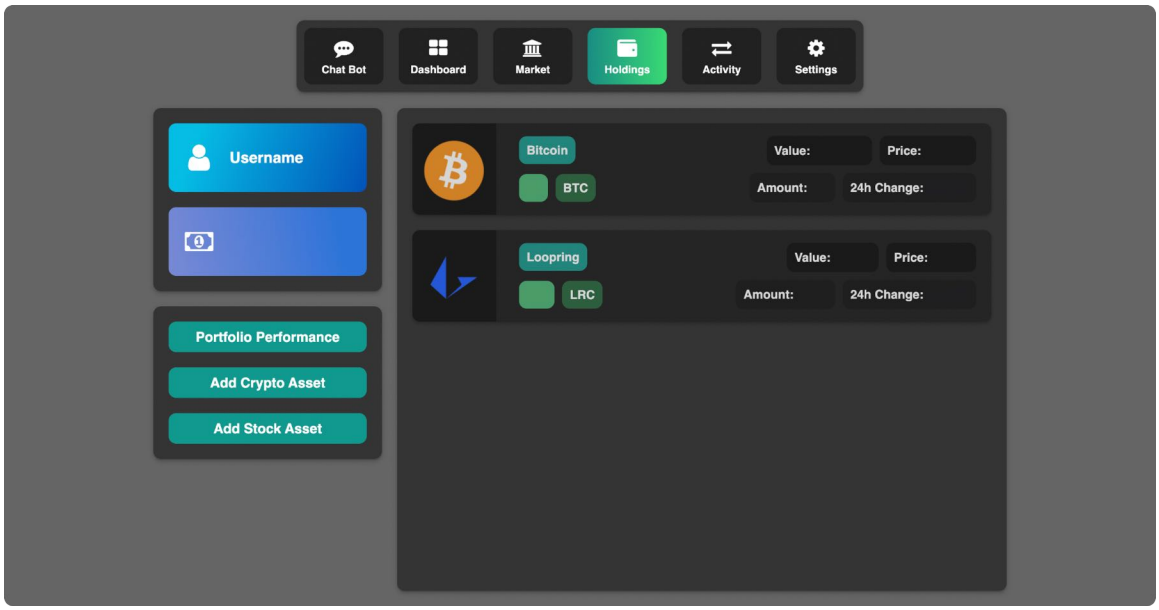


Figure 11 - Holdings Page Mockup

Please note that there is no background for the mockups as it was not clear at this stage which background would be used.

5. Development Approach

Once the functional and non-functional requirements were clear, and mockups of the user interface were designed, choosing a tech stack became a much simpler task. “When choosing a technology stack for app development, you first need to rely on the requirements of your project” (Puzhevich, 2020). After carefully considering the user requirements, the tech stack described below was chosen.

5.1. Database

One of the key concepts that CryptoShare aims to incorporate into its software is the idea that user data can be portable and fully under the user’s control. When paired with the goal of having the software be easily hostable, SQLite emerged as the ideal DBMS. While MySQL, SQL Server, and MongoDB require whoever is hosting the application to install and set up other software, SQLite allows for most of the benefits of MySQL with none of the hassle. On the mobile app, Redux would be used to store and manage the state of the application, such as the active theme and other user settings.

5.2. API

Choosing server-side technologies suitable to the requirements of the project was one of the more challenging aspects of the planning process. This is due to the fact that a Natural Language Processing (NLP) library will be needed for the chat bot feature of the application to function. Furthermore, the API will need to support web sockets in order for the user to be able to chat with the AI. The technologies that follow were chosen based on the aforementioned factors.

5.2.1. Node.js

Due to Node.js’ support for web sockets, and its wide array of libraries, it was decided that it would be a better choice compared to PHP and Python (which also had libraries for NLP, but were not as compatible with the chosen client-side technologies).

5.2.2. TypeScript

In order to build a more robust and less ambiguous API, TypeScript will be used instead of vanilla JavaScript. TypeScript is a strongly-typed version of JS, that reduces ambiguity in the code, and allows for much easier debugging and development. Considering the API will handle sensitive user data, reliability is a top priority.

5.2.3. GraphQL

GraphQL is an open-source data query and manipulation language for APIs. It acts as middleware between the client and the API, and allows clients to make custom requests where they can ask for exactly what they need and not receive any additional useless data. This speeds up query times and improves performance (Why adopt GraphQL?, 2022). CryptoShare aims to provide an open-source platform for finance management that anyone can contribute to; as such, its API needs to be well-documented, and easily understandable. GraphQL simplifies this objective, while also offering several advantages as mentioned.

5.2.4. Socket.IO and Natural Language Processing

Web sockets are ideal when it comes to situations where real-time communication is required, with fast reaction times and bi-directional data transmission (When to use a HTTP call instead of a WebSocket, 2016). While web sockets can be used without additional libraries, Socket.IO “provides additional guarantees like fallback to HTTP long-polling or automatic reconnection” (Socket.IO, 2022). Through its use, CryptoShare can provide real-time feedback to users to let them know whether they are connected to the bot, or if they are able to send it messages. Furthermore, the Natural Language Processing will be done server-side, and if the process takes too long, a normal HTTP request could time out, whereas a web socket would simply send a message to the user when it is ready. Further details on the NLP functionality can be found in Appendix 2.

5.3. Client-Side

Due to the uniform design of the software across the different platforms, one of the most important aspects when choosing the client-side technology is the ability to make the UI look the same irrespective of the platform. Other considerations include support for the libraries used for the cryptographic functions, as users would not be able to access and modify their data across different platforms otherwise.

5.3.1. Web App

The web app will utilize HTML, CSS, and vanilla JavaScript in order to provide users with a single-page web application. Through the use of web sockets and HTTP requests, the pages would be populated with content. No templates or frameworks will be used, as the user interface will need to be the same across the different platforms, and the templates might not be available on all of them; furthermore, CryptoShare aims to offer a unique and customizable interface that templates cannot provide.

5.3.2. Desktop App

Through the use of Electron, the desktop app can have the exact same codebase as the web application. This would simplify debugging and testing, while also providing users with software that feels native to the system. Additionally, it offers cross-platform support, automatic updates, and increased development speed and productivity. Its success has resulted in popular applications such as Discord, Slack, Atom, Trello, and others to use it (Dryka and Pluszczewska, 2020). In order to run the application, a Node.js server would be used along with Express, that would use Electron as a compatibility layer between the operating system and application. When building the application for production, Electron would allow for the Node.js server and its modules to be neatly compressed into an executable file that users can run in order to use CryptoShare without having to install any other software.

5.3.3. Mobile App

Choosing the technologies that will be used for the mobile app is yet another difficult decision due to the amount of options, and their individual advantages and disadvantages. After thoroughly researching the available choices, it became clear that there were three frameworks that matched the user requirements: Xamarin, Flutter, and React Native, as they are the most commonly used and supported ones. (Manchanda, 2021). While all of these frameworks provide the ability to develop a cross-platform mobile app, they each have their own advantages and disadvantages. Xamarin, compared to the other two frameworks, has an inferior user interface creation tool, and the framework itself has been decreasing in popularity and usage (Martin, 2021). Flutter and React Native are both frameworks that put good UI design and UX at the forefront of issues they aim to resolve, and both offer a stable platform, with a reliable hot-reload/fast-refresh functionality, allowing the developer to see changes they have made instantly without having to rebuild the application. All three frameworks have package managers with a wide array of third-party libraries that can be used as well.

Ultimately, React Native was the most sensible and popular option (Liu, 2021); not only does NPM have a much larger selection of libraries to use, but JavaScript being the framework's programming language means that many of the functions coded for the web application could be reused in the mobile app; this would make debugging and testing much easier. Additionally, the framework supports Redux, which is a library that makes state management reliable, simple, and predictable (Bugl, 2017).

TypeScript will also be used, as mobile applications are normally more “fragile” than web applications and can actually crash, requiring a more robust and reliable codebase.

5.4. Cryptography

CryptoShare will make use of hashing, as well as both symmetric and asymmetric encryption in order to protect user data.

5.4.1. Hashing with “bcrypt”

When choosing a hashing algorithm, it is of paramount importance to consider the rapid increase in computing power. By using “bcrypt”, it is possible to easily increase the number of iterations used when hashing, for example, passwords. Increasing this number makes it much more expensive to brute-force the hash in order to reverse it (Arias, 2021). One of the advantages that bcrypt provides is the fact that the salt is stored alongside the resulting hash, which makes storing passwords much easier. CryptoShare will use bcrypt to hash user passwords, which can later be compared to the hash of a provided password by using the same salt; this is used to log the user in.

5.4.2. RSA

RSA encryption will be used when transmitting login credentials to and from the API. This would ensure that even if the host does not have SSL/HTTPS set up, user details are always encrypted during transit. When started, the API would generate a public and private key pair. Its public key would be relayed to clients that connect to it, and the clients would then be able to encrypt data using the key. From there, the API could decrypt any received data using its private key.

5.4.3. AES

AES encryption will be used when making requests to the GraphQL API. In this case, data would be encrypted on the client-side using the user’s key, which is unknown to the server. As a result, users can send data to the API to be stored without ever revealing what the data actually is. This is ideal in terms of security, as neither a man-in-the-middle or compromised API server could steal user data.

5.5. Continuous Integration and Testing

GitHub Actions will be used as part of Continuous Integration. Unit and integration tests will be written using the Jest testing framework, and will be run every time a commit is pushed to the main branch. The “testing” section of this report will go into further detail about exactly what tests are run, and why.

5.6. Tools

A number of tools will be used throughout the development of the project.

- The code editor that will be used is Visual Studio Code, though various extensions will be used to turn it into a full IDE.
- In order to test and run the mobile app, Android Studio and its Android Emulator will be used.
- GitHub Desktop will be utilized to interact with the VCS (Version Control System).
- Photoshop will be used to design many of the assets that will be used in the application, such as the background and app icon.
- Google Chrome, Firefox, and Safari will be used to test the web application on different browsers.

5.7. Version Control System

Due to its ease-of-use, suitable features, and popularity, GitHub was chosen as the version control system of the project (Kalliamvakou et al., 2014). Through the use of a VCS, it is possible to keep track of changes in the codebase, while always having a previous point to revert to should a new but broken implementation be too complicated to manually reverse. Considering the complexity and codebase size of CryptoShare, the aforementioned features are invaluable. Furthermore, the GitHub repository acts as an off-site backup should any of the devices used during development stop working. Regular commits ensure no work is ever lost, and that the project can be accessed on any device that supports Git.

6. Project Management

Competent project management can be the difference between a project succeeding or failing (Gido and Clements, 2014). As such, it was important that the project's schedule and sprints be planned only after it was clear what the user requirements were, what the design of the application would look like, and the development approach that would be taken to achieve the objectives of the project. By having the aforementioned information before any detailed planning, it was possible to take into account the developer's experience with the technology, the complexity of the software and codebase, as well as the potential obstacles that could present themselves throughout the project.

6.1. Risk Assessment

During the initial planning and writing of the project initiation document, a risk plan was designed to either prevent or decrease the impact of potential risks that could manifest themselves.

Below, a table containing the identified risks can be found, along with a Risk Assessment Matrix (RAM) to better visualize each risk's impact and probability of occurring.

At the start of the project, the developer was also experiencing severe arm pain, which got better over time. However, it was a major concern at the time, as the developer could barely type for a few minutes before having to stop. This was the risk with the highest score as a result, but thankfully it was not an issue in the end, and the project was completed on time without worsening the condition.

ID	Risk	Probability	Impact	Score	Solution
R1	Tech stack unfamiliarity	3	3	9	Research the different technologies, read through the documentation, and have fallback plans with already familiar technologies.
R2	Poor code quality	2	4	8	Follow OOP principles, use design patterns, remember the importance of user data security etc.
R3	Data loss	1	4	4	Use version control, and regularly commit.
R4	Poor time management	2	3	6	Follow Gantt chart and sprint plans strictly to avoid delays.

R5	Scope creep	3	2	6	Focus on requirements based on user stories. Avoid developing additional features that would only be useful to a minority of users.
R6	Medical issues (arm pain)	3	5	15	Self care, being careful not to code too much too often, and to stick to the Gantt chart.
R7	Unclear requirements	2	4	8	Create realistic user personas, and derive clear and focused user stories based on said personas. Ensure the requirements are both achievable and within the scope of the project.

		Impact				
		1 Little to no effect	2 Tangible effect but not critical	3 Serious impact but not disastrous	4 Significant impact potentially disastrous	5 Severe impact likely disastrous
Probability	1 Very Unlikely	1	2	3	4	5
	2 Somewhat Likely	2	4 R3	6 R4	8 R2, R7	10
	3 Likely	3	6 R5	9 R1	12	15 R6
	4 Highly Likely	4	8	12	16	20
	5 Virtually Certain	5	10	15	20	25

6.2. Gantt Chart

A detailed Gantt chart was designed at the beginning of the project to visualize the schedule of the project. The chart, along with the product backlog shown earlier, were used to keep the project on schedule, and to iteratively develop the project with a holistic approach in mind. The chart was strictly followed and can be seen below.

ID	Task Name	Duration	Start	Finish
1	**Sprint 0 - Planning**	**14 days**	**October 1, 2021**	**October 20, 2021**
2	UI Design Concept Inspiration	5 days	October 1, 2021	October 7, 2021
3	High-Level Planning	4 days	October 6, 2021	October 10, 2021
4	User Stories & Requirements	2 days	October 15, 2021	October 18, 2021
5	Setting Up Environment	2 days	October 19, 2021	October 20, 2021
6	**Sprint 1 - Database (SQLite)**	**14 days**	**October 21, 2021**	**November 9, 2021**
7	Entity Relationship Diagram	5 days	October 21, 2021	October 27, 2021
8	Tables	5 days	October 28, 2021	November 3, 2021
9	Views	4 days	November 4, 2021	November 9, 2021
10	**Sprint 2 - API (Node.js, TypeScript, GraphQL)**	**14 days**	**November 10, 2021**	**November 29, 2021**
11	UML Diagram	2 days	November 10, 2021	November 11, 2021
12	Helper Functions & Classes	3 days	November 12, 2021	November 16, 2021
13	Endpoints	9 days	November 17, 2021	November 29, 2021
14	**Sprint 3 - Bot API (Node.js, TypeScript, Socket.IO)**	**14 days**	**November 30, 2021**	**December 17, 2021**
15	Setup	2 days	November 30, 2021	December 1, 2021
16	WebSocket	3 days	December 2, 2021	December 6, 2021
17	Responses	2 days	December 7, 2021	December 8, 2021
18	Functions	2 days	December 9, 2021	December 10, 2021
19	NLP	6 days	December 10, 2021	December 17, 2021
20	**Sprint 4 - UML, Navigation, Login, Settings**	**14 days**	**December 18, 2021**	**January 5, 2022**
21	UML Diagram	2 days	December 18, 2021	December 20, 2021
22	Navigation & Common UI Elements	3 days	December 21, 2021	December 23, 2021
23	Login	3 days	December 24, 2021	December 28, 2021
24	Encryption	3 days	December 29, 2021	December 31, 2021
25	Settings	4 days	January 1, 2022	January 5, 2022
26	**Sprint 5 - Cryptocurrency Functionality**	**14 days**	**January 6, 2022**	**January 25, 2022**
27	Cryptocurrency Market Data	3 days	January 6, 2022	January 10, 2022
28	Cryptocurrency Holdings	2 days	January 11, 2022	January 12, 2022
29	Cryptocurrency Transactions	3 days	January 13, 2022	January 17, 2022
30	Staking Calculator	3 days	January 18, 2022	January 20, 2022
31	Mining Calculator	3 days	January 21, 2022	January 25, 2022
32	**Sprint 6 - Stock Functionality**	**14 days**	**January 26, 2022**	**February 14, 2022**
33	Stock Market Data	4 days	January 26, 2022	January 31, 2022
34	Stock Holdings	4 days	February 1, 2022	February 4, 2022
35	Stock Transactions	4 days	February 5, 2022	February 9, 2022
36	Dividend Calculator	3 days	February 10, 2022	February 14, 2022
37	**Sprint 7 - Chat Bot, Income Tracker, Budgeting System**	**14 days**	**February 15, 2022**	**March 4, 2022**
38	Chat Bot	4 days	February 15, 2022	February 18, 2022
39	Income Tracker	4 days	February 19, 2022	February 23, 2022
40	Budgeting System	3 days	February 24, 2022	February 28, 2022
41	Tax Calculator	2 days	March 1, 2022	March 2, 2022
42	Insurance Calculator	2 days	March 3, 2022	March 4, 2022
43	Mortgage Calculator	2 days	March 3, 2022	March 4, 2022
44	**Sprint 8 - Testing In Production**	**14 days**	**March 5, 2022**	**March 23, 2022**
45	Database	3 days	March 5, 2022	March 8, 2022
46	API	3 days	March 9, 2022	March 11, 2022
47	Web App	3 days	March 12, 2022	March 15, 2022
48	Desktop App	2 days	March 16, 2022	March 17, 2022
49	Mobile App	4 days	March 19, 2022	March 23, 2022

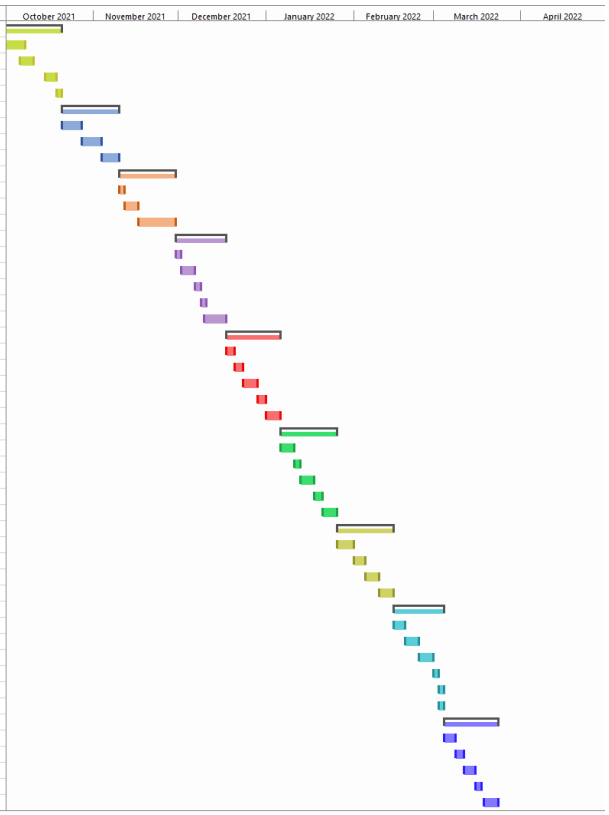


Figure 12 - Gantt Chart

6.3. Kanban Board

Throughout the development phase of the project, an incredibly useful tool known as a “Kanban Board” was used. GitHub Projects provides access to said tool, which can be used to separate tasks from the product backlog into 3 columns, usually called “To Do”, “In Progress” and “Done”. While it seems like a simple concept, it massively impacts productivity, and ensures that development stays on schedule, with every developer knowing exactly which parts of the project are being worked on at any given time (Kniberg and Skarin, 2010). Appendix 1 contains screenshots of all the different boards that were created for the project; there were 7 in total, including: Database, API, Bot, Web App, Desktop App, Mobile App, and Testing.

6.4. GitHub Issues

Another useful GitHub feature that was used extensively was “GitHub Issues”, which allows developers (and users) to create “Issues”, which can be labeled and categorized, and later closed when resolved. Issues can be referred to in commits, and can be used as a to-do list of items that require attention. Throughout the project’s duration, 81 issues were identified in total; 40 of which were bugs, 34 improvements that required additional coding, 3 documentation changes, and 4 that did not fit any categories.

