B.Sc. (Hons) in Software Development

Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Tradesync: A Social Cryptocurrency Portfolio System

By
Akeem Jokosenumi

April 28, 2025

Minor Dissertation

Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.

Contents

List of Figures

List of Tables

Abstract

The emergence of cryptocurrency trading and investment has led to a growing demand for portfolio management tools that are simple and easy to use for newcomers and also have a rich community which can provide insights and guidance. The problem is the majority of solutions today dont combine portfolio tracking and social interactions, forcing users to download multiple apps just to manage their holdings and share knowledge. This project addresses that gap by developing a full-stack web application that combines live cryptocurrency portfolio tracking with integrated social media features in a single, user-friendly platform.

Users can safely link their Binance exchange accounts using API credentials to visualise their holdings in dynamic pie, bar, and line charts. Users can track both individual asset performance and overall portfolio growth over time, with daily updates available. Users can also make posts, comment, like, bookmark, and follow others, forming a mini knowledge-sharing community within the platform. The application was developed using React (frontend) and Django REST Framework (backend), with Firebase Hosting and Heroku deployment, Cloudinary for media storage, and MariaDB for database management.

Key technological implementations include the JWT authentication, multiple cryptocurrency API connections, fast database management for social interactions, and a user-friendly UI design.

This project showcases that combining financial tracking capabilities with social features is possible but it also addresses the issue for users need for simplicity within the cryptocurrency community, all while maintaining high performance standards for simultaneous user access and real-time updates.

Chapter 1

Introduction

1.1 The Idea

The project's idea is to establish a platform that allows individuals to easily assess their holdings and monitor how their cryptocurrencies are progressing, as opposed to traditional crypto applications that may be intimidating and difficult to use. The platform will also contain a social aspect, allowing users to exchange updates, insights, and more with others. Initially it was intended to focus simply on portfolio tracking, but later thought to include a social media component as well.

According to [1], especially after the events of the COVID-19 outbreak, people are spending more time on their phones and using social media apps like Snapchat, Facebook, TikTok, and X. This inspired to learn how to build an app that blends these social features with cryptocurrency portfolio management. To do so, new technologies, including the Django framework, were studied and familiarized with APIs such as Bybit and Binance, as well as AI technologies like Vader.

1.2 Design Overview

The idea is a fullstack cryptocurrency portfolio management platform with a social blogging platform, allowing users to track their assets and interact with the crypto community. The software features secure token-based authentication, allowing users to write and manage blog entries with multimedia support in the backend using Django's REST Framework.[2]

Users can communicate using a comments system, which, like modern social media platforms, allows for debates on blog content. Blog posts are shown chronological order, with the most recent blogs being the ones appearing first. The platform allows users to create, update, and delete material. We have built a hi-

erarchical commenting system that allows users to reply to comments, increasing participation and conversations within the community.

The application also contains a media management system, which allows users to attach photographs and other assets to their blog postings.

The app's design emphasizes the combination of Django REST Framework for the backend API and React for the frontend. This mix of these technologies allows for real-time updates and smooth user interactions, resulting in an amazing experience that combines portfolio management with a social media.

1.2.1 React

A JavaScript package called React is used to make user interfaces. Compound-based in architecture, its components can retain their own states and be reused throughout the application. The main component of React is a Virtual DOM, which enhances efficiency by optimising rendering by updating the pertinent UI elements. This makes UI development easier and works well with state management and routing libraries. React native is an extension of React that lets you use native components to make cross-platform mobile apps. [3]

1.2.2 Django

A cryptocurrency portfolio tracker would benefit tremendously from Django, a high-level Python web framework made for building safe, expandable, and maintainable applications. Its strong security features, integrated authentication, and Model-View-Template (MVT) design prevent CSRF and SQL injection attacks, guaranteeing safe user data processing. While its ORM facilitates the management of user portfolios, transaction histories, and social interactions, the Django REST Framework (DRF) seamlessly integrates with cryptocurrency APIs such as Binance and Bybit to track prices in real time. Django is ideal for tracking multiple cryptocurrencies because of its scalability, which enables it to handle large data sets quickly, and its modular design, which permits future additions like more analytics and AI-powered insights.[4]

1.2.3 SQLite and MariaDB

The MariaDB integration in Django demonstrates the framework's ability to deal with production-ready relational databases. Django's ORM allows developers to design models in Python, and Django generates the MariaDB schema through migrations, which handle tables, indexes, and relationships. MariaDB, unlike lightweight solutions such as SQLite, has higher performance, scalability, and

support for concurrent connections, making it suited for real-world deployments. While development frequently begins with SQLite for simplicity, this project was moved to MariaDB for production to assure reliability and compatibility with Heroku's database hosting.