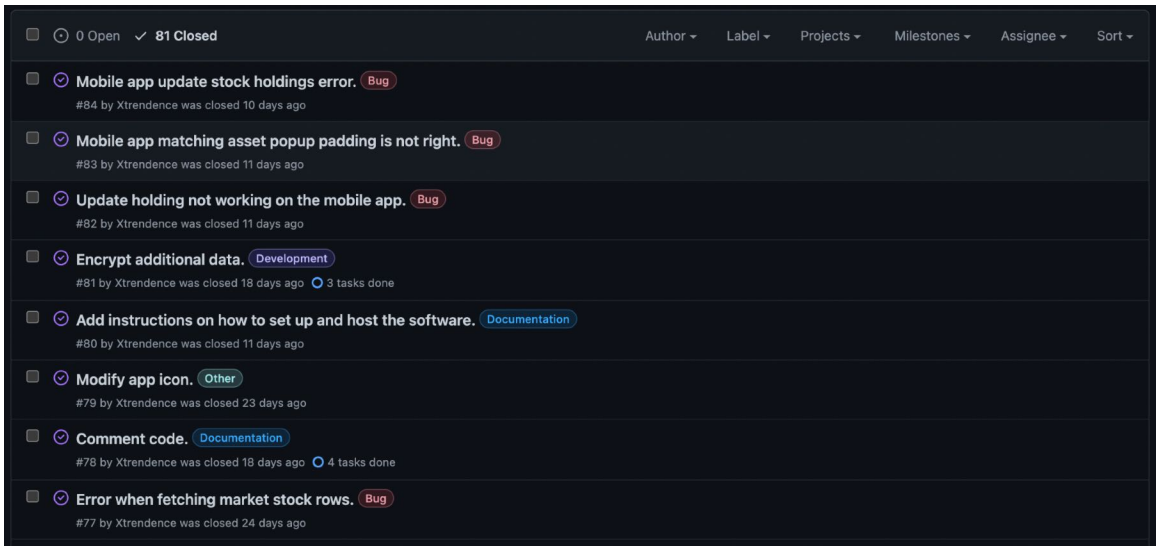


Figure 13 - GitHub Issues List

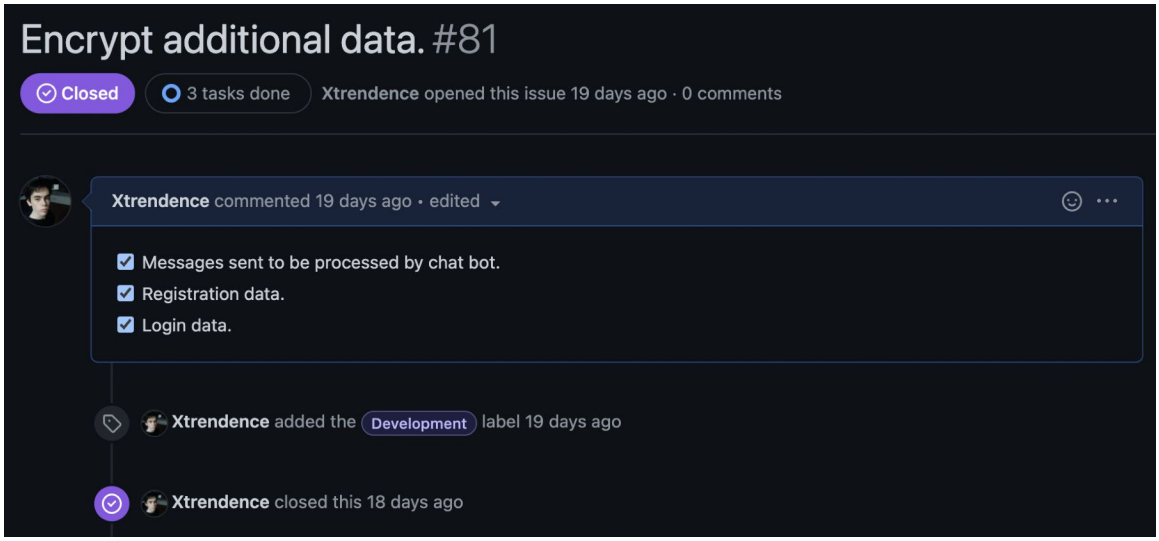


Figure 14 - GitHub Issue #81

As seen above, issues also frequently included a description or checklist, with tasks that needed to be completed in order for the issue to be resolved and closed.

7. Development

The development phase of the project entailed actually utilizing the gathered user requirements and mockups to develop the system with the chosen tech stack, while the UML diagrams and ERD acted as points of reference to know exactly what the software consisted of; this ensured that good software engineering practices were followed.

7.1. Initial Challenges

There were three major obstacles that required a solution before development could begin as they posed potentially severe limitations on the project:

1. The third-party stock market API had a limit of 100 requests per day.
2. Users needed to be able to change their passwords without having to re-encrypt all their data with the new password.
3. Data stored with Electron's implementation of "localStorage" does not persist after quitting the application.

In order to resolve the aforementioned issues, the following ideas were thought of and eventually implemented:

1. Admins hosting an instance of CryptoShare were given an additional option on the settings page to choose between an "internal" and "external" stock market API. The internal one fetches data directly from Yahoo Finance, whereas the external one requires an API key from each user and fetches data from an intermediary that also uses Yahoo Finance. The internal one would be used when CryptoShare is only used by a handful of individuals at most, as the IP address of the host device would be blocked by Yahoo Finance if too many requests are made. Once the number of users reaches a point where they are being rate limited by the API, the admin would switch to the external API, at which point users would have to provide an API key they can get for free. At this point, users would be limited to 100 requests per day. In order to lessen the impact of this, all stock market data is cached and only refreshed once per day. The data is also shared between users, so if user A fetches data on Apple's stock (AAPL), and then user B does the same, user B would be given data cached from when user A made their request. This would essentially be a sort of "crowdsourced" solution, where the more people use CryptoShare, the less likely they are to experience the negative consequences of having to use the external API. Meanwhile individuals who host CryptoShare and are the only user would be able to use it without any API keys or rate limits.

2. When users register an account, an encryption key is generated on the client-side, and is encrypted with their password. This encryption key is then used to encrypt all their data, and is never shared with the API, and only an encrypted copy is stored in the DB. When users change their password, only the encryption key needs to be decrypted and re-encrypted with the user's new password, instead of the entirety of the user's data.
3. An object called "appStorage" was created as part of the facade design pattern. The methods within the object would automatically check which platform the user was on (web or desktop), and use the appropriate storage mechanism accordingly ("localStorage" on the web app, and "electron-store" on the desktop app). This object would act as a drop-in replacement for local storage.

Implementations of the solutions above can be seen in Appendix 2, with more detailed explanations.

7.2. Sprints

The schedule of the project consisted of 9 sprints in total, all of which will be evaluated and discussed below. Each sprint's duration was 2 weeks, with 8 hour work days and 40 hour weeks.

Please note that the tables below contain brief summaries of each sprint's outcome, and that Appendix 2 goes into further technical detail regarding the work that was carried out, the challenges that were faced, as well as the solutions used to overcome them.

Sprint 0	**Start: 01/10/2021**	**End: 20/10/2021**
Tasks	UI Design Concept Inspiration High-Level Planning User Stories & Requirements Setting Up Environment	
Outcome	Sprint 0 marked the initiation of the project, so the tasks mostly involved setting things up and getting ready to start development. During this sprint, a lot of research was done on the potential technologies that could be used. The project was also carefully planned out. Both of these activities proved to save a lot of time and trouble later on, as compatibility between platforms was planned from the start and did not become an issue at any point. The tasks in this sprint were completed earlier than expected, and no challenges were faced.	

Sprint 1	**Start: 21/10/2021**	**End: 09/11/2021**
Tasks	Entity Relationship Diagrams Tables Views	
Outcome	This sprint entailed getting the SQLite database set up, which went according to plan. However, it was discovered that SQLite does not support the “cascade” action by default, requiring the API to handle it.	

Sprint 2	**Start: 10/11/2021**	**End: 29/11/2021**
Tasks	API UML Class Diagram API Helper Functions / Classes API Endpoints	
Outcome	Sprint 2 consisted of API development, which was completed earlier than expected without any issues.	

Sprint 3	**Start: 30/11/2021**	**End: 17/12/2021**
Tasks	Bot Setup Bot WebSocket Bot Responses Bot Functions Bot NLP Functionality	
Outcome	During this sprint, a combination of Natural Language Processing and Regular Expressions were used in order to develop the chat bot as detailed in Appendix 2. Challenges were faced when considering edge cases, where a user might for example ask if they can afford an item, but include the symbol for their currency next to the price of the item. These issues were resolved, and the tasks were completed on time.	

Sprint 4	**Start: 18/12/2021**	**End: 05/01/2022**
Tasks	App UML Class Diagram Login Encryption Settings	
Outcome	Sprint 4 was completed earlier than expected, with minor issues that are explained in Appendix 2.	

Sprint 5	Start: 06/01/2022	End: 25/01/2022
Tasks	Cryptocurrency Market Data Cryptocurrency Holdings Cryptocurrency Transactions Staking Calculator Mining Calculator	
Outcome	This sprint involved more work than the other sprints, mainly due to the fact that charts had to be generated based on the performance of the user's assets, which were derived from their trading activity. Major challenges were faced during the implementation of this feature, but the tasks were completed on time. Appendix 2 contains a discussion regarding this sprint.	

Sprint 6	Start: 26/01/2022	End: 14/02/2022
Tasks	Stock Market Data Stock Holdings Stock Transactions Dividend Calculator	
Outcome	Minor challenges were faced when completing the tasks above, as the stock market data was not compatible with the cryptocurrency market data. The tasks were ultimately finished on time.	

Sprint 7	Start: 15/02/2022	End: 04/03/2022
Tasks	Chat Bot Income Tracker Budgeting System Tax Calculator Insurance Calculator Mortgage Calculator	
Outcome	Implementing the chat bot on mobile posed some challenges, but all the tasks were completed by the end of the sprint without other issues.	

Sprint 8	Start: 05/03/2022	End: 23/03/2022
Tasks	Testing In Production	
Outcome	This sprint consisted of manually testing the production build of each app on physical devices. Any issues were fixed before the end of the sprint.	

8. Testing and Quality Assurance

A number of quality assurance and testing methods were used to ensure the project and its codebase were at a commercial level, and ready to be pushed to production and distributed to be utilized by real users. Due to the developer's previous experience with publishing applications to the Google Play Store, as was the case with CryptoShare's predecessor and several other mobile apps, judging the performance and quality of the application was easier. Despite this, unit, integration, and usability tests allowed for a more objective evaluation.

8.1. Unit and Integration Testing

In order to automate the testing process of the API, 35 unit and integration tests were written to test every API endpoint and functionality. The tests were written with the Jest framework, and the Continuous Integration through GitHub Actions was set up to run them every time a commit was made. This ensured that the API remained functional as new features were added, and parts of it were modified.

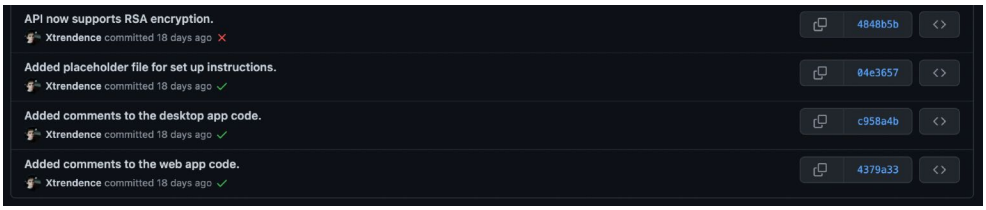


Figure 15 - GitHub Commits Showing Continuous Integration

```
● API Testing › RESTful API Tests › Verify token.

  expect(received).toEqual(expected) // deep equality

  Expected: "AnotherTest"
  Received: undefined

    282 |             test("Verify token.", async () => {
    283 |                     let verify: any = await requests.verifyToken(user.userID, user.token);
  > 284 |                     expect(verify.username).toEqual("AnotherTest");
        |                                             ^
    285 |             });
    286 |
    287 |             test("Logout.", async () => {

    at tests/api.test.ts:284:28
    at fulfilled (tests/api.test.ts:24:58)
```

Figure 16 - Automated Test Results

As seen in the images above, CI proved to be invaluable during development, as it was made abundantly clear exactly which tests failed and why. A full description of each test and its results can be seen in Appendix 3.

8.2. Usability Testing

Usability tests were carried out at the end of every other sprint starting from sprint 4, which was the first sprint to involve any user interface development. Participants were asked to complete a variable number of high-level tasks related to the features that were added that sprint. Any problems identified during the session were added as GitHub Issues, and were either resolved as soon as possible, or at the end of the subsequent sprint depending on the priority of the issue. The tests were carried out over video calls, where the participant would share their screen, and notes would be taken based on their performance and thoughts. To test the mobile app, the participants were given control over the developer's device, as asking participants to set up an Android emulator was deemed to be an unreasonable expectation.

In order to comply with the codes of ethical conduct, participants were asked for consent before starting the session, and any notes taken did not include any information that could identify the participants. The full results of each usability test can be found in Appendix 4. A total of 6 participants took part in the test; 2 at the end of sprint 4, another 2 at the end of sprint 6, and the last 2 at the end of sprint 8. During each test, one participant used the mobile app, and another the web app. The API and web app were hosted for them, and they were given a URL they could use to access it without having to install and set up Node.js for themselves.

8.3. Production Build Testing

Aside from the automated white box testing done with the unit and integration tests, manually testing the final release/product build was equally as important. As such, once the coding for the actual software had been completed, an entire final sprint was set aside to thoroughly test the production build of the application for each platform. Edge cases and corner cases were tested for, and the software itself was tested to ensure it ran as expected on physical devices, and on other operating systems; this is due to the fact that the majority of the development was done using Chrome for the web app, a laptop running macOS for the desktop app, and an Android emulator for the mobile app.

The production testing, however, was done on 2 additional browsers, including FireFox and Safari. The desktop app was tested on a Windows machine, and a Linux virtual machine. Finally, the mobile app was tested on a real Android phone. The tests were successful, and no major issues were found. Appendix 5 contains the full test results, including tests and modifications made to ensure the application complied with the Web Accessibility Initiative guidelines through the use of a WAI evaluation tool.

9. Project Closure

“Project closure is a time to take stock, assess the successes and failures, and take learnings that you can transfer to future projects” (Rowland, 2019). Indeed, evaluating the performance and state of a project is invaluable when it comes to actually learning and improving one’s abilities.

9.1. Aims and Objectives Review

The overall goal of CryptoShare was to provide its users with a transparent and secure piece of software that they could use to manage their finances, including their cryptocurrency and stock investments. While this goal was achieved, it is important to revisit the objectives that were identified at the start of this report.

• *Objective 1: Research the market and feedback obtained from CryptoShare’s predecessor to help identify user requirements.*

As seen in sections 2 and 4, this objective was accomplished, and key missing features were found in existing solutions. Furthermore, CryptoShare’s predecessor was used to gain insight into what users want in a financial portfolio application.

• *Objective 2: Research the legal, social, ethical and professional requirements.*

This objective was met in section 3 by researching the relevant laws and regulations surrounding FinTech applications. The social, ethical, and professional implications of the project were also explored and discussed.

• *Objective 3: Develop a self-hosted and performant API that can be easily deployed on any system without much setup necessary for server-side functionality.*

As evidenced by the content of section 7, Appendix 6, and Appendix 8, a GraphQL API was developed, with SQLite as its DBMS, which made deploying the CryptoShare API incredibly easy, as the user would only need Node.js installed without any other additional software or configuration required.

• *Objective 4: Develop a production-ready, free, and open-source unified web, desktop, and mobile application for client-side functionality.*

Similarly to the previous objective, section 7 and Appendix 8 show that this objective was met, and a fully working application was developed for every intended platform.

9.2. Project Post-Mortem

As part of the project post-mortem, the development approach, including the technologies and tools used to develop the project will be evaluated. Additionally, the project management approach, and its general direction will be discussed to identify its performance. Finally, a personal reflection will be provided in order to explore some personal opinions regarding the project, as well as any meaningful learnings throughout the months.

9.2.1. Tech Stack Evaluation

A great deal of research and thought was put into the development approach chosen for this project, including the tech stack. With the goal of creating a unified interface, having a tech stack with highly compatible technologies was an absolute necessity. This eventually proved to provide a significant boost in productivity and efficiency, as many of the functions written for one platform could simply be used in the application of another platform without having to rewrite the code. This not only sped up the development process, but also debugging and testing. The choice to avoid templates and frameworks for the web app and, by extension, the desktop app, also proved to be the ideal approach, as a custom user interface could be created from scratch that would be uniform across all platforms; the use of React Native for the mobile app was especially helpful in this area, as the stylesheets used in the framework are an abstraction that is similar to CSS.

GraphQL was, perhaps, one of the best pieces of technology in the entire tech stack, as it made API development completely painless and highly organized. Paired with SQLite, the whole API was made incredibly easy to set up and deploy.

Using RSA and AES as the asymmetric and symmetric encryption algorithms respectively ensured that compatible cryptographic libraries could be found for every platform due to the popularity of the algorithms. Minor issues were faced, as discussed in Appendix 2, but this was, overall, a success in terms of providing security and privacy.

9.2.2. Project Management Evaluation

Despite every sprint being completed in time without delays, it is worth pointing out that the project's schedule and workload were incredibly demanding. A conscious decision was made to plan the project only after having decided exactly what development approach and which technologies would be used, and what the requirements were. This decision was, perhaps, one of the only reasons the project was completed on time.

By having a clear understanding of the workload, it was possible to better plan the minutiae of the project, and to have a realistic estimate of how long each task would take. Additionally, the use of GitHub Projects as a Kanban Board made it possible to decompose the problem into smaller parts that could be tackled without being overwhelming. The flexibility and iterative nature of the Agile methodology also made it easy to rectify any issues with the software, and to have a working application at the end of every sprint; this also made it simpler to explain how much progress was made during each standup meeting with the supervisor of the project as shown in Appendix 7. Looking at the commit chart provided by GitHub, it is possible to see that the project was worked on regularly, with sprints involving the majority of client-side functionality having the most commits (January through February):

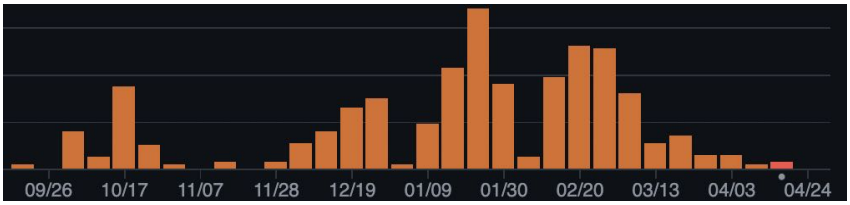


Figure 17 - Commit Frequency

9.2.3. Personal Reflection

I can say, without a doubt, that this project was one of the best learning experiences. Before starting my university education, I had around 3-4 years of programming experience from working on solo projects, so seeing the difference and progress in this project was incredibly rewarding. While I had worked on smaller scale projects that I published to the Google Play Store, and released on GitHub, I had never developed a project as big and complex as this one. Furthermore, previous projects had always been rather disorganized and unprofessional, whereas this one was fully planned from about a month or two before the start of my final year, and was regularly worked on with a schedule I adhered to at all times. When deciding which technologies to use, I was fully aware of the fact that I could select ones I was already highly familiar with, and know with full certainty that I could avoid having to put the effort in to learn new things. However, I wanted my final project to be something I could learn a lot from, and something I could actually be proud of. GraphQL, NLP, SQLite, TypeScript and, to an extent, React Native, were all technologies I had no previous experience with; they were all selected not only due to the fact that they perfectly fit the requirements of the project, but also due to their popularity in the industry at the moment. Skills learned from this project will be invaluable when looking for a job, and I cannot wait to show and talk to them about CryptoShare. I am also very excited to actually publicly release this software, as I know there are quite a lot of people who have been interested in the project since I stopped development on its predecessor to focus on this one.

9.2.4. Areas of Improvement and Additional Features

There are a number of features in the final state of the software that could be improved upon or added if given additional time. Namely, the chat bot could be fleshed out to include a host of new features and functionalities. In its current state, it requires the user to formulate their instructions in very specific ways that may not feel natural to them. Furthermore, it does not have any conversational responses, and only performs tasks. Another idea that would have been implemented with additional time is the usage of a secondary API for both stock and cryptocurrency market data. Currently, only CoinGecko and Yahoo Finance are used. Using a secondary API would allow for a fallback in case the primary ones are not responding, in addition to faster data refresh/refetch times.

9.3. Conclusions

Overall, the objectives that were set out at the initiation of the project were met, and the time spent on researching and choosing an appropriate tech stack paid off; the in-depth analysis of the current market proved to be worthwhile, as the finished product offered features that current solutions do not.

Before starting development, the relevant legal, social, ethical, and professional issues were also thoroughly researched to ensure that the application complied with all the relevant laws and regulations, while being developed in a responsible and ethical manner wherever and whenever possible.

Highly detailed plans, along with a clear project management approach made development progress smoothly. One of the key contributors in this sense was also the approach taken when it came to solving problems; instead of using trial and error to solve any major issues, such as the ones pointed out at the start of section 7, solutions were planned out and designed before beginning any sort of actual implementation. Thanks to the methodical approach taken, by the end of the project, a fully working, open-source, self-hosted, and cross-platform application that individuals can use to manage their finances was developed. Furthermore, the rigorous testing and iterative development process resulted in a more refined final product that had been tested for usability every major step of the way as part of a more user-centric development process.

As previously mentioned, if given additional time, there were certain features that would have been worthwhile to improve upon or add.

References

Africa.cardano.org. 2022. Cardano. [online] Available at: <https://africa.cardano.org/> [Accessed 27 March 2022].

Apollo Docs. 2022. Why adopt GraphQL?. [online] Available at: <https://www.apollographql.com/docs/intro/benefits/> [Accessed 7 April 2022].

Arias, D., 2021. Hashing in Action: Understanding bcrypt. [online] Auth0. Available at: <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/> [Accessed 8 April 2022].

Baus, A., 2022. Important regulations in fintech developers need to comply with. [online] DECODE. Available at: <https://decode.agency/article/fintech-regulations-for-developers/> [Accessed 6 April 2022].

"Bitcoin Is Dead." 2022. "Bitcoin Is Dead". [online] Available at: <https://www.bitcoinisdead.org/> [Accessed 27 March 2022].

Bugl, D., 2017. Learning Redux. Packt Publishing Ltd.

Coinmarketcap.com. 2022. Global Cryptocurrency Market Charts. [online] Available at: <https://coinmarketcap.com/charts/> [Accessed 27 March 2022].

Dryka, M. and Pluszczewska, B., 2020. 5 Reasons Why You Should Build Electron Desktop App. [online] Brainhub.eu. Available at: <https://brainhub.eu/library/electron-desktop-app/> [Accessed 8 April 2022].

GDPR. 2016. GDPR. [online] Available at: <https://gdpr-info.eu/> [Accessed 6 April 2022].

Gido, J. and Clements, J., 2014. Successful project management. Cengage Learning.

Goldman Sachs. 2022. Goldman Sachs. [online] Available at: <https://www.goldmansachs.com/> [Accessed 27 March 2022].

Google Trends. 2022. Google Trends. [online] Available at: <https://trends.google.com/trends/explore?date=today%205-y&geo=GB&q=cryptocurrency,stocks> [Accessed 27 March 2022].

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M. and Damian, D., 2014, May. The promises and perils of mining github. In Proceedings of the 11th working conference on mining software repositories (pp. 92-101).

Kennon, J., 2022. Remember the Most Important Rule of Investing. [online] The Balance. Available at: <https://www.thebalance.com/the-most-important-rule-of-investing-357325> [Accessed 6 April 2022].

Kniberg, H. and Skarin, M., 2010. Kanban and Scrum-making the most of both. Lulu. com.

Legislation.gov.uk. 2018. Data Protection Act. [online] Available at: <https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted> [Accessed 6 April 2022].

Legislation.gov.uk. 1995. Disability Discrimination Act. [online] Available at: <https://www.legislation.gov.uk/ukpga/1995/50/contents> [Accessed 6 April 2022].

Liu, S., 2021. Cross-Platform Mobile Frameworks Used By Global Developers 2020 | Statista. [online] Statista. Available at: <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>.

Maguire, M. and Bevan, N., 2002, August. User requirements analysis. In IFIP world computer congress, TC 13 (pp. 133-148). Springer, Boston, MA.

Manchanda, A., 2021. Cross-Platform App Frameworks In 2021. [online] Insights - Web and Mobile Development Services and Solutions. Available at: <https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/>.

Martin, S., 2021. Flutter Vs React Native Vs Xamarin: Which Is The Best Choice For 2019?. [online] Medium. Available at: <https://towardsdatascience.com/flutter-vs-react-native-vs-xamarin-which-is-the-best-choice-for-2019-2bc30fc76442>.

Puzhevich, V., 2020. How to Choose the Right Technology Stack for Your Project. [online] SCAND. Available at: <https://scand.com/company/blog/choosing-a-technology-stack/> [Accessed 7 April 2022].

Quiroz-Gutierrez, M., 2022. El Salvador rakes in 30% boost to crypto tourism, but there's a dark side. [online] Fortune. Available at: <https://fortune.com/2022/02/23/el-salvador-bitcoin-law-tourism-up-30-percent-imf-senate/> [Accessed 27 March 2022].

Rowland, M., 2019. How to manage project closure. [online] Apm.org.uk. Available at: <https://www.apm.org.uk/blog/how-to-manage-project-closure/> [Accessed 9 April 2022].

Socket.io. 2022. Socket.IO. [online] Available at: <https://socket.io/docs/v4/#what-socketio-is> [Accessed 8 April 2022].

Stelzer, T., 2020. Investing in a coronavirus world: A brave new dawn?. [online] Finder. Available at: <https://dvh1deh6tagwk.cloudfront.net/finder-us/wp-uploads/sites/3/2019/06/investing_in_a_coronavirus_world_paper.pdf> [Accessed 27 March 2022].

WebMD. 2022. When the Stock Market Gets You Down. [online] Available at: <https://www.webmd.com/balance/features/when-stock-market-gets-you-down> [Accessed 6 April 2022].

Windows Developer Blog. 2016. When to use a HTTP call instead of a WebSocket. [online] Available at: <https://blogs.windows.com/windowsdeveloper/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/> [Accessed 8 April 2022].

Wurmser, Y., 2020. Apps Far Outpace Browsers in US Adults' Mobile Time Spent. [online] Insider Intelligence. Available at: <https://www.emarketer.com/content/the-majority-of-americans-mobile-time-spent-takes-place-in-apps> [Accessed 11 April 2022].

Yahoo Finance. 2022. S&P 500 INDEX. [online] Available at: <https://finance.yahoo.com/quote/%5Espx/> [Accessed 27 March 2022].

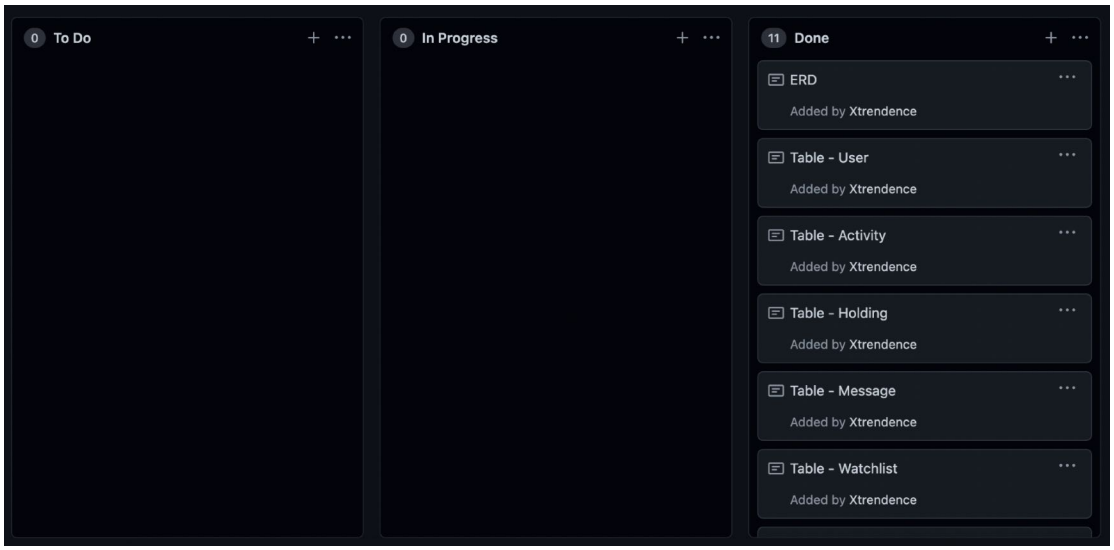
Appendices

Below, several appendices have been provided to either go into more detail about certain aspects of the project, or to offer additional yet useful information in general.

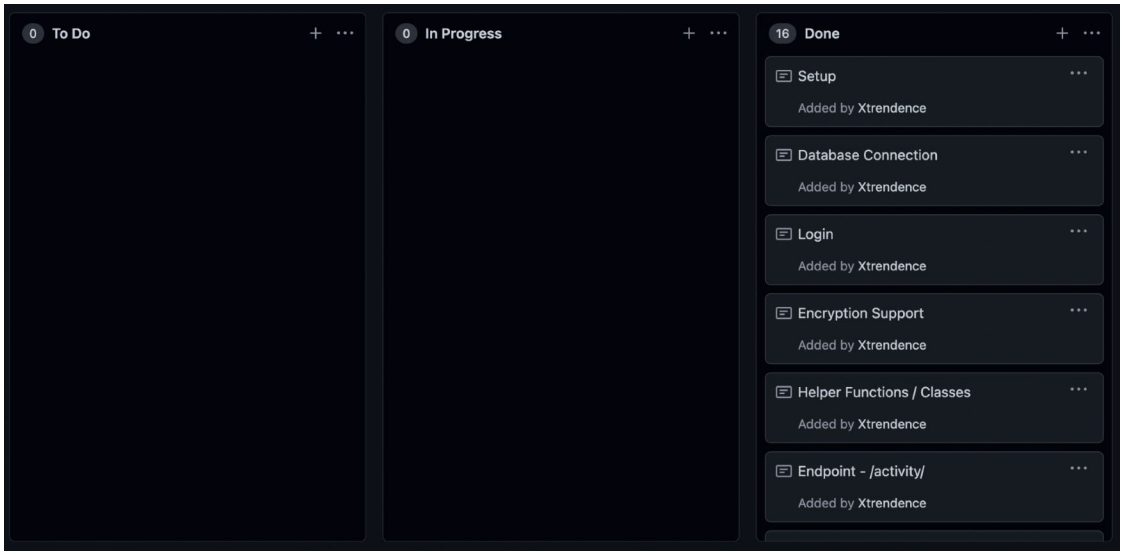
Appendix 1: Kanban Boards

A total of 7 project boards were created using GitHub Projects. At the start of each sprint, the relevant tasks would be moved to the “In Progress” column. Once a task was completed, it was moved to the “Done” column. The screenshots below were taken at the end of the project, so all of the tasks appear in the rightmost column.

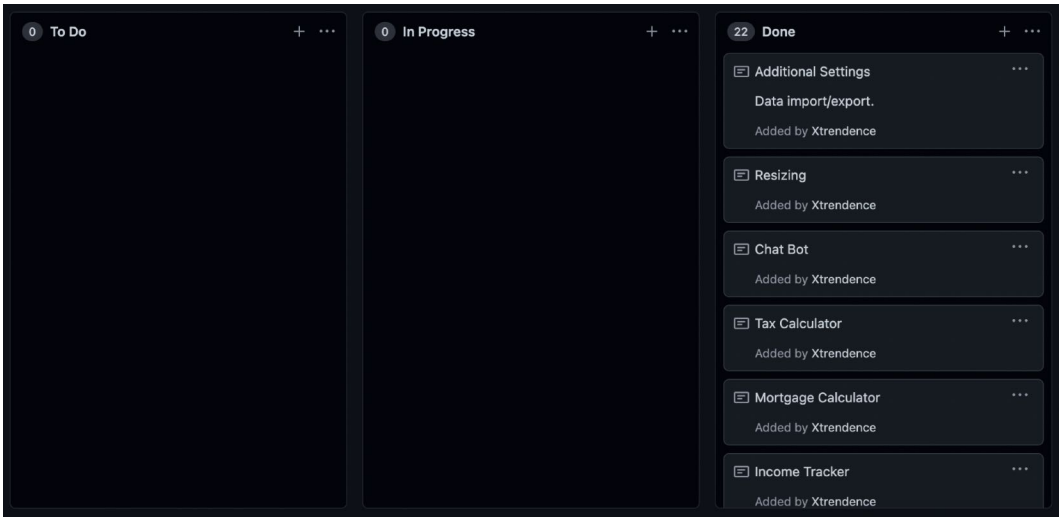
Database



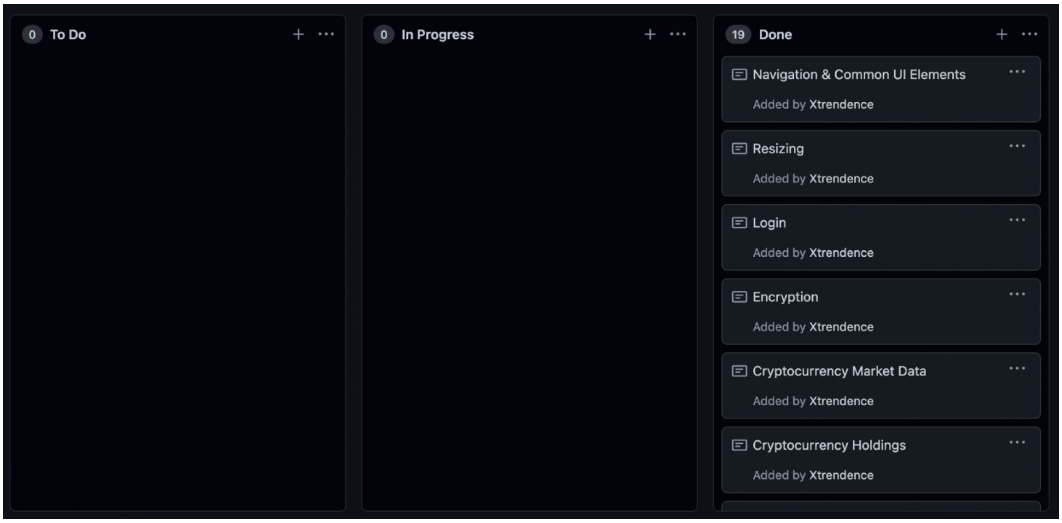
API



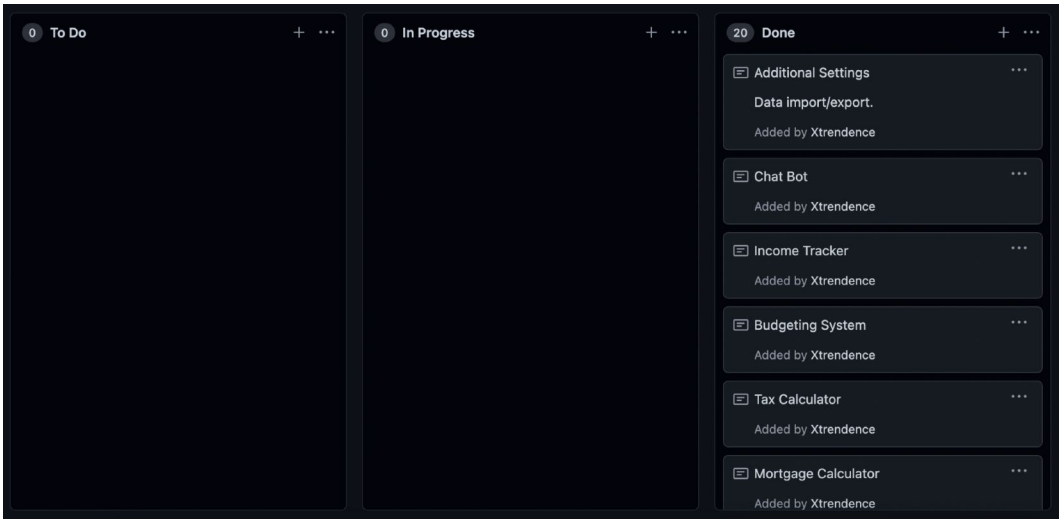
Web App



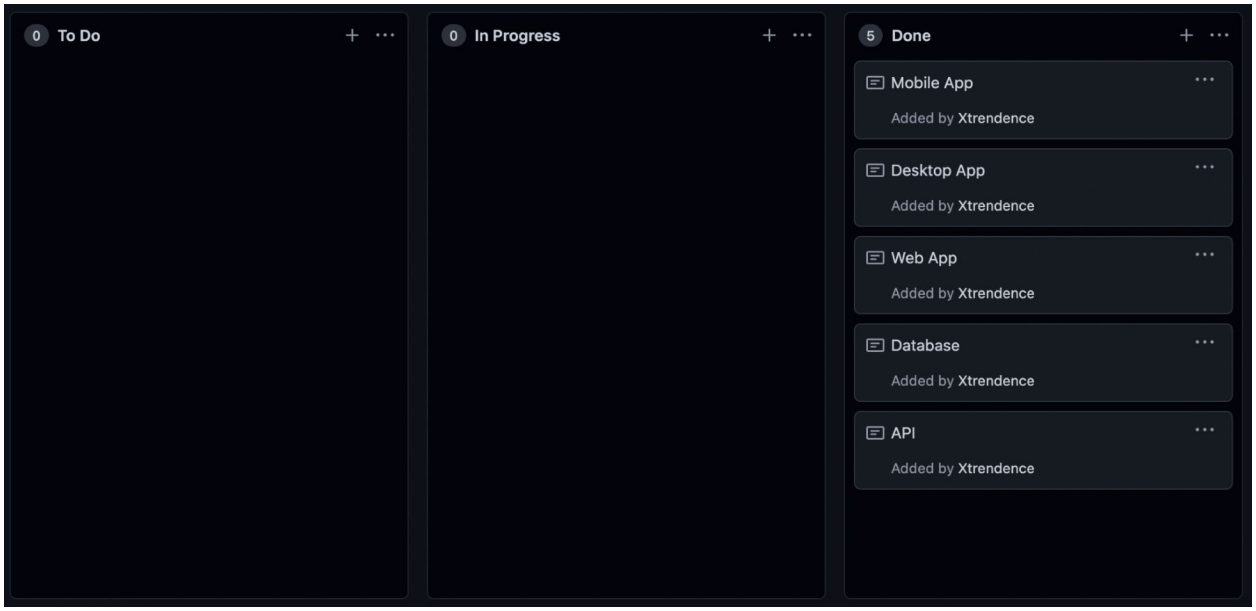
Desktop App



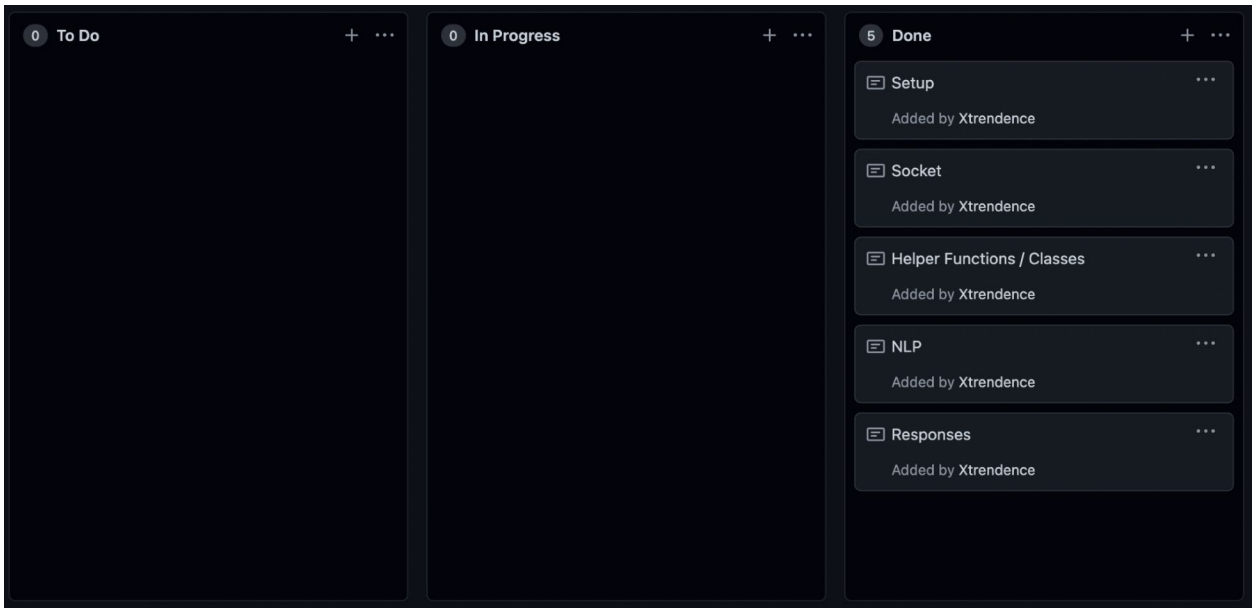
Mobile App



Testing


0 To Do
0 In Progress
5 Done
Mobile App
Added by Xtrendence
Desktop App
Added by Xtrendence
Web App
Added by Xtrendence
Database
Added by Xtrendence
API
Added by Xtrendence

Bot


0 To Do
0 In Progress
5 Done
Setup
Added by Xtrendence
Socket
Added by Xtrendence
Helper Functions / Classes
Added by Xtrendence
NLP
Added by Xtrendence
Responses
Added by Xtrendence

Appendix 2: Software Engineering

Considering CryptoShare's codebase has more than 100,000+ lines of code, only the most noteworthy parts will be analyzed below. These include solutions to major challenges faced during development, and innovative approaches towards handling limitations and other issues.