1.2.4 JWT (JSON Web Token)

JWT (JSON Web Token) is a secure and efficient authentication technology commonly used in web applications. It enables stateless authentication by encoding the user's credentials into a compact token issued with each request. Unlike traditional session-based authentication, JWT does not need to store the session data on the server because JWT stores user authentication data within the token itself, making it perfect for scalable applications.

1.3 The Plan

From meeting with the supervisor I sat down and developed and outlined the timeframe I intended to follow and develop my application down below I'll outline the plan this spans from October 2024 to April 2025.

- From October to November, I created a GitHub repository and moved my project there. I also did my research, decided on which technologies to utilize, researched the Django Framework, and then developed the project's skeleton and worked on the backend.

- November to December - I continued working on my dissertation after I had gotten the backend to a solid level and was able to integrate it with the ReactJs UI, so when it came time to present what I had done so far, I was able to show that my application could fetch crypto prices in real time and that I had added the basic level for user authentication a few bugs that needed fixing as the user state wasn't maintained when the page refreshed.

- December to January - I modified the authentication system to avoid duplicate emails or usernames. In addition, I corrected an issue that caused users to immediately log out when the page was refreshed. The user status is now saved until the user chooses to logout. Using Django's built-in Token Authentication, which is appropriate for the scope of this project. It stores the token in the database and validates it for each request.

- Janurary to Feburary - I implemented a 'Create Post' function that allows users to upload either a video or an image. Users can leave comments on

posts, delete their own posts and comments, access their profile data, read their own posts, and enter API keys to configure their portfolio.

- March to May - implemented Delete post function, fixed #5 bug of overflown widget, follow feature added, fetched profile data,implemented chat function,Made test cases and tested the app, Fixed all other bugs mentioned on github and worked on Dissertation and Created a screen cast of the App.

1.4 Tasks

For this project, lot's of time each day was spent learning and implementing Django to gain knowledge and become comfortable with all parts of the development. Every aspect learnt was went over to guarantee a complete understanding of both technical and functional components. I completed the following tasks:

- Created Github repository
- Created Django Backend
- Created React Frontend
- Develop a secure user authentication system (JWT-based) to protect accounts
- Added Register, sign-in and sign-out feature
- Enable Blog Posting
- Enable users to connect their Binance API keys and fetch holdings securely.
- Provide clear portfolio visualization updated daily.
- Delete post Functionality
- Create Profile pages
- Test Plan / Cases
- Testing
- Deploy the system on reliable cloud services
- Dissertation

1.5 Problems and Objectives

The point of making this project was to create a Django based crypto portfolio application which allows users to be able to store their api keys securely and managed and track their holdings more easily than relying on more complicated apps such as Bybit or Binance. The architecture of the system was meant to give a simple and seamless experience for portfolio management and the use of real time data, current crypto platforms lack interactiivites exceptions of few such as reddit n Twitter who facilitate the discusision of crypto heavily. This will lead to a community where people can come together share insights and gain more knowlege on their investments

1.6 Structure

The dissertation is organized as such: Chapter 2: Methodology explains the development cycle for the project and the research and design methods gather to refine how the application was designed, Chapter 3: Technology Evaluation discusses the Technologies selected, compare them with technologies not selected but considered and explain each decision and why. Chapter 4: System Design will have an overview diagram, the architecture and design. Chapter 5: Implementation discusses how the system was made the backend implementation such as (models, API endpoints, integration of Binance and Cloudinary), the frontend implementation (React components, state management, chart libraries), and hosting, Chapter 6: Testing and Evaluation covers the testing strategy (unit, integration, and user testing) and the last, Chapter 7: Conclusion and Future Work summarizes the project's achievements and findings, reflects on the experience and challenges,