Stock Market API Limitations

The first step in solving the 100 requests per day limitation of the stock API was to provide two separate approaches that the admin could choose from. The internal API would fetch data from the Yahoo Finance API directly, but needed to run on the server-side to work. As a result, if a CryptoShare host were to have 100 users, and they all had 10 stocks, then refreshing their page would cause 1,000 requests to be sent to the Yahoo Finance API, which would result in the host device's IP being blacklisted. To solve this, a third-party API was used, which requires an API key from each user.

```
// The CryptoShare API uses two third-party APIs for stock market data. The first one is the Yahoo Finance API directly,
the second is an API that also uses the Yahoo Finance API, but requires an API key to work. This is required because if
CryptoShare is used by multiple people, then its API would be making potentially hundreds of requests to the Yahoo Finance
API directly, which might result in rate limits for all users. The admin can switch the API to the external one through the
web app or desktop app's settings page, at which point each user would require an API key from the external API provider to
access stock market functionality.
let stockAPITypes = {
    internal: `http://localhost:${Utils.portAPI}`,
    external: "https://yfapi.net"
};
```

When a user asks for the market data of a stock, it is cached in the SQLite DB of the server:

```
if(row === undefined || !Utils.validJSON(row.historicalData) || Utils.refetchRequired(JSON.parse
(row.historicalData).time)) {
    try {
        let historicalData: any = await getHistoricalData(assetSymbol, keyAPI);

        if(Utils.validJSON(historicalData) && "error" in JSON.parse(historicalData)) {
            reject(`!${JSON.parse(historicalData).error}!`);
            return;
        }

        db.runQuery("INSERT OR REPLACE INTO Stock (assetSymbol, historicalData, priceData) VALUES (?
        , ?, ?)", [assetSymbol, historicalData, row?.priceData || ""]);

        let stock = new Stock(assetSymbol, historicalData, row?.priceData || "");
        resolve(stock);
    } catch(error) {
        console.log(error);
        reject(error);
    }
} else {
    let stock = new Stock(row.assetSymbol, row.historicalData, row.priceData || "");
    resolve(stock);
}
```

When market data is requested, the API checks whether the cached data is older than 24 hours. If it is, then the data is fetched again from the stock market API:

```
let refetch: any = await getSymbolsToRefetch(symbols) || {};
let refetchSymbols = Object.keys(refetch);

try {
    if(refetchSymbols.length > 0) {
        let priceData: any = await getPriceData(refetchSymbols, keyAPI);

        if("error" in priceData) {
            reject(`!${priceData.error}!`);
            return;
        }

        let fetchedSymbols = Object.keys(priceData.priceData);

        fetchedSymbols.map(symbol => {
            db.db?.get("SELECT * FROM Stock WHERE assetSymbol = ?", [symbol], async (error, row) => {
                if(error) {
                    console.log(error);
                } else {
                    if(row !== undefined) {
                        db.runQuery("INSERT OR REPLACE INTO Stock (assetSymbol, historicalData,
                        priceData) VALUES (?, ?, ?)", [symbol, row.historicalData || "", JSON.stringify(
                        { time:priceData.time, priceData:priceData.priceData[symbol] })]);
                    } else {
                        db.runQuery("INSERT OR REPLACE INTO Stock (assetSymbol, historicalData,
                        priceData) VALUES (?, ?, ?)", [symbol, "", JSON.stringify({ time:priceData.
                        time, priceData:priceData.priceData[symbol] })]);
                    }
                }
            });
        });
    }
} catch(error) {
    console.log(error);
}
```

Otherwise, cached data is returned:

```
// Returns cached stock data.
function getStocks(symbols: any) {
    return new Promise((resolve, reject) => {
        try {
            let output: any = [];

            for(let i = 0; i < symbols.length; i++) {
                let symbol = symbols[i].toUpperCase();

                db.db?.get("SELECT * FROM Stock WHERE assetSymbol = ?", [symbol], async (error, row) => {
                    if(error) {
                        console.log(error);
                    } else {
                        if(row !== undefined) {
                            let stock = new Stock(symbol, row.historicalData || "", row.priceData);
                            output.push(stock);
                        }
                    }
                });
            }

            let check = setInterval(() => {
                if(output.length === symbols.length) {
                    resolve(output);
                    clearInterval(check);
                }
            }, 100);
        } catch(error) {
            reject(error);
        }
    });
}
```

To determine whether the cached data needs to be re-fetched, a UNIX timestamp is saved alongside the data when it is first fetched. A function is then used to check this timestamp, and see if it is older than a predefined number of seconds:

```
// Determines if a UNIX timestamp is older than a predefined number of seconds.
static refetchRequired(time: string) {
    let refetchTime = this.marketRefetchTime;
    return (Math.floor(new Date().getTime() / 1000)) - refetchTime > parseInt(time);
}
```

By default, data is re-fetched every 24 hours:

```
// Default: 3190.
static portAPI: number = 3190;

// Default: 86400 (24 hours).
static marketRefetchTime: number = 86400;

static db: DB | undefined;
static dataFolder: string = "./data/";
static dbFile: string = path.join(this.dataFolder, "data.db");
static adminFile: string = path.join(this.dataFolder, "adminSettings.txt");
static publicKeyFile: string = path.join(this.dataFolder, "public.key");
static privateKeyFile: string = path.join(this.dataFolder, "private.key");

static defaultAdminSettings: any = {
    stockAPIType: "internal",
    userRegistration: "enabled"
};
```

Since the cached data is not tied to the user requesting the data, users can share market data in order to provide a sort of “crowdsourced” solution. So if there are 100 users, and the admin has set the application to use the external server, then they all need to provide an API key. This would mean they each get 100 requests per day. However, if they all have AAPL (Apple) as a stock holding, for example, then only 1 of them would use a request to fetch the market data; the others would simply use the data that was cached when the user originally made the request.

Changing Passwords Without Re-Encrypting Data

After discussing the matter with Nathan Clarke, a professor in cybersecurity and digital forensics at the University of Plymouth, he suggested the use of a separate encryption key instead of using the user's password to encrypt their data. By doing this, a user could change their password, and the only piece of data that would need to be re-encrypted would be their encryption key.

The first step was writing a function that would output a cryptographically random key:

```
/**
 * Generates a random password, and a cryptographically random salt, from which a 256-bit AES key is derived.
 * @returns {string} - The AES key.
 */
static generateAESKey() {
    let result = "";
    let characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
    let charactersLength = characters.length;

    for(let i = 0; i < charactersLength; i++) {
        result += characters.charAt(Math.floor(Math.random() * charactersLength));
    }

    let salt = CryptoJS.lib.WordArray.random(128/8);

    return CryptoJS.PBKDF2(result, salt, { keySize: 256/32 }).toString(CryptoJS.enc.Base64);
}
```

Then, on the client-side, the user's key would be encrypted with their password when creating an account:

```
let key = CryptoFN.generateAESKey();
let encrypted = CryptoFN.encryptAES(key, inputCreatePassword.value);

createAccount(username, inputCreatePassword.value, encrypted).then(async result => {
    if(result.data.createUser === "Done") {
        await appStorage.setItem("key", key);
        await appStorage.setItem("firstLogin", "true");

        popup.hide();

        buttonExistingAccount.click();

        inputLoginUsername.value = username;
        inputLoginPassword.value = inputCreatePassword.value;

        buttonLoginAccount.click();

        successNotification("Account Created", "You will now be logged in...");
    } else {
        errorNotification(result.data.createUser);
    }
}).catch(error => {
    errorNotification(error);
    console.log(error);
});
```

When changing their password, their key would be encrypted using the new password:

```
if(newPassword === repeatPassword) {
    let key = await appStorage.getItem("key");

    if(empty(key)) {
        errorNotification("Couldn't change encryption key.");
        return;
    }

    let encrypted = CryptoFN.encryptAES(key, newPassword);

    changePassword(userID, token, encrypted, currentPassword, newPassword).then(result => {
        if("error" in result) {
            errorNotification(result.error);
        } else {
            if("username" in result) {
                successNotification("Password Changed", "Your password has been changed.");
```

Their details would then be encrypted using RSA, and decrypted on the server-side (this would ensure security during transit):

```
app.post("/changePassword", async (request, response) => {
    try {
        let keys: any = await Utils.checkKeys();

        let userID = request.body.userID;
        let token = request.body.token;
        let key = request.body.key;

        let encryptedCurrentPassword = request.body.currentPassword;
        let encryptedNewPassword = request.body.newPassword;

        let currentPassword = await CryptoFN.decryptRSA(encryptedCurrentPassword, keys.privateKey);
        let newPassword = await CryptoFN.decryptRSA(encryptedNewPassword, keys.privateKey);

        response.send(await Utils.changePassword(userID, token, key, currentPassword, newPassword));
    } catch(error) {
        response.send({ error:error });
    }
});
```

Finally, the API would replace the existing key in the DB with the new one.

Desktop App Data Not Persisting

In order to use a different storage mechanism based on whether the user was on the web app or desktop app, the ID attribute of the HTML tag was used:

```
// The app's platform ("web" or "app") depends on the ID of the document element.
const appPlatform = document.documentElement.id;

// In order to have the same codebase across the web and desktop app, there are a few empty variables that are only given a
value when the app platform is set to "app".
let electron = null;
let ipcRenderer = null;

if(appPlatform !== "app" && typeof require === "undefined") {
    var require = () => {
        return "";
    };
}

if(appPlatform === "app") {
    var sha256 = require("sha256");
    window.$ = window.jQuery = require("jquery");
    electron = require("electron");
    ipcRenderer = electron.ipcRenderer;
}
```

Then, an object was created as a drop-in replacement for “localStorage” by offering the same functions, but while differentiating between the platform being used:

```
// On the desktop app, localStorage doesn't persist. Instead of using an if statement every time localStorage is used (or
"electron-store" accessed through the "ipcRenderer" on the desktop app), an object called "appStorage" is used as a drop-in
replacement for localStorage that takes care of everything.
const appStorage = {
    setItem(key, value) {
        return new Promise(async (resolve, reject) => {
            try {
                if(appPlatform === "web" || appBypass()) {
                    localStorage.setItem(key, value);
                    resolve();
                } else {
                    ipcRenderer.invoke("setItem", { key:key, value:value }).then((response) => {
                        resolve(response);
                    }).catch(error => {
                        reject(error);
                    });
                }
            } catch(error) {
                reject(error);
            }
        });
    },
```

When the desktop app is used, the data is sent to the local Electron server, which uses the “electron-store” library to store the data on the user’s drive.

Incompatibility Between Cryptocurrency and Stock Market Data

During sprint 6, a minor issue was faced. Since cryptocurrencies are traded 24/7, while stocks are only traded during business hours, the market data was incompatible. A function was written to fill in the gaps in time:

```
// Converts historical stock market data to match the format of historical crypto market data (for example, stocks aren't traded on
weekends, so there's only 5 data points per week).
function parseStockHistoricalDataAsCrypto(days, historicalData) {
    let parsed = [];
    let parsedObject = {};
    let datedPrices = {};
    let prices = historicalData?.indicators?.quote[0]?.close;
    let times = historicalData?.timestamp;

    for(let i = 0; i < times.length; i++) {
        let date = formatDateHyphenated(new Date(times[i] * 1000));
        datedPrices[date] = prices[i];
    }

    for(let i = 0; i < days.length; i++) {
        let day = days[i];
        let currentTime = times[i] * 1000;
        let currentDay = formatDateHyphenated(new Date(currentTime));
        let currentPrice = datedPrices[currentDay];
        parsedObject[currentDay] = [currentTime, currentPrice];

        if(!(day in parsedObject)) {
            let value = previousValueInObject(parsedObject, i - 1);
            if(empty(value)) {
                value = nextValueInObject(parsedObject, i);
            }

            parsedObject[day] = value;
        }

        let time = parsedObject[day][0];
        let price = parsedObject[day][1];

        parsed.push([time, price]);
    }

    return parsed;
}
```

By doing so, stock market data could be passed to the same function that generates charts for cryptocurrency prices.

Incompatibility Between Encryption Algorithms

During sprint 4, it was discovered that data encrypted on the web app could not be decrypted on the server-side. This was due to the cryptography library used on the Node.js API being different to the client-side one. As a result, the padding mode used on the web app was not supported, and had to be changed:

```
/**
 * @param {string} plaintext - The string to encrypt.
 * @param {string} publicKey - The public RSA key to encrypt the data with.
 * @returns {string} - A Base64 encoded version of the ciphertext.
 */
static encryptRSA(plaintext, publicKey) {
    return new Promise((resolve) => {
        if(appPlatform === "app") {
            forge = require("node-forge");
        }

        publicKey = forge.pki.publicKeyFromPem(publicKey);
        resolve(btoa(publicKey.encrypt(plaintext, "RSA-OAEP")));
    });
}
```

Calculating Holdings Based on Trading Activity

During sprint 5, a major challenge was faced, as calculating the holdings of a user based on their trading activity was much more difficult than anticipated. Due to the length of the code, a link will be provided below to the function instead of screenshots:

https://github.com/Xtrendence/CryptoShare/blob/main/web/assets/js/functions/holdings.js#L775

Comments are included to explain what the code is doing.

Natural Language Processing Functionality

A third-party library called “node-nlp” was used in order to develop CryptoShare’s chat bot. Tokenization is used to break sentences down into smaller tokens, which are then processed in order to extract useful entities and pieces of data. To do so, multiple extraction techniques are used, such as number extraction and date extraction.

For example, if the user types the sentence “I bought 5 BTC yesterday”, the chat bot first breaks down the sentence into tokens, and extracts the date. Utilizing the Named Entity Recognition included with the library, the word “yesterday” is converted into a date (e.g. if today is 24-03-2022, then the word yesterday would be converted into 23-03-2022). Subsequently, the rest of the sentence is processed using regular expressions. First, the type of activity is identified by checking whether the sentence contains the words “bought”, “buy”, “sold”, “sell”, or words that imply a transfer rather than a trade.

```
case utterance.match("(bought|buy)")?.input:
    category = "activity-or-transaction";
    action = "buy";
    break;
case utterance.match("(sold|sell)")?.input:
    category = "activity-or-transaction";
    action = "sell";
    break;
case utterance.match("(transfer|transferred|sent|send|received)")?.input:
    category = "activity-or-transaction";
    action = "transfer";
    break;
```

Once the user’s intent has been identified, the relevant regular expression is used to extract further data from the sentence. The sentence is also checked to determine whether or not the user has provided the price at which they bought or sold an asset (this is not a required piece of data, as historical market data can be used instead).

```
if(intent.action.match("(buy|sell)") && !intent.utterance.match("(at|@)") && intent.utterance.match("(for)")) {
    regex = /\w+(?=\s+((for )\$?[0-9]\d*\.?\d))/;
    valueGiven = true;
}

if(intent.action.match("(buy)") && !intent.utterance.match("(at|@)") && !intent.utterance.match("(for)")) {
    regex = /(?<=bought [0-9]*.[0-9]* )\w+/gi;
    valueGiven = true;
}

if(intent.action.match("(sell)") && !intent.utterance.match("(at|@)") && !intent.utterance.match("(for)")) {
    regex = /(?<=sold [0-9]*.[0-9]* )\w+/gi;
    valueGiven = true;
}

if(intent.action === "transfer") {
    regex = /(transfer |transferred |send |sent |received )\$?\d*\.?\d\s+[A-Z]*/gi;
}
```

```
let asset = match[0];
if(intent.action === "transfer") {
    asset = match[0].split(" ").pop();

    if(intent.utterance.includes("from")) {
        let from = intent.utterance.match(/(from )+[A-Z]*/gi)[0].split(" ")[1];
        details["from"] = capitalizeFirstLetter(from);
        details["to"] = "Me";
    } else if(intent.utterance.includes("to")) {
        let to = intent.utterance.match(/(to )+[A-Z]*/gi)[0].split(" ")[1];
        details["from"] = "Me";
        details["to"] = capitalizeFirstLetter(to);
    }
}

details["amount"] = parseFloat(entities[0].resolution.value);
details["asset"] = asset;
```

Once the amount and symbol of the asset have been extracted, the date is converted into a format consistent with other trading activity records, and the request is processed, which entails interacting with the GraphQL API to save the data to the database.

```
if(entities[1]?.typeName.includes("number")) {
    if(intent.action !== "transfer") {
        if(valueGiven) {
            details["price"] = parseFloat(entities[1].resolution.value) / details.amount;
        } else {
            details["price"] = parseFloat(entities[1].resolution.value);
        }
    }

    if(!empty(lastEntity) && lastEntity?.typeName.includes("date")) {
        details["date"] = lastEntity.resolution.values[0].value;
    }
} else if(entities[1]?.typeName.includes("date")) {
    details["date"] = entities[1].resolution.values[0].value;
} else if(lastEntity?.typeName.includes("date")) {
    details["date"] = lastEntity.resolution.values[0].value;
}

if(!("date" in details)) {
    details["date"] = new Date().toISOString().split("T")[0]
}

processRequest(details);
```

Appendix 3: Automated Test Results

A total of 35 unit and integration tests were written in order to automate the API testing.

```
PASS  tests/api.test.ts (6.789 s)
  API Testing
    Server Tests
      ✓ Ensure the API server is running. (20 ms)
      ✓ Ensure the bot server is running. (18 ms)
    Database Tests
      ✓ Ensure the DB exists. (1 ms)
      ✓ Ensure all tables exist. (1 ms)
    GraphQL API Tests
      User
        ✓ Create user. (172 ms)
        ✓ User exists. (11 ms)
        ✓ Login. (125 ms)
        ✓ Recreate user. (247 ms)
      Watchlist
        ✓ Create watchlist. (9 ms)
        ✓ Read watchlist. (9 ms)
        ✓ Delete watchlist. (16 ms)
      Messages
        ✓ Create message. (8 ms)
        ✓ Read message. (8 ms)
        ✓ Delete message. (21 ms)
      Transactions
        ✓ Create transaction. (9 ms)
        ✓ Read transaction. (11 ms)
        ✓ Update transaction. (14 ms)
        ✓ Delete transaction. (15 ms)
```

```
      Budget
        ✓ Update budget. (15 ms)
        ✓ Read budget. (7 ms)
      Settings
        ✓ Update settings. (14 ms)
        ✓ Read settings. (7 ms)
      Holdings
        ✓ Create holding. (7 ms)
        ✓ Read holdings. (8 ms)
        ✓ Update holding. (13 ms)
        ✓ Delete holding. (14 ms)
      Activities
        ✓ Create activity. (8 ms)
        ✓ Read activities. (8 ms)
        ✓ Update activity. (15 ms)
        ✓ Delete activity. (15 ms)
      Stock
        ✓ Get stock price. (1403 ms)
      Crypto
        ✓ Get global. (206 ms)
    RESTful API Tests
      ✓ Change password. (214 ms)
      ✓ Verify token. (6 ms)
      ✓ Logout. (6 ms)

Test Suites: 1 passed, 1 total
Tests:       35 passed, 35 total
Snapshots:   0 total
Time:        6.849 s
```

The tests above were all run every time a commit was made to the main branch, which ensured that the code was always working as intended, and that any issues could be identified automatically without manually having to test every endpoint and resolver.

```
describe("GraphQL API Tests", () => {
    describe("User", () => {
        test("Create user.", async () => {
            let create: any = await requests.createAccount("Testing", "test", "testkey");
            expect(create.data.createUser).toEqual("Done");
        });

        test("User exists.", async () => {
            let exists: any = await requests.userExists("Testing");
            expect(exists.data.userExists).toEqual("Testing");
        });

        test("Login.", async () => {
            let login: any = await requests.login("Testing", "test");
            user = login;
            expect(user.userID).toEqual(1);
        });

        test("Recreate user.", async () => {
            let deleteUser: any = await requests.deleteUser(user.token, user.userID);
            let userExists: any = await requests.userExists("Testing");
            let createUser: any = await requests.createAccount("AnotherTest", "test", "testKey");
            let login: any = await requests.login("AnotherTest", "test");
            user = login;
            expect(deleteUser.data.deleteUser).toEqual("Done");
            expect(userExists.data.userExists).not.toEqual("Testing");
            expect(createUser.data.createUser).toEqual("Done");
            expect(user.userID).toEqual(2);
        });
    });
```

The image above shows an example of 4 Jest test functions written to test the user-related resolvers of the GraphQL API. The tests would begin by creating a user, then checking if the created user was actually successfully created. The next test would be to log the user in, and ensure that the correct userID was returned by the API. Finally, the user would be deleted and recreated in order to test the delete functionality; the user would then be logged in again so that the next test functions can use a valid token to populate the database with other data, and test the other endpoints and resolvers.

```
beforeAll(async () => {
    api = new API();
    await api.start();
});

afterAll(async () => {
    user = {};
    store = {};

    await api?.kill();
    socket.close();
});
```

The “beforeAll" and “afterAll” functions of the Jest testing framework were used as the setup and teardown functions that most testing frameworks provide. In this case, they are used to start the server before the tests begin, and to stop it at the end once the tests are done. Failing to do so would result in Continuous Integration workflows to fail, as they would keep running endlessly.

Appendix 4: Usability Test Results

Session 1 - End of Sprint 4

Tasks

1. Register an account.
2. Log in with the registered account.
3. Interact with the settings page and change some settings.

Summary of Results

- The UI looks good.
- Registration process is very smooth.
- Colors start to get too repetitive.
- Toggles are too big.
- Settings being separated into categories is very nice.
- Borders can be too thin to separate content.

Bugs

- User settings not fetched correctly

Session 2 - End of Sprint 6

Tasks

1. Register an account.
2. Log in with the registered account.
3. Check what the price of Bitcoin was a week ago.
4. Record a trading activity.
5. Check the performance of your holdings.

Summary of Results

- Very clean UI.
- Separating pages by color is a cool detail.
- Input fields should have labels.
- Date format should be changeable.
- Date picker should be added to the mobile app.
- Activities can be confusing.

Bugs

- Missing holdings.

Session 3 - End of Sprint 8

Tasks

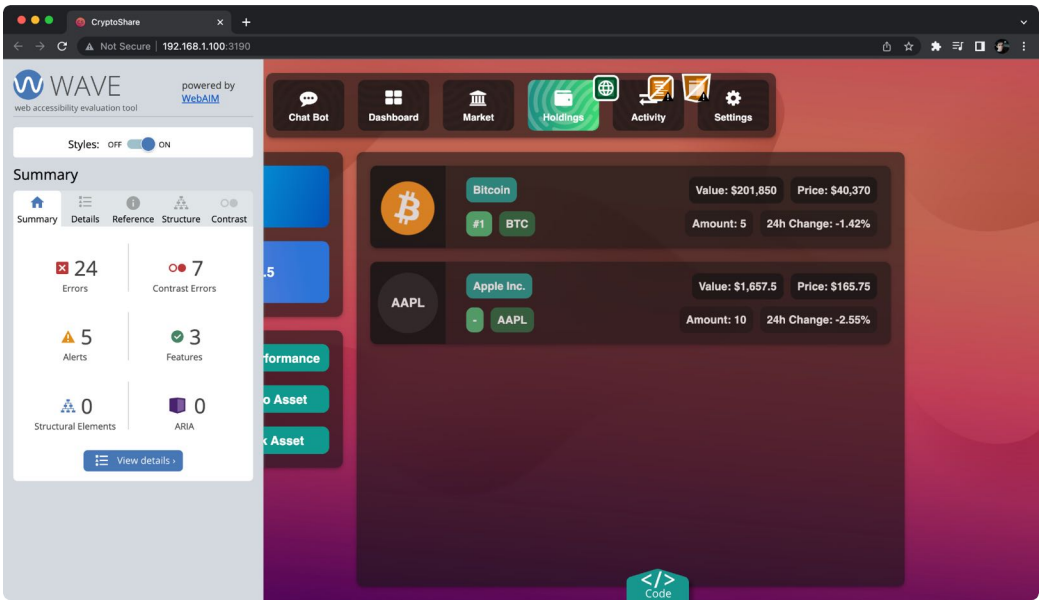
1. Register an account.
2. Log in with the registered account.
3. Use the chat bot to record a trading activity where you bought 5 BTC on the 1st of January, 2021.
4. Set your budgeting preferences.
5. Record a transaction where you bought a bus ticket for $5 yesterday.
6. Edit the transaction from $5 to $10.
7. Add Ethereum to your watchlist.
8. Check what the market cap of LRC (Loopring) is.
9. Record an activity where you bought 2 ETH on the 25th of October, 2021.
10. Check the performance of your holdings.

Summary of Results

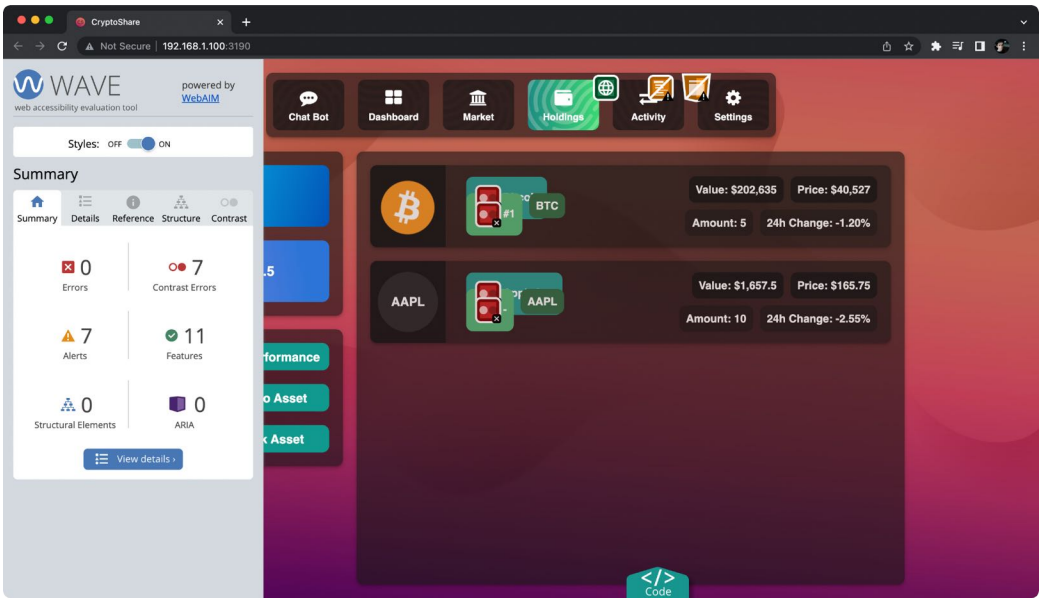
- Chat bot was confusing at first, but the “Help” button was useful.
- Setting my budget was easy.
- Recording a transaction was easy.
- Editing the transaction was also easy.
- Adding an asset to my watchlist was a bit confusing.
- Checking the market cap of LRC was simple.
- Popup size is not right when recording an activity.
- Date picker should show today’s date.
- Date picker arrows should be more visible.
- Holdings performance was easy to check.

Appendix 5: Production Build Test Results

Before testing the logic of the software, a WAI Evaluation Tool was used to check whether or not the application complied with the Web Accessibility Initiative guidelines.



When first tested, the application had 31 errors, 7 of which were contrast errors. The major errors were due to input fields not having a corresponding label tag, and buttons with icons being seen as “empty” by potential screen readers. These issues were fixed:



Contrast errors could be solved by simply changing the application’s theme.

The next step in testing the production build of the software involved testing all its features, as well as trying to input invalid data to test for edge cases and corner cases. The table below shows a full list of every test that was ran, any data that was entered, and the results of each test:

Test	Input Data	Result
Registration with invalid details	Space character as username and password	Invalid data error shown as expected
Registration with existing username	Taken username	User already exists error
Sending invalid message to chat bot	Numbers, symbols, random words etc.	Either no reply from the bot, or telling the user it does not know what they mean
Sending requests to the bot worded in an invalid way	“I bought some BTC yesterday.”	Error shown as expected since the bot cannot turn “some” into a number
Checking item affordability using bot	“Can I afford a $20 pizza?”	Worked as intended
Setting income with bot	“Set my income to $50000.”	Worked as intended
Setting holdings with bot	“Set my BTC holdings to 5.”	Worked as intended
Recording a transaction with the bot	“I bought a train ticket for $50 yesterday.”	Worked as intended
Recording a trading activity with the bot	“I bought 2 BTC today.”	Worked as intended
Editing watchlist with the bot	“Remove ETH from my watchlist.”	Worked as intended
Creating a transaction with invalid data	0, -1, alphanumeric strings, and symbols entered as amount and date	Error shown as expected informing the user that the data is invalid
Creating a transaction with valid data	Amount: 50, Type: Spent, Category: Food, Date: 2022-03-01, Notes: Restaurant	Worked as intended

Trying to enter invalid budget data	Budget data adding up to less than or greater than 100, and using words instead of numbers	Error shown as expected as the percentages need to add up to 100, and must be numbers
Changing budget data with valid values	10, 10, 10, 10, 10, 10, 40	Worked as intended
Trying to enter an invalid yearly income	-1	Error shown as expected
Changing income to a valid number	30000	Worked as intended
Changing budgeting date range	February 2022	Worked as intended
Adding non-existent asset to watchlist	TESTSYMBOL	Error shown as expected
Adding asset to watchlist	ETH	Worked as intended
Removing asset from watchlist	-	Worked as intended
Viewing global market data	-	Worked as intended
Searching for non-existent asset	TESTSYMBOL	Error shown as expected
Searching for an asset	BTC	Worked as intended
Viewing cryptocurrency market data	-	Worked as intended
Viewing stock market data	-	Worked as intended
Using market charts	-	Worked as intended
Creating a cryptocurrency holding	Symbol: ETH, Amount: 2	Worked as intended
Viewing holdings	-	Worked as intended
Creating a stock holding	Symbol: AAPL, Amount: 10	Worked as intended
Updating holding	Symbol: ETH, Amount: 5	Worked as intended
Deleting holding	-	Worked as intended

Creating an activity	Symbol: ETH, Asset Type: Crypto, Amount: 5, Date: 2022-03-01, Fee: 0, Notes: Investment, Activity Type: Buy, Exchange: Kraken, Pair: ETH/USD, Price: 3000	Worked as intended
Viewing activities	-	Worked as intended
Updating an activity	Symbol: ETH, Amount: 10	Worked as intended
Searching for an activity	“ETH”	Worked as intended
Deleting an activity	-	Worked as intended
Using tools and calculators	-	Worked as intended
Modifying settings	-	Worked as intended
Changing password	“NewPass”	Worked as intended
Logging out	-	Worked as intended
Logging out everywhere	-	Worked as intended
Settings sync	-	Worked as intended
Generating QR code	-	Worked as intended
Logging in using QR code	-	Worked as intended
Admin actions	-	Worked as intended
Resetting data	-	Worked as intended
Exporting data	-	Worked as intended
Importing data	CSV and JSON files	Worked as intended
Managing data	-	Worked as intended
Changing currency	-	Worked as intended
Setting stock API key	Stock API key	Worked as intended
Deleting account	-	Worked as intended

Appendix 6: API Documentation

CryptoShare's API is split into 3 main parts; the first part is the regular Node.js API routes set up with Express, the second is the Socket.IO event handler, and the third is the GraphQL endpoint that has 41 resolvers. The documentation below will be separated into the aforementioned 3 parts.

Express Routes

Method	Route	Input	Response	Description
GET	/status	-	{ status:"online" }	Used to check if the API is working.
GET	/keyRSA	-	Public RSA key	Used to get the public RSA key of the server.
POST	/login	username, password	Object with userID, username, key, token, and settings or error	Given a username and RSA encrypted password, logs the user in.
POST	/logout	userID, token	"Done" or error	Given a userID and token, logs the user out.
POST	/logoutEverywhere	userID, token	"Done" or error	Given a userID and token, logs the user out of all their active sessions.
POST	/changePassword	userID, token, key, currentPassword, newPassword	Object with userID, username, key, and token or error	Given a userID, token, key, and RSA encrypted passwords, changes the user's password.
POST	/verifyToken	userID, token	Object with userID, username, key, token, and settings or error	Given a userID and token, checks if the login/session token is valid.
POST	/adminAction	userID, username, token, action	Object with admin settings	Performs an admin-only action.

Socket.IO Event

Event Name	Input	Response	Description
message	Object containing the properties “message” and “privateKey”	An object containing the NLP processed result of the user’s message	Used to process the user’s message using Natural Language Processing.

GraphQL Resolvers

In GraphQL, all requests are made to a single endpoint, but the content of the request allows the API to know what the client intends to do. Resolvers are functions that can take any number of inputs, and provide an output that the API relays as a response to the request. Therefore, resolvers can be thought of as the traditional endpoints that a RESTful API might have. When setting up a GraphQL API, a schema needs to be created, where the resolvers are defined along with their inputs and the data types. A “Query” is an operation where data is only read, whereas a “Mutation” involves creating, updating or deleting data. In this sense, GraphQL is already documented as the schema is easy to read, but tables have been created below to make it easier as the schema is written in code. The resolvers have been separated according to the database table they interact with. Please note that, as per GraphQL’s documentation, an exclamation mark next to a data type means that a particular input cannot be NULL.

Type	Resolver	Input	Response	Description
Query	userExists	username: String!	Error or username	Used to check if a username is taken.
Query	readUser	token: String!, userID: Int!	A “User” object	Used to get the details of a user.
Mutation	createUser	username: String!, password: String!, key: String!	Error or success message	Used to create a user.
Mutation	updateUser	token: String!, userID: Int!, password: String!, key: String!	Error or success message	Used to update a user’s details.
Mutation	deleteUser	token: String!, userID: Int!	Error or success message	Used to delete a user’s account.

Type	Resolver	Input	Response	Description
Query	readActivity	token: String!, userID: Int!	An array of “Activity” objects	Used to get the trading activities of a user.
Mutation	createActivity	token: String!, userID: Int!, activityAssetID: String!, activityAssetSymbol: String!, activityAssetType: String!, activityDate: String!, activityType: String!, activityAssetAmount: String!, activityFee: String!, activityNotes: String!, activityExchange: String!, activityPair: String!, activityPrice: String!, activityFrom: String!, activityTo: String!	Error or success message	Used to create/record a trading activity.
Mutation	updateActivity	token: String!, userID: Int!, activityTransactionID: String!, activityAssetID: String!, activityAssetSymbol: String!, activityAssetType: String!, activityDate: String!, activityType: String!, activityAssetAmount: String!, activityFee: String!, activityNotes: String!, activityExchange: String!, activityPair: String!, activityPrice: String!, activityFrom: String!, activityTo: String!	Error or success message	Used to update an activity’s details.
Mutation	deleteActivity	token: String!, userID: Int!, activityID: Int!	Error or success message	Used to delete an activity.
Mutation	deleteActivityAll	token: String!, userID: Int!	Error or success message	Used to delete all activity data of a user.

Type	Resolver	Input	Response	Description
Query	readMessage	token: String!, userID: Int!	An array of "Message" objects	Used to get the chat bot messages of a user
Mutation	createMessage	token: String!, userID: Int!, message: String!	Error or success message	Used to create a message.
Mutation	updateMessage	token: String!, userID: Int!, messageID: Int!, message: String!	Error or success message	Used to update a message's details.
Mutation	deleteMessage	token: String!, userID: Int!, messageID: Int!	Error or success message	Used to delete a message.
Mutation	deleteMessageAll	token: String!, userID: Int!	Error or success message	Used to delete the message data of a user.

Type	Resolver	Input	Response	Description
Query	readTransaction	token: String!, userID: Int!	An array of "Transaction" objects	Used to get the transaction data of a particular user.
Mutation	createTransaction	token: String!, userID: Int!, transactionType: String!, transactionDate: String!, transactionCategory: String!, transactionAmount: String!, transactionNotes: String!	Error or success message	Used to create a transaction.
Mutation	updateTransaction	token: String!, userID: Int!, transactionID: String!, transactionType: String!, transactionDate: String!, transactionCategory: String!, transactionAmount: String!, transactionNotes: String!	Error or success message	Used to update a transaction's details.

Mutation	deleteTransaction	token: String!, userID: Int!, transactionID: String!	Error or success message	Used to delete a transaction.
Mutation	deleteTransactionAll	token: String!, userID: Int!	Error or success message	Used to delete all the transaction data of a user.

Type	Resolver	Input	Response	Description
Query	readHolding	token: String!, userID: Int!	An array of “Holding” objects	Used to get the holding data of a user.
Mutation	createHolding	token: String!, userID: Int!, holdingAssetID: String!, holdingAssetSymbol: String!, holdingAssetAmount: String!, holdingAssetType: String!	Error or success message	Used to create a holding.
Mutation	updateHolding	token: String!, userID: Int!, holdingID: Int!, holdingAssetID: String!, holdingAssetSymbol: String!, holdingAssetAmount: String!, holdingAssetType: String!	Error or success message	Used to update a holding’s details.
Mutation	deleteHolding	token: String!, userID: Int!, holdingID: Int!	Error or success message	Used to delete a holding.
Mutation	deleteHoldingAll	token: String!, userID: Int!	Error or success message	Used to delete all the holding data of a user.

Type	Resolver	Input	Response	Description
Query	readSetting	token: String!, userID: Int!	A “Setting” object	Used to get the settings of a user.
Mutation	createSetting	token: String!, userID: Int!, userSettings: String!	Error or success message	Used to create a settings row.

Mutation	updateSetting	token: String!, userID: Int!, userSettings: String!	Error or success message	Used to update user settings.
Mutation	deleteSetting	token: String!, userID: Int!	Error or success message	Used to delete user settings.

Type	Resolver	Input	Response	Description
Query	readBudget	token: String!, userID: Int!	A “Budget” object	Used to get the budget data of a user.
Mutation	createBudget	token: String!, userID: Int!, budgetData: String!	Error or success message	Used to create a budget row.
Mutation	updateBudget	token: String!, userID: Int!, budgetData: String!	Error or success message	Used to update budget data.
Mutation	deleteBudget	token: String!, userID: Int!	Error or success message	Used to delete budget data.

Type	Resolver	Input	Response	Description
Query	readWatchlist	token: String!, userID: Int!	An array of “Watchlist” objects	Used to get the watchlist data of a user.
Mutation	createWatchlist	token: String!, userID: Int!, assetID: String!, assetSymbol: String!, assetType: String!	Error or success message	Used to create a watchlist row to add an item to a user’s watchlist.
Mutation	updateWatchlist	token: String!, userID: Int!, watchlistID: Int!, assetID: String!, assetSymbol: String!, assetType: String!	Error or success message	Used to update watchlist row data.
Mutation	deleteWatchlist	token: String!, userID: Int!, watchlistID: Int!	Error or success message	Used to delete a watchlist row.
Mutation	deleteWatchlistAll	token: String!, userID: Int!	Error or success message	Used to delete all watchlist data of a user.