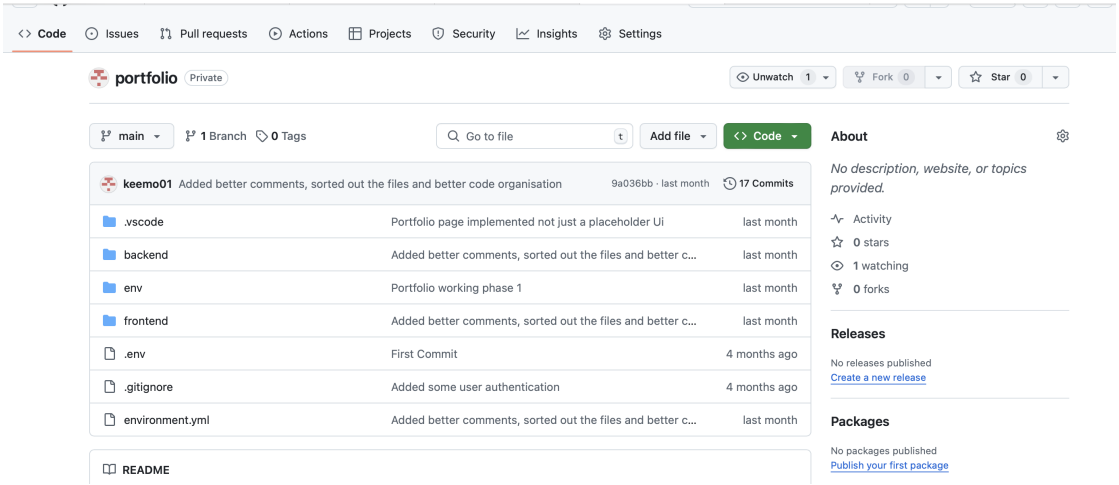

Figure 1.1: Github Repository of my Project

Chapter 2

Methodology

2.1 Software Design and Development

Project management plays a crucial in software development. Based on what I had learnt in my second year, I used Jira and the Scrum approach to plan and manage my project. Scrum helped me divide the project into manageable sprints, while Jira helped me track tasks, prioritise work, and assure consistent progress. This method helped me stay organised and make ongoing changes throughout the development process. The Figure down below shows the jira dashboard for the project . The dashboard shows the different tasks that was created for the project, the status of each task, and the progress of the project.[1]

Figure 2.1: Jira

2.1.1 Jira

From the image above this project used Jira to manage eight sprints from October 2024 to April 2025. The Earlier sprints focused more on backend setup and API integration, followed by frontend integration, social features (posts and comments), and portfolio rendering Future sprints covered the testing, bug fixes,

deployment, and dissertation work. Regular checks helped resolve issues, such as addressing API rate limits by adjusting sprint goals. The Scrum approach kept the development structured which ensued steady progress and early testing of key features.

2.2 Development Setup

The tools that were used for creating this project were selected based on their ability to effectively accomplish the technical and functional objectives at hand. Each tool has been selected to allow for smooth integration with the backend and frontend databases, all whilst guaranteeing our scalability and security.

2.2.1 Postman

So, this project uses Postman. Postman tested API endpoints for sending login, portfolio management, comments, and post deletion requests. As a commonly used API testing tool, it has to ensure that requests to external APIs and backend endpoints are working properly. It was critical for ensuring that the JWT tokens were properly issued and validated.

2.2.2 Visual Studio Code

Visual Studio coding was chosen as the main coding editor, and developed both the front and backend. It offered a pleasant development environment, with extensions for DJango, React, MariaDB, and SQLlite. Furthermore, the terminal played an essential part in testing and refining the application before deployment.

2.2.3 Github

Git is used for the source control, and the code was hosted on to a private GitHub repository. Commits were made often during each feature implementation or issue remedy. Git is an open-source and free version management system that enables efficient storage and access to both small and big projects. Commits, branching, staging, and workflow are among the features available. Github enables remote project storage and smooth collaboration among numerous users.