Type	Resolver	Input	Response	Description
Query	readCoin	token: String!, userID: Int!, assetID: String!, assetSymbol: String!, currency: String!	A “Coin” object	Used to get the cached data of a coin.

Type	Resolver	Input	Response	Description
Query	readStockHistorical	token: String!, userID: Int!, keyAPI: String!, assetSymbol: String!	A “Stock” object	Used to get the cached historical data of a stock.
Query	readStockPrice	token: String!, userID: Int!, keyAPI: String!, symbols: [String]!	An array of “Stock” objects	Used to get the cached price data of one or more stocks.

Appendix 7: Biweekly Stand-Up Meetings

The supervisor of the project held biweekly stand-up meetings to check its progress. To prepare for the meetings, PowerPoint presentations were created. They can be seen by following the link below:

https://github.com/Xtrendence/CryptoShare/tree/main/docs/Meetings

After the first few meetings, the actual work carried out during the previous or current sprint was shown through screen sharing.

Appendix 8: User Guide

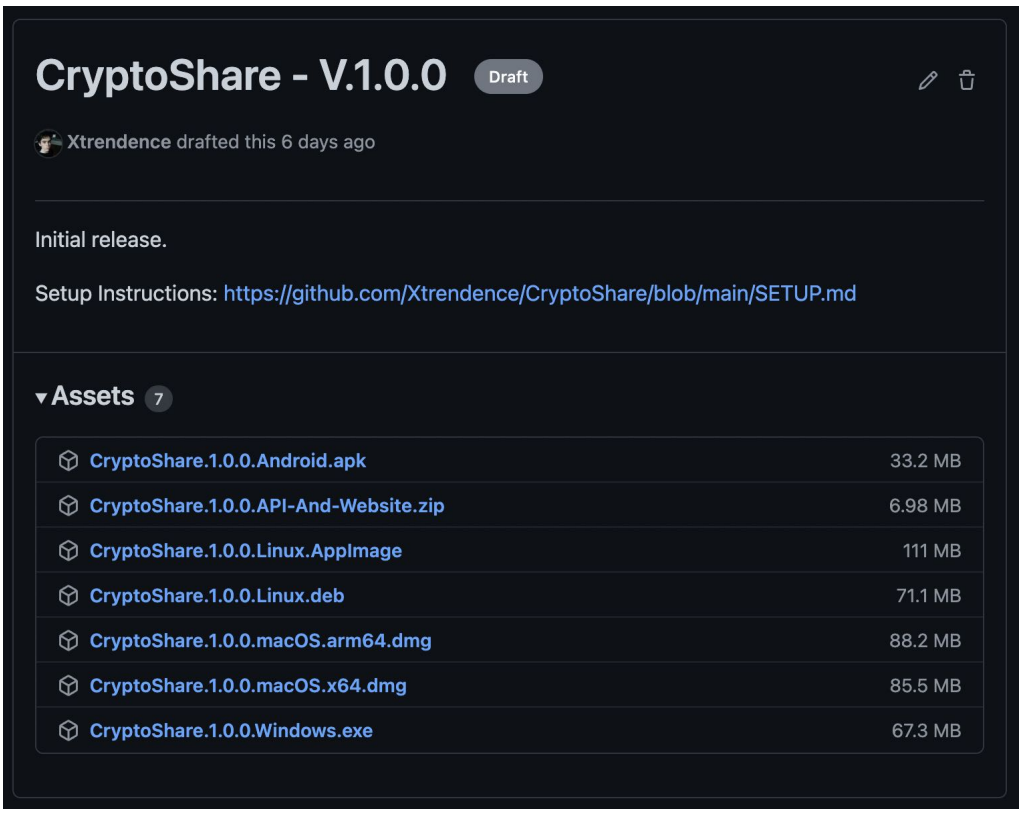
Considering the application is meant for investors, a user guide has been created to make it clearer for readers who are not investors themselves what the application can do, and how a user would utilize it. The rest of this document is purely screenshots and explanations on how to interact with the application, including the setup process. Please note that setup instructions have been provided on the repository as well for regular users who would not read this report:

https://github.com/Xtrendence/CryptoShare/blob/main/SETUP.md

Setting Up the API

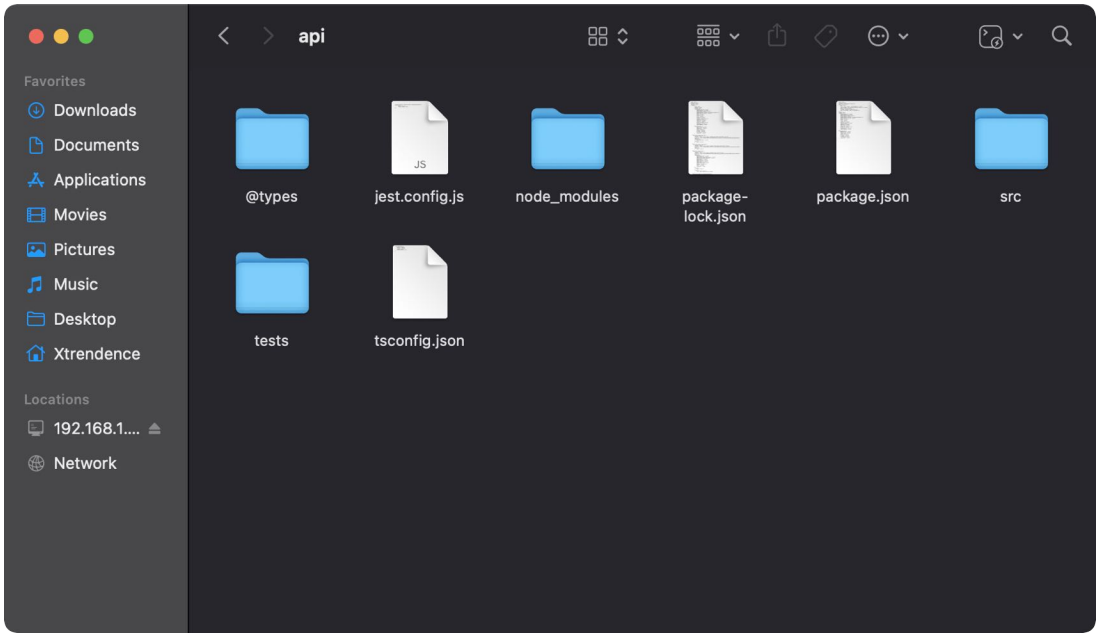
In order to set up the API, users must download a ZIP archive containing both the API and the web app from the Releases section on the GitHub repository:

https://github.com/Xtrendence/CryptoShare/releases

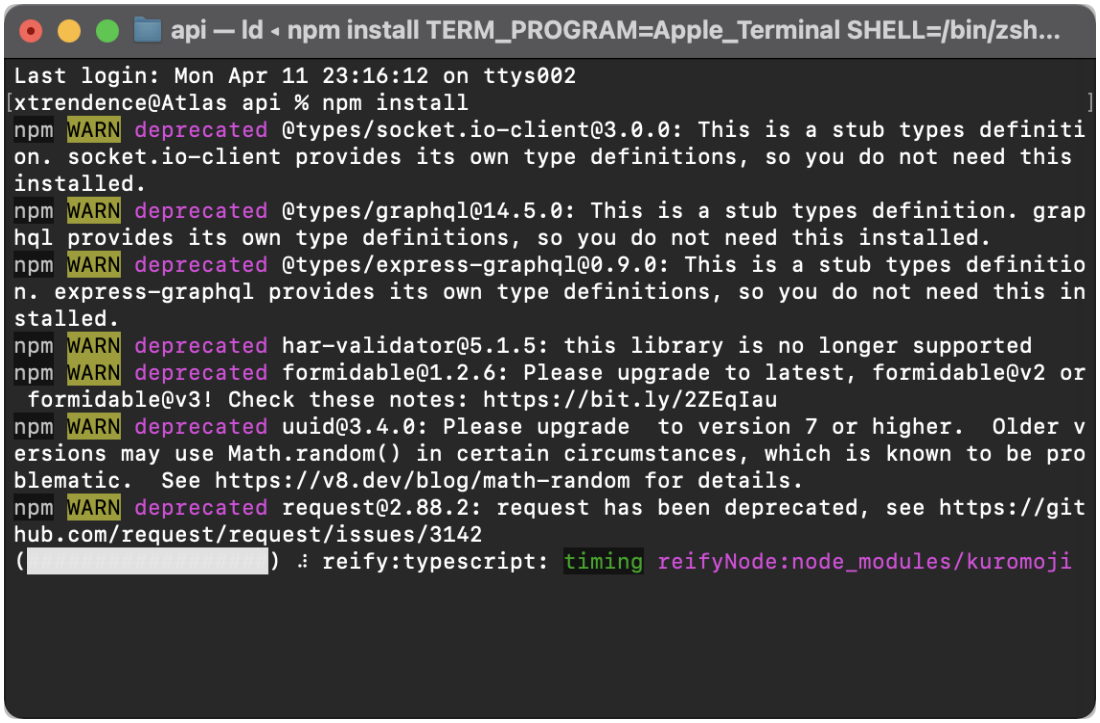


There are a number of choices for each platform. iOS is not included, as Apple charges a $100/year developer fee to be able to codesign applications.

Once downloaded, the contents of the archive can be extracted. The user must navigate to the “api” folder:



They must then use a CLI of their choice, such as Terminal, Git Bash, Command Prompt, or PowerShell in order to open the directory in the CLI. Once opened, they can run “npm install” to install the required dependencies for the API:

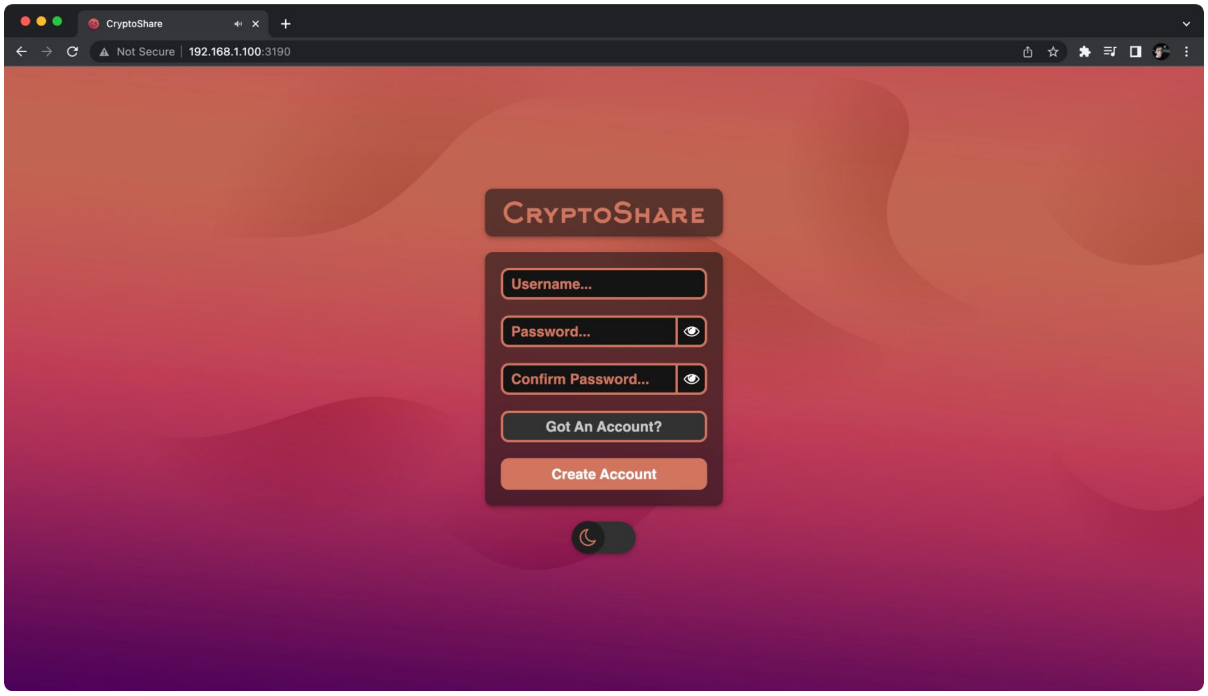


Once NPM has finished installing the Node.js modules, the server can be started using the “npm start” command:

```
api — node ◂ npm start TERM_PROGRAM=Apple_Terminal SHELL=/bin/zs...

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
xtrendence@Atlas api % npm start

> api@1.0.0 start
> npm run ts-node


> api@1.0.0 ts-node
> ts-node --files src/index.ts

 ----------------------------
 Starting Server... (23:17:11)
 Started Server (23:17:11)
  Web Interface:  http://192.168.1.100:3190
  GraphQL API:  http://192.168.1.100:3190/graphql
  Bot Server:  http://192.168.1.100:3190
```

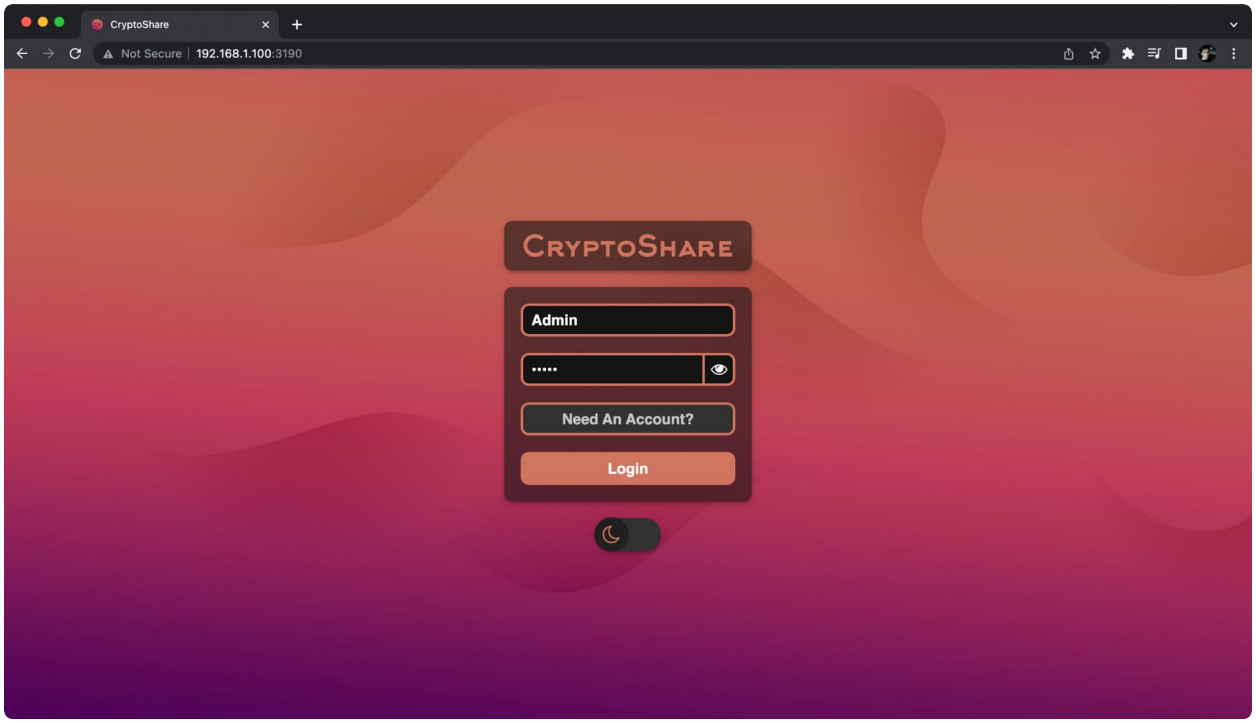
Web App

Navigating to the “Web Interface” URL on any device on the same network would allow the user to access the web app (please note that the desktop app is not included in this guide, as it is completely identical to the web app in terms of UI):

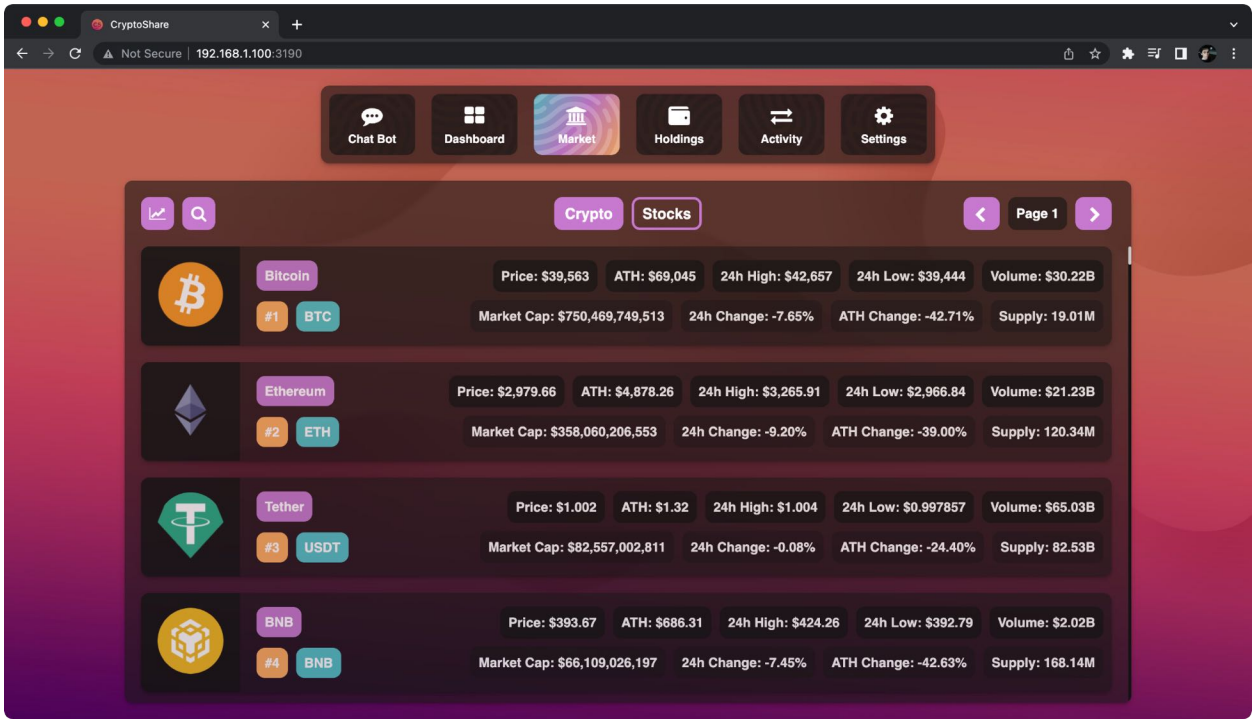


They are then able to create an account.