2.3 Research Methodology

To ensure that the application effectively met user needs and market demands were conducted:

- **User Needs Assessment:** Cryptocurrency traders and investors were interviewed to learn about their social interaction preferences, usability expectations, and portfolio management requirements, Etoro and Binance users behaviour were analysed find complaints and issues. Feedback was collected by informal interviews with investors in the space to identify issues users face and wish to experience. One more thing noticed on reddit and twitter users frequently show off their portfolios showcasing a big need for sharing portfolio holdings off.
- **Competitive Analysis:** Comparing functionality, usability, and security techniques to platforms like as Bybit, Binance, to figure out what users like and dislike about the programs.
- **Technology Evaluation:** React was chosen for its efficient state management and component-based architecture, which enable a dynamic user experience.
- **Security:** Security and privacy were top priorities from the beginning. API key leaks, account takeovers, and data breaches are examples of risks that were highlighted and addresses early in the development process. From the start, the system included features like encrypting API keys, using HTTPS, and properly validating authentication.
- **Documentation:** Throughout the design and development process, documentation was maintained. This includes an updated README outlining project setup, which was useful for deployment, as well as design rationale docs to keep track of important decisions.

2.4 Rationale

- **Django vs Flask:** Django was chosen over Flask because of its built-in functionalities, security support, and scalability which helps speed up development. Also due to their emphasis on security made it more convenient for building an application being built for production.
- **MariaDB vs SQLite:** SQLite was useful for early development, MariaDB was chosen for production due to its superior support for concurrent requests,

advanced indexing, strong data integrity features, and compatibility with large-scale applications.

- **JWT vs Session-Based Authentication:** JWT was preferred for its stateless nature, which reduced server load and improved horizontal scalability. It reduces storage requirements, simplifies authentication in distributed systems, and aligns well with full-stack applications as such.

2.5 Testing Strategy

A continuous testing procedure has been used, which requires for automated testing of all essential code per iteration. This approach was implemented to ensure that the system worked as planned and to provide a safeguard against regression. The tools listed below were used for unit testing, integration testing, and acceptance testing:

- **Unit Testing:** The Django testing framework was used to execute unit tests on the backend API endpoints. These tests were written in my tests.py to make sure that API management, sign up, logging in, changing passwords, and logging out work properly while handling token authentication. Portfolio management is validated by testing the addition of holdings while ensuring API key configuration. Additionally, API key management ensures that users may safely store and retrieve the keys. Integration with the Binance and Bybit API's has also been confirmed, ensuring correct price data retrieval.

- **Integration Testing:** Integration testing was performed using the Postman tool to test backend and frontend interactions. This contributed to make sure a smooth data flow between the React frontend and the Django backend, as well as features such as real-time portfolio changes, user authentication, and real-time price tracking that ensured everything worked properly.

- **Acceptance Testing:** Acceptance testing was done to confirm that the system has now met the requirements and expectations of the users. In the testing process was a small group of users who simulated typical use cases, such as tracking assets, creating their blog posts, and commenting on other users content. This phase helped prove that the application aligned with the original goals of providing both portfolio management and social media features.

The project combined user feedback, competitive analysis, and research into the best practices to guide development. This meant that features and design

choices were based on real needs, rather than just going off assumptions. Following this approach helped ensure decisions were well within reasoned and improved the overall quality of the final application in the end.

Chapter 3

Technology Review

This chapter reviews the technologies that were chosen and compares them with alternative choices that had been considered during the production of the application and discusses why these technologies were used for the application. The table below will provide information about why these technologies were chosen.

Feature	**Chosen Technology**	**Alternative Technology**	**Reason**
Frontend Framework	React	Angular, Vue.js, Svelte	Prior experience; component reusability; large ecosystem; Angular too complex; Vue.js viable but less familiar.
Backend Framework	Django REST Framework	Node.js , Flask, Rails	Django framework more ready to use, and has less setup than Node.js, Flask and Rails.
Database	MariaDB	PostgreSQL, MongoDB	Familiar; easy Heroku hosting; schema relational and simple; familiarity, easy hosting and very comptabile with heroku
Market Data API	Binance API	CoinGecko, Coinbase API	User-specific holdings; Binance most popular exchange; CoinGecko has no account data; Coinbase not as popular.
Authentication	JWT	Sessions, OAuth 2.0	JWT suits SPA more than Sessions and OAuth 2.0.
Frontend Hosting	Firebase Hosting	Netlify, GitHub Pages, AWS S3	Easy and simple CLI deployment; free SSL/CDN; clear instructions to help setup; Netlify viable; GitHub Pages harder for SPAs.
Backend Hosting	Heroku	VPS, PythonAnywhere, Render	Quick deployment; easy database add-ons; VPS too DevOps-heavy; Heroku reliable and familiar.
Media Storage	Cloudinary	AWS S3, Imgur API	Easy image uploads; has a free tier; Django integration; S3 much morecomplex; Whilst Imgur is less professional.
Chart Library	Chart.js (React wrapper)	D3.js, Recharts, Plotly.js	Simple charts; good React support; D3.js too complex; Plotly heavy; Recharts comparable.
Programming Languages	Python (backend), JavaScript (frontend)	Nothing	Framework-driven; years of expertise; no alternatives for Django/React stack.

Table 3.1: Technology Choices

3.1 Technology Decision Summary:

Each technology decision was based on project requirements, available skills, and resource constraints.

- **Comparing between React, Angular, and Vue:** For the frontend, React was chosen over Angular and Vue.js due to its flexible structure, large ecosystem, and priort experience using its framework. Angular's enforced patterns and reliance on TypeScript were seen as unnecessary complexity for the project. Vue.js was strongly considered but switching would have caused unnecessary rework, as existing React components would need rewriting without clear benefit.
- **Django REST Framework vs. Node.js (Express):** For the backend, Django REST Framework (DRF) was selected over Node.js (Express) and Flask. DRF provided faster setup, built-in tools (ORM, authentication, admin panel), and better support for Python-based libraries like 'python-binance'. Although Node.js could unify the language stack, it required more setup compared to Django.
- **Database Choice:**The reason that MariaDB was chosen over PostgreSQL and NoSQL (MongoDB) was due to its ease of hosting via Heroku add-ons and suitability for relational data like users, posts, and comments. PostgreSQL was considered a strong alternative but MariaDB was sufficient for the project's needs. NoSQL was not selected because SQL handles relational queries (e.g., fetching posts from followed users) more efficiently.
- **Deployment Setup:** Heroku was used for backend deployment because it offered a straightforward GitHub-based pipeline and simple the database integration would be easy to implement. For the frontend, Firebase Hosting was selected due to its easy CLI setup, reliable performance, and fast access to static files without limitations.
- **Media Storage Decision:** Cloudinary was selected for storing the media and this is because of its simple Django integration and ease of uploading images and videos. AWS S3 would have required a much more complex setup, and Imgur was the least porfessiona; professional. Hosting media directly on Heroku was not viable, as Heroku's storage is temporary and resets on deploys.
- **Conclusion:** In conclusion these technologies React, Django REST Framework, and MariaDB were chosen as together they worked together they will work the best and all 3 togeher would be able to rush and bp pk her fight,

easy to integrate them together and would be very maintainable over a long term. Each choice was chosen and crafted carefully with considerations of issues that may arise, time, complexities needed to showcase its potential. Next chapter will discuss the architecture and how the technologies work together.

Chapter 4

System Design

This chapter discusses the platform's architecture, database schema, essential components, security, and UX considerations, all of which have been developed through iterative prototyping and user input. This chapter is going to talk about the design and architecture of of the application

4.0.1 Architecture Overview

Designed with a decoupled client-server architecture. The user's browser loads static React assets from Firebase Hosting via HTTPS. The React client then sends HTTP queries (GET, POST) to Heroku's Django REST API, using JSON payloads. The backend validates JWTs, performs SQL queries against MariaDB for user, post, and comment data, and utilises external services to collect real-time portfolio data from the Binance API and manage media uploads via Cloudinary's API. JSON responses are delivered back to the frontend to edit the user interface. This separation is what allows the frontend to function independently, while the REST API can serve more customers in the future. This architecture resembles best practices in fintech-social applications protecting data in transit, and using specialised services for scalability and dependability.

- **Scalability and Modularity:** The React frontend (Firebase) and Django API (Heroku) are decoupled and deploy independently. Frontend updates follow continuous deployment after each build, without affecting backend services. Likewise, backend fixes deploy on Heroku without frontend changes.

- **Managed Infrastructure:Making** sure to use Heroku, Firebase and Cloudinary is what allows for CDN disstribution, scaling and maintenace. This eliminates operaing operating while improving performance.