Once they have created an account, they can use the login form to log in:

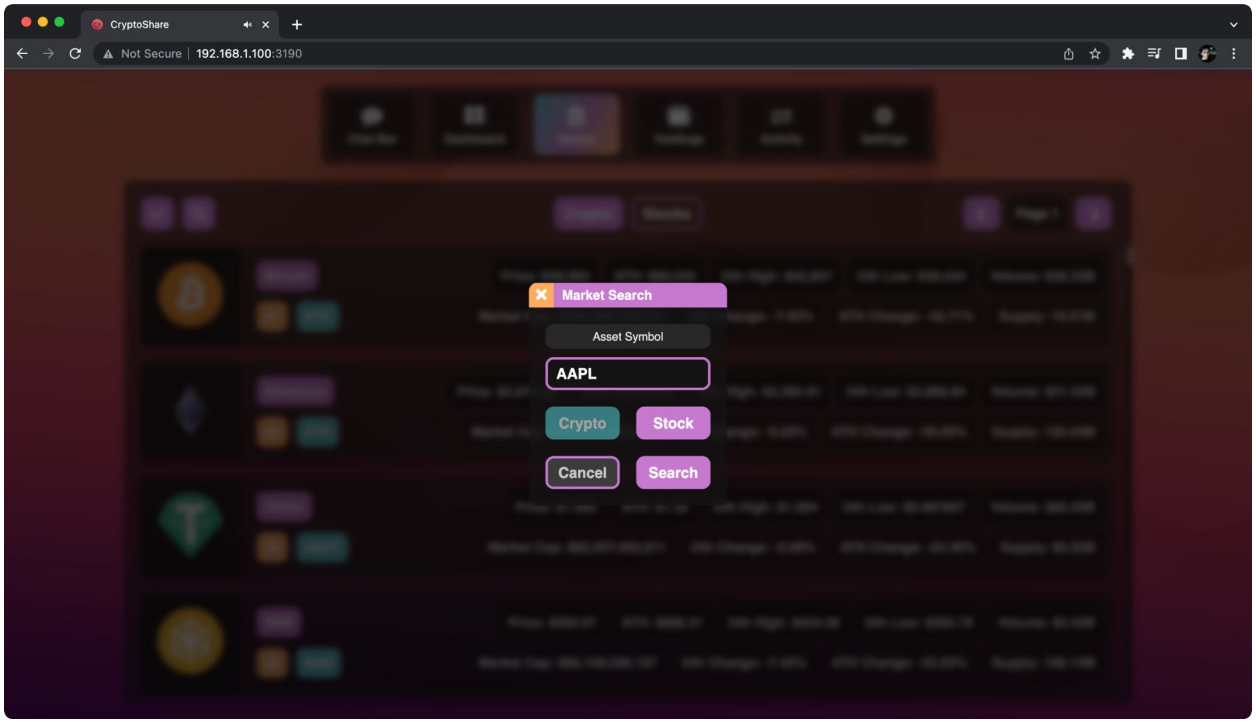


Upon logging in, they will be presented with the default page, which is the “Market” page, which they can use to view cryptocurrency and stock market data:



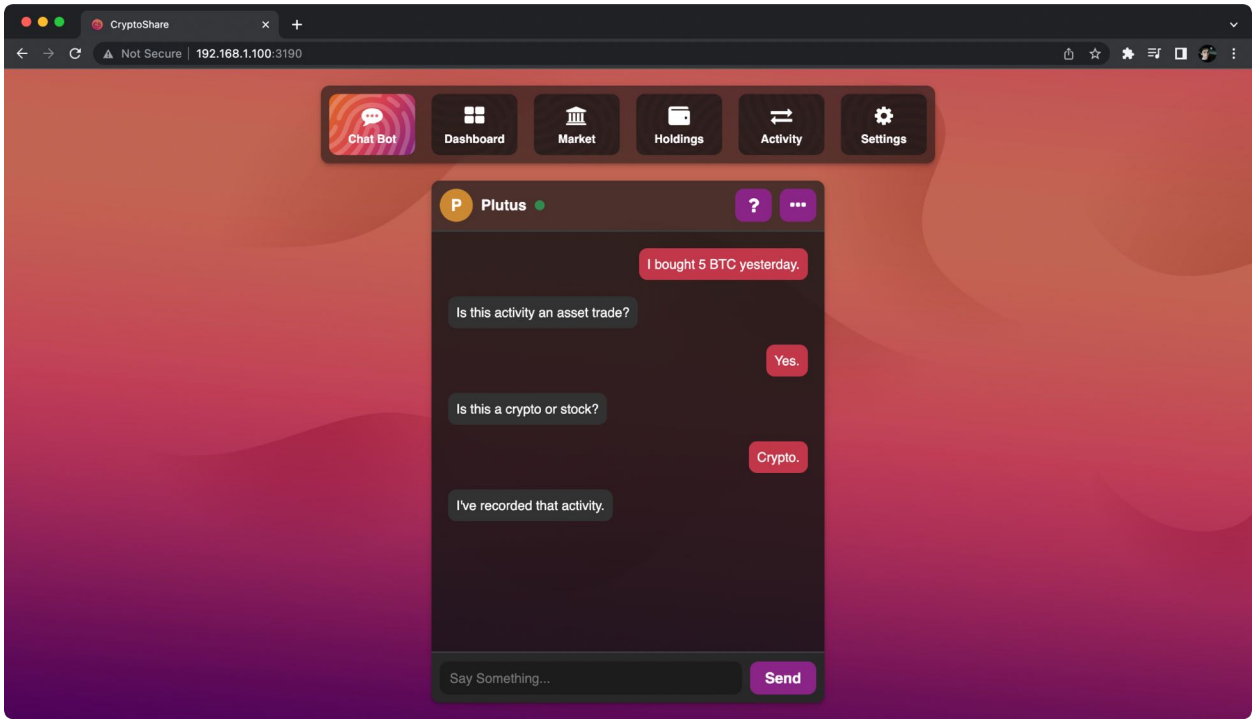
Clicking on a particular cryptocurrency or stock would show its price chart and market data:

Optionally, users can also search for specific coins or stocks:

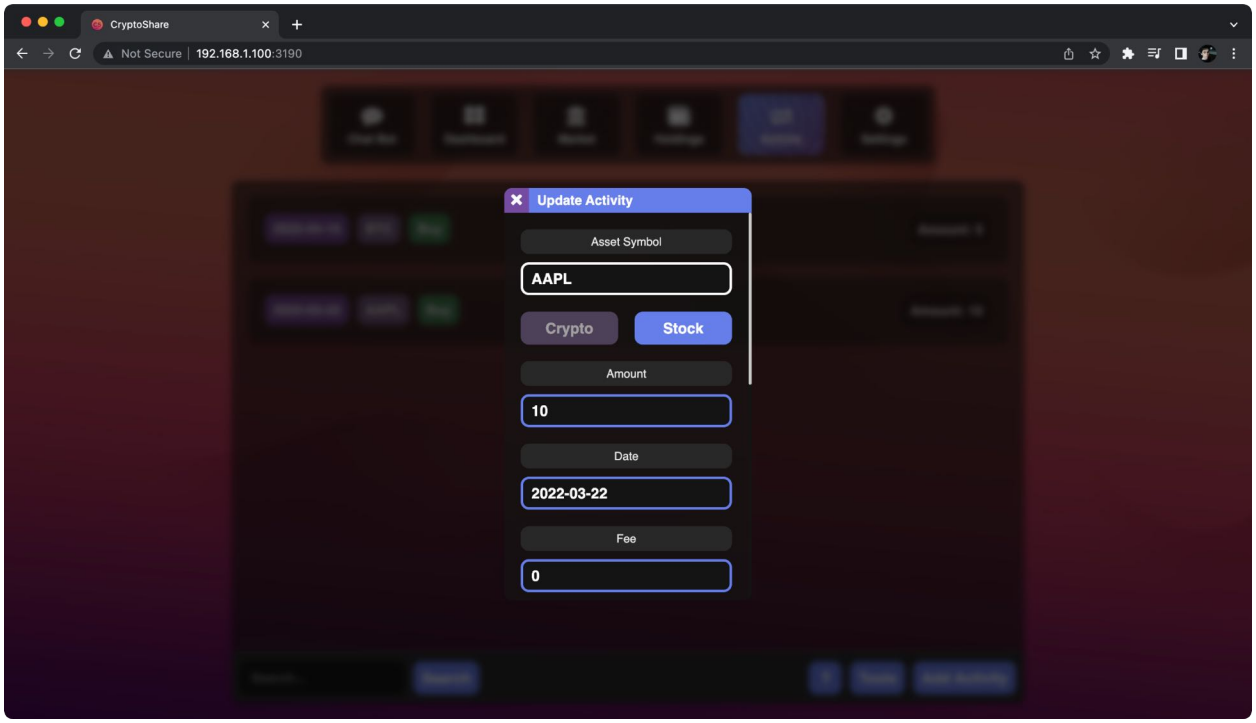


In this case, Apple's stock market data has been searched for and is being shown in the image below:

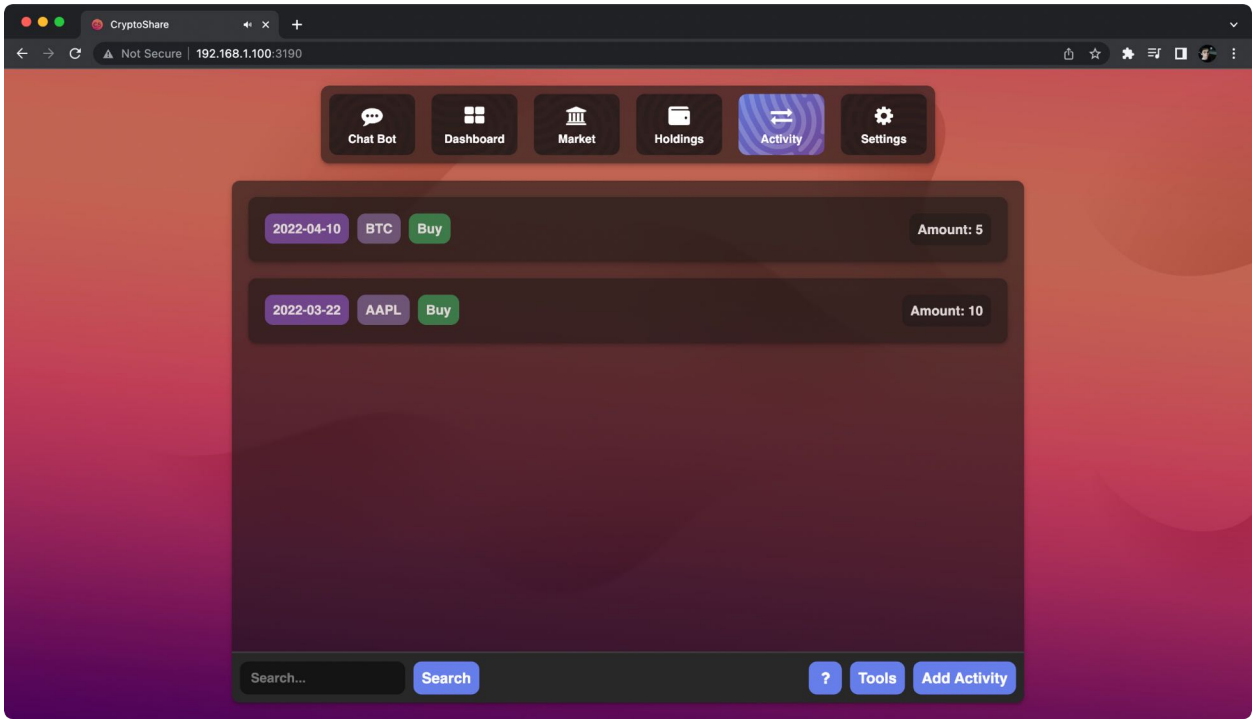
One of the functionalities the chat bot offers is the ability to record trading activity:



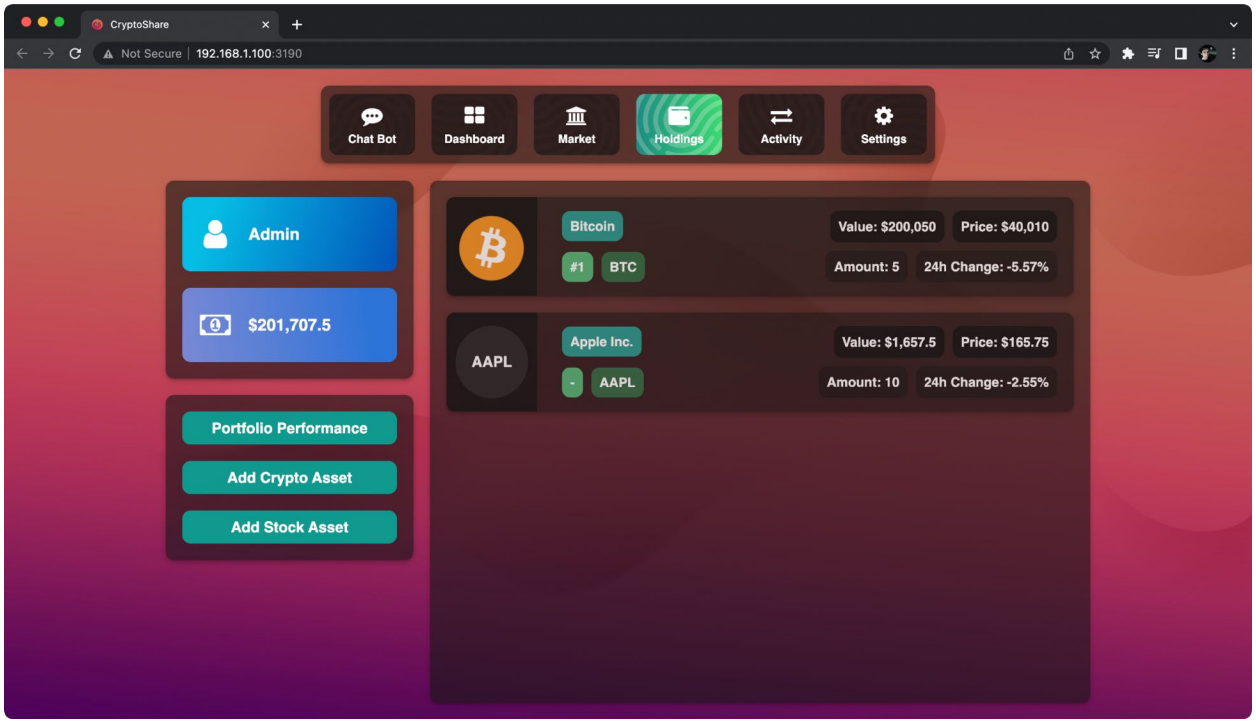
Users can also record trading activities through the “Activity” page’s create/update popup:



Once a trading activity is recorded, it shows up on the “Activity” page:

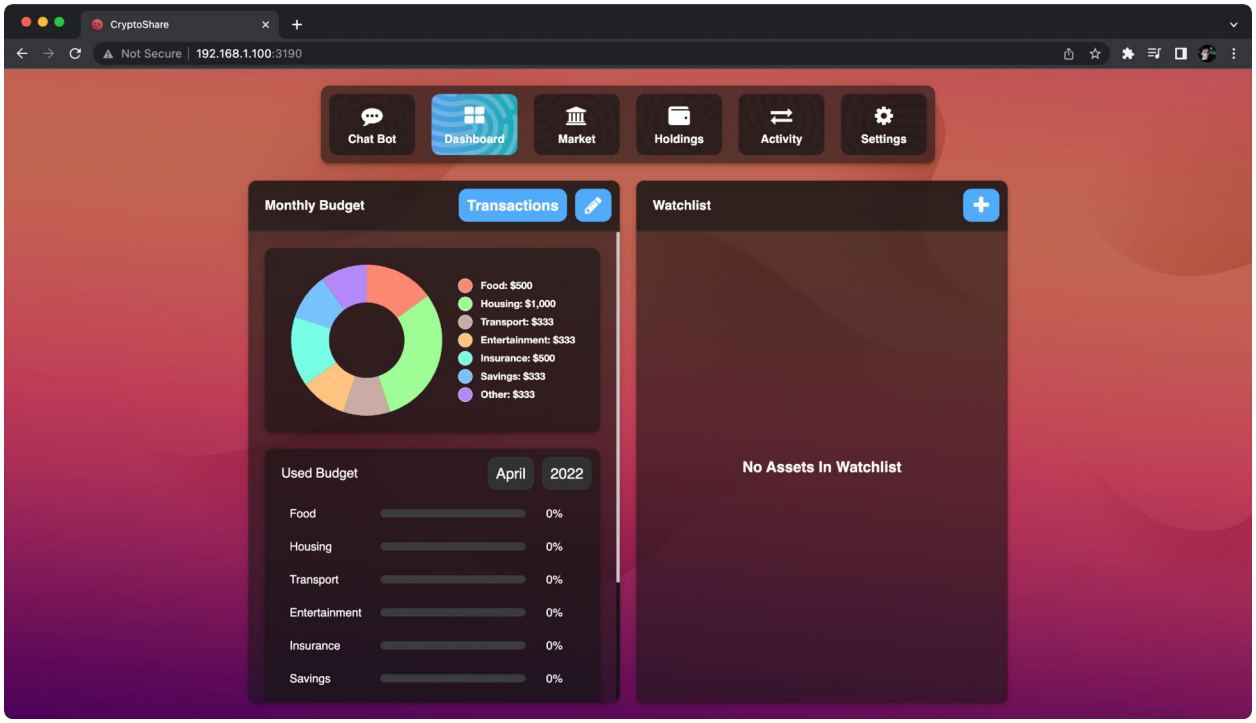


Optionally, users can enable a setting that calculates their holdings based on their trading activity:

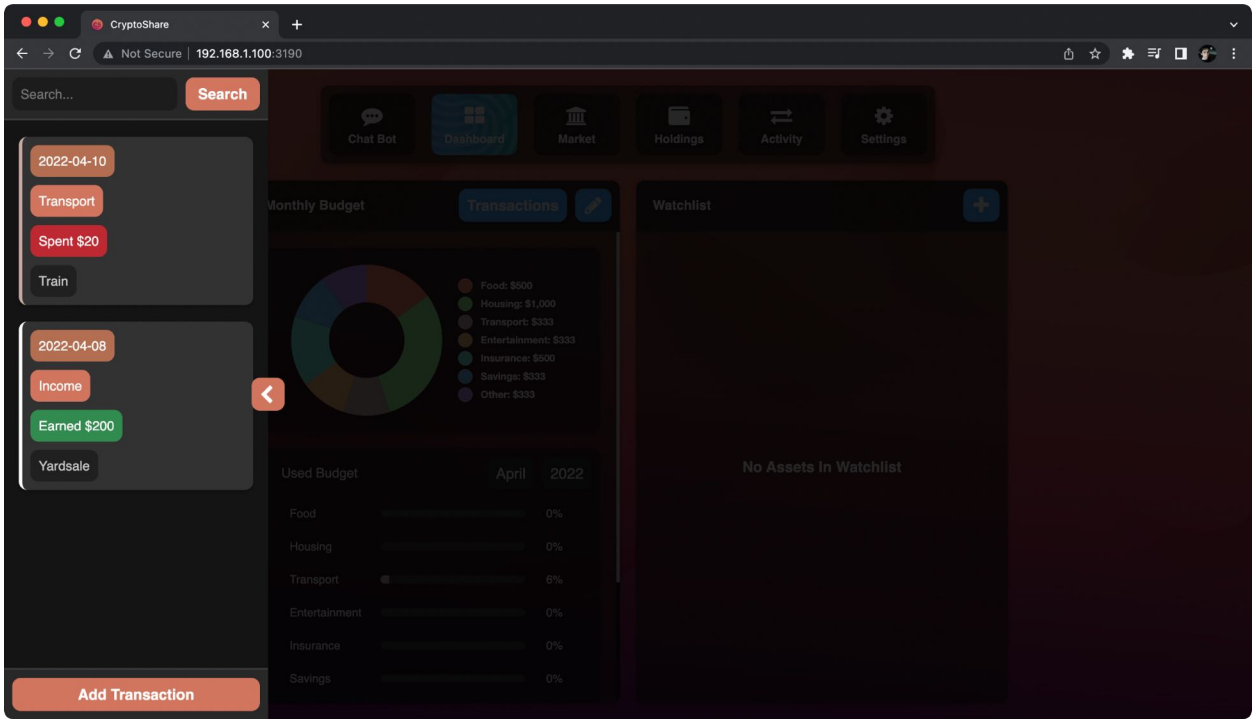


Clicking on a holding would show its performance:

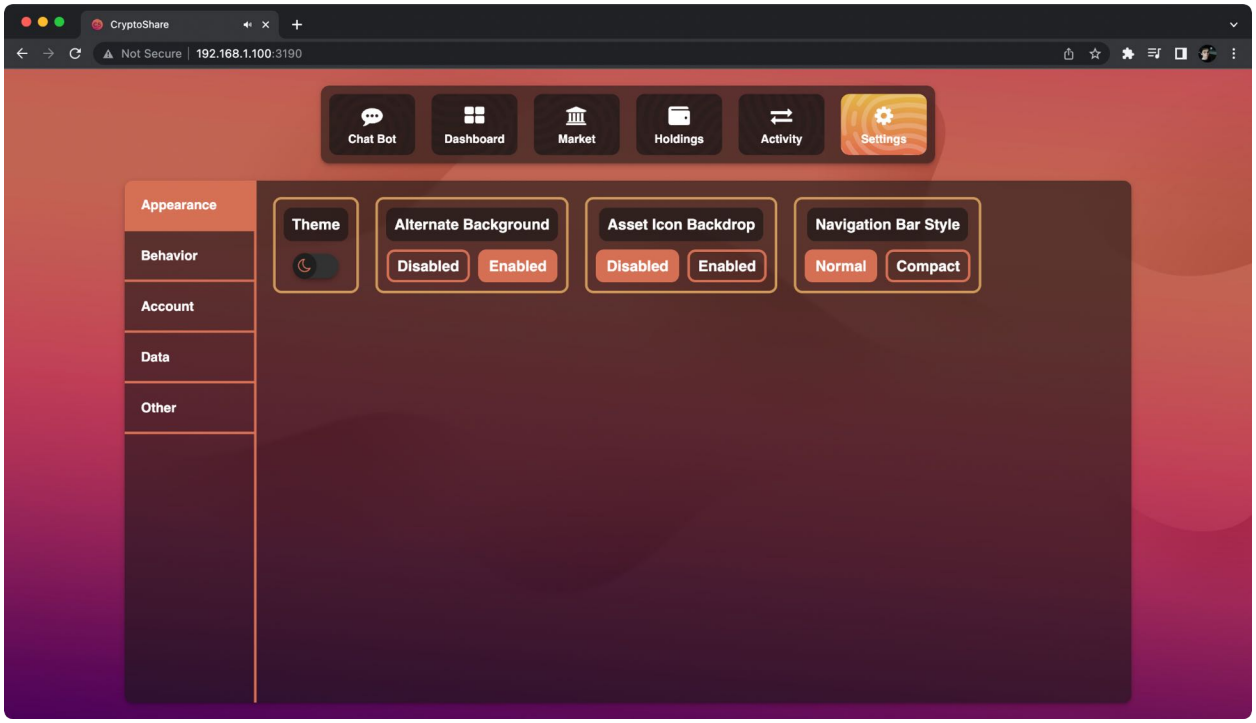
Users can set their budget, record transactions, and manage their finances, or keep an eye on specific coins or stocks through the “Dashboard” page:



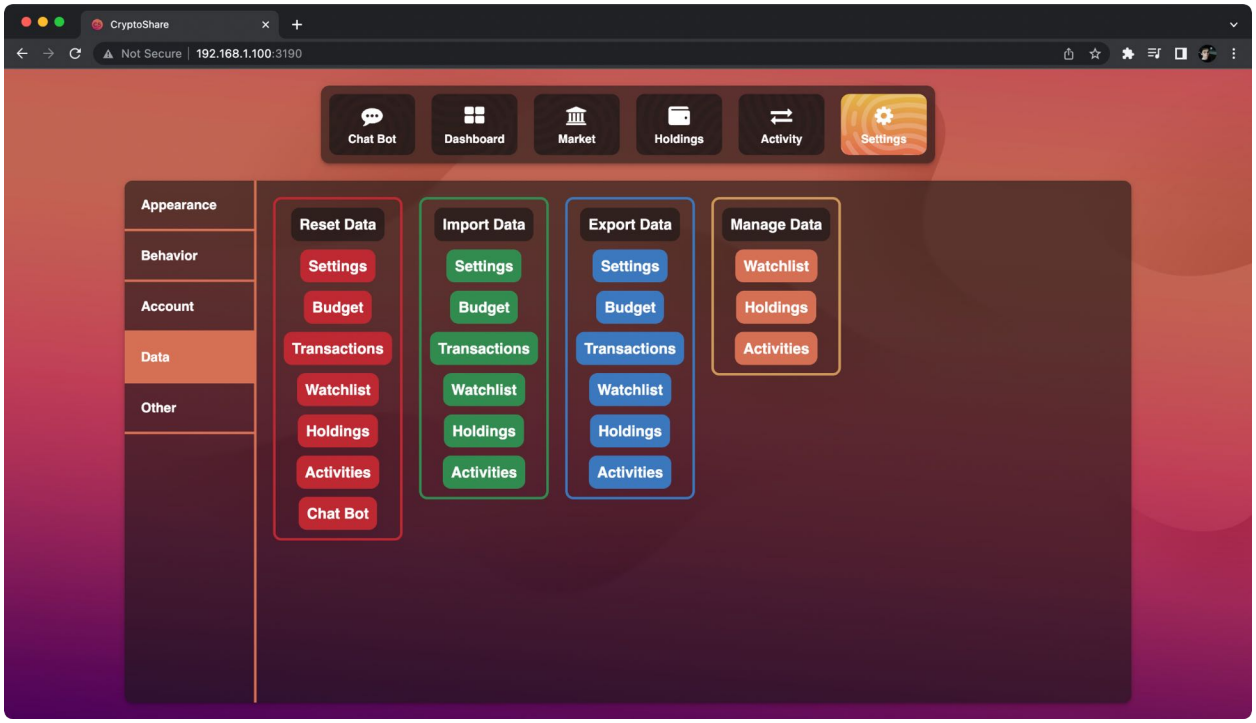
They can use the “Transactions” button to perform CRUD actions on their transactions:



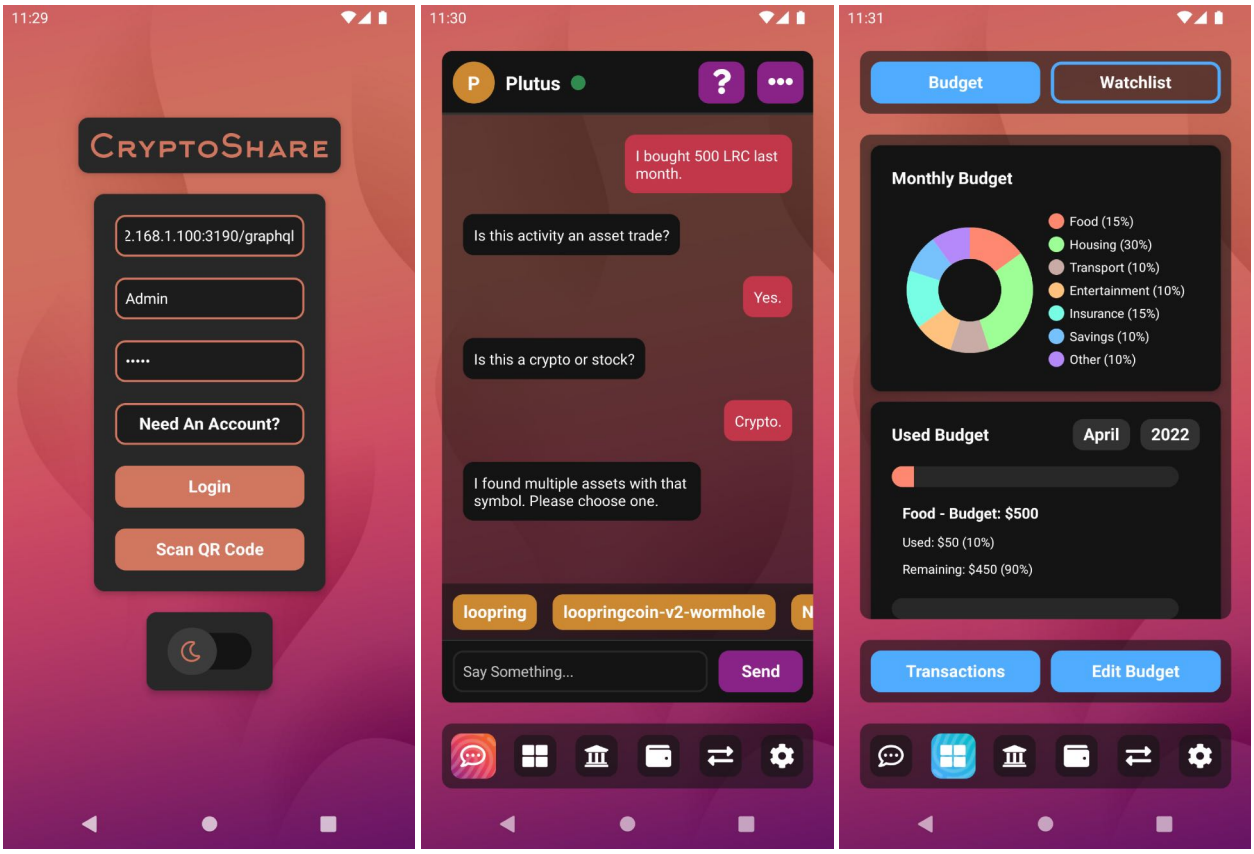
Through the “Settings” page, users can change the appearance and behavior of CryptoShare:



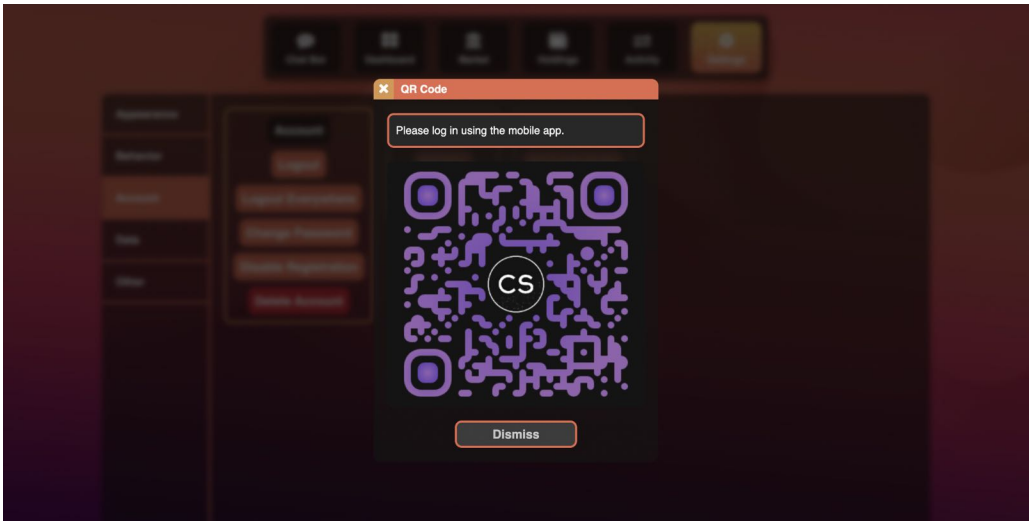
They can also manage their account and data:

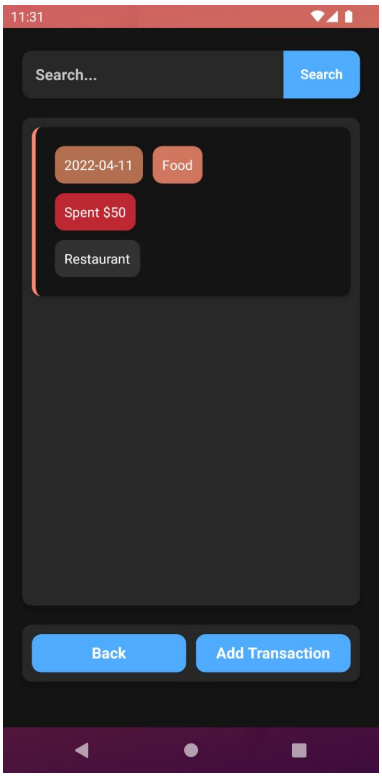


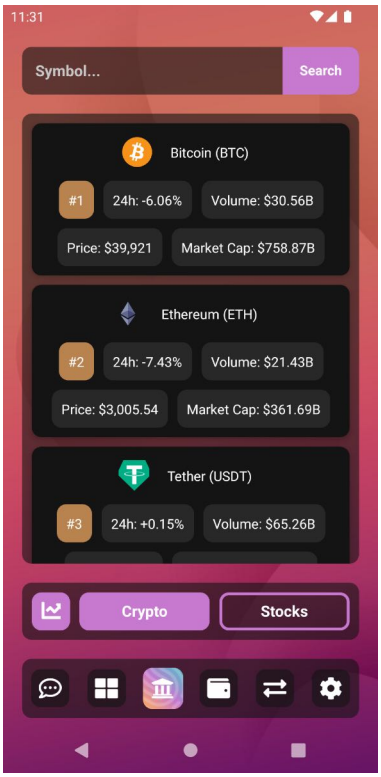
Mobile App

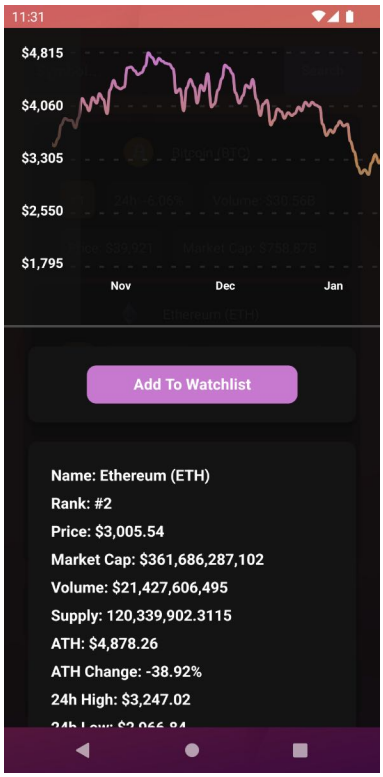


Since one of the main goals of CryptoShare was to offer a unified experience irrespective of the platform the user is on, the mobile app is almost identical to the web app. Users can log in by entering the URL of the API, either by entering the IP address of the device hosting it, or by using the QR code scanner to log in:

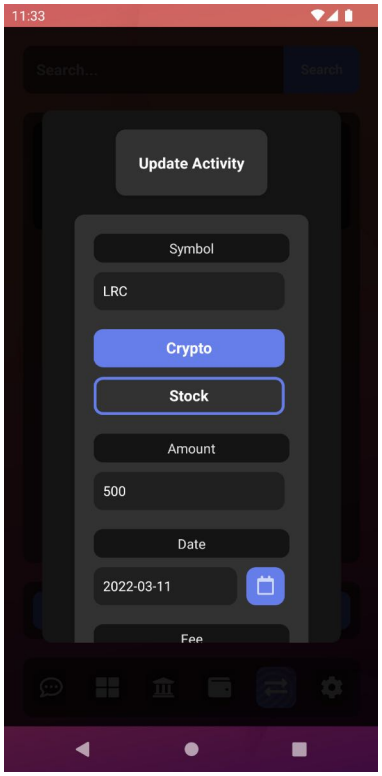


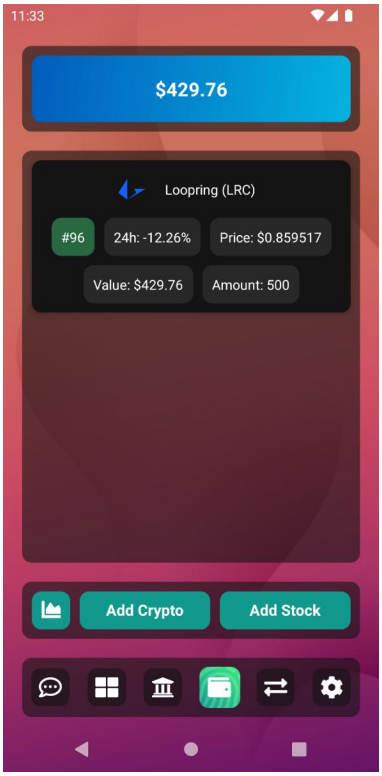


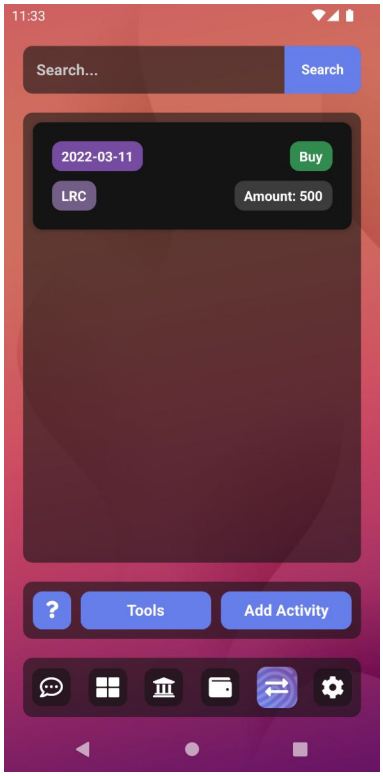


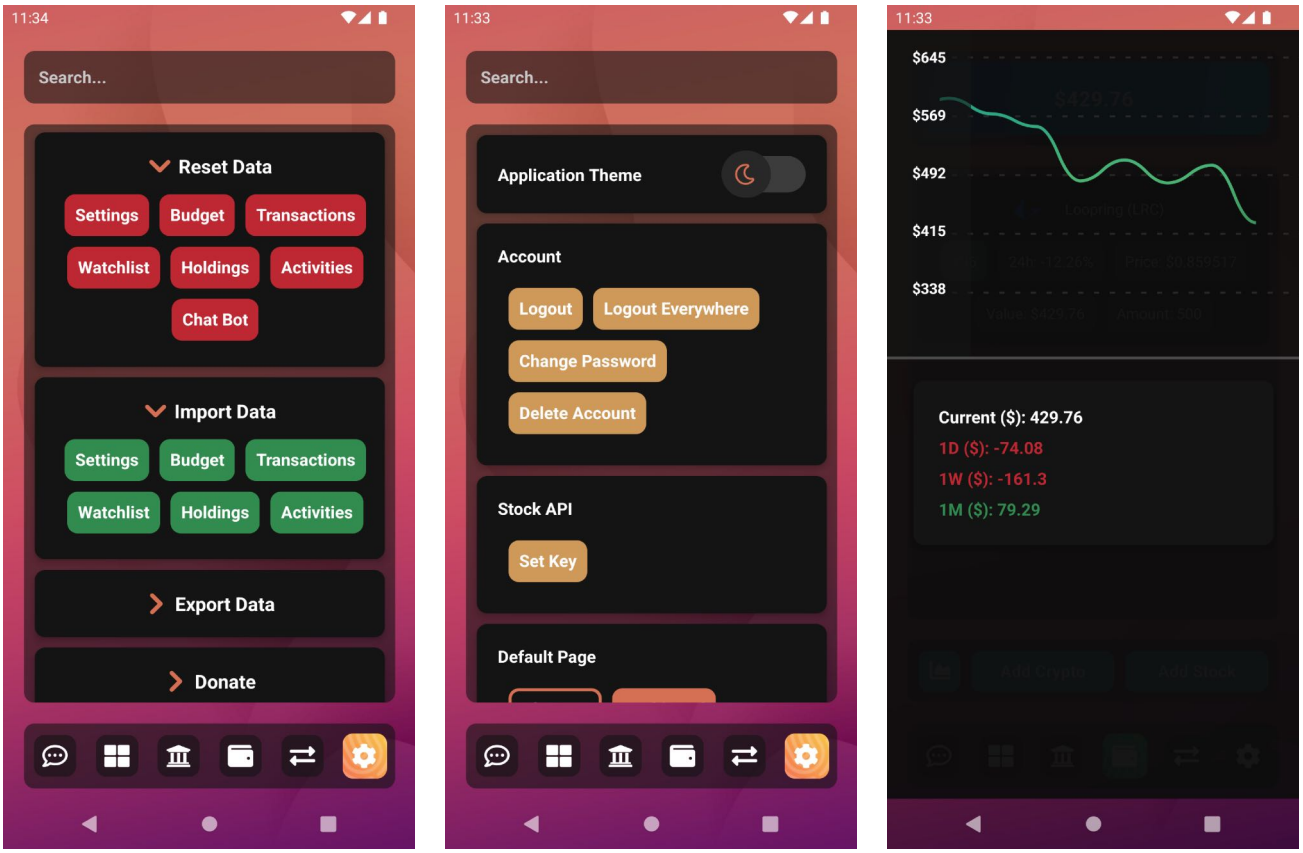


Managing transactions, viewing market data, checking charts etc. is all possible on the mobile app as well.









When designing the mobile app, one of the key goals was to provide just as much functionality as the web and desktop applications, without cluttering the UI. This is because, on mobile, users spend 88% of their time using apps rather than mobile websites (Wurmser, 2020). As a result of statistics like this, it is incredibly important for users of the mobile app to feel as though they have the same amount of control and functionality, and that the application actually “feels good” to use. Responsive websites and applications that simply load a website in a WebView component are not good enough to provide a seamless experience.