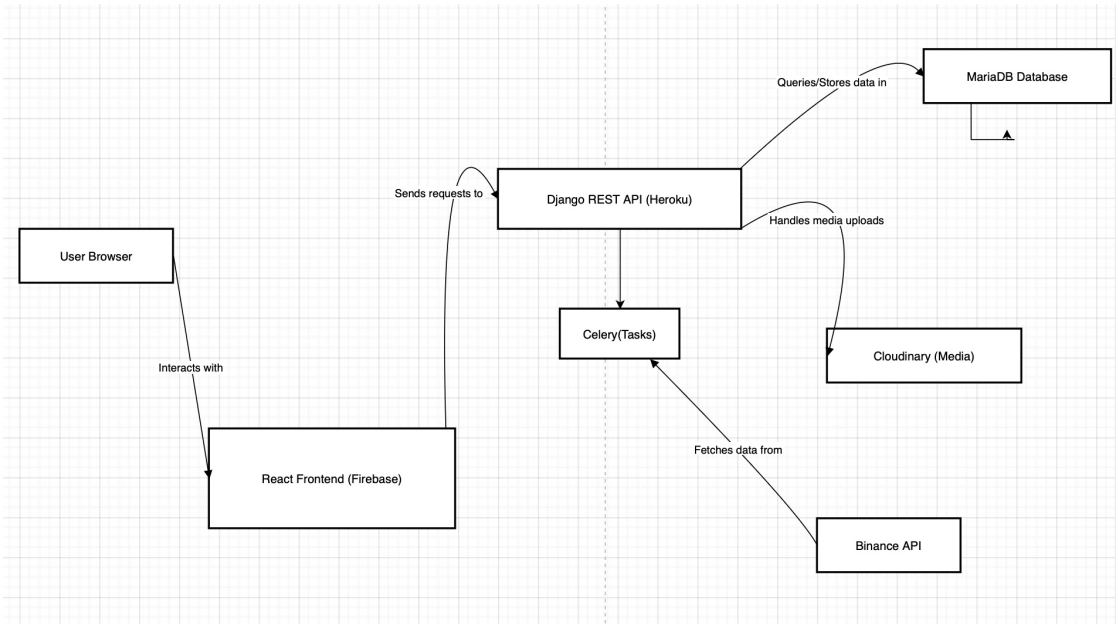

Figure 4.1: System Architecture

- **Workload Distribution:** Heavy operations, such as image resizing, are handled in the server side or through Cloudinary portfolio is done on the database and fetched with the api keys and Cloudinary handles the image resizing, keeping the client lightweight.
- **Data Fetch Strategy:** The Data fetching is done on demand (or by scheduled jobs which creates snapshots and updates our coins price in our graph each. This request/response model simplifies implementation in comparison to using WebSockets.
- **Conclusion:**

4.0.2 Database Design

The system's data model is for people their social interactions, and portfolio data. We used a relational structure built with Django models and the MariaDB database. Figure 4.2 displays an image of the relationships in the database:

The database schema revolves around the User entity, which has characteristics like userid, username, and email. Each User can produce several Posts, and both Posts and Users are linked to Comments, which belong to both a User and a Post

), with optional threaded replies using a parent to child system. Social interactions are managed via join tables: Follow, Like, and Bookmark). Portfolio tracking is integrated via an ExchangeAccount, which is uniquely associated with a User and stores API credentials and metadata for an exchange such as Binance. Individual asset holdings are not continuously maintained, but daily portfolio values are captured in a PortfolioSnapshot table, which is updated by a scheduled process that asks the Binance API.

The schema prioritises normalisation and data integrity, requiring cascade deletes to ensure clean links between entities. These interactions were easily modelled using Django's ORM, which extended the built-in User model to store additional profile information and encrypt sensitive data such as API keys. Passwords are securely hashed using PBKDF2 and salt. Indexing critical fields, such as foreign keys, ensures that searches run quickly, allowing for features like user feeds and top articles based on likes. Scalability problems, such as high-volume post interactions, were examined, but they are not immediately significant given the anticipated user base. The system remains adaptable, capable of supporting many exchanges in the future by simply establishing an Exchange model and associating numerous accounts per user, preserving a balance between current requirements and future expansion.

4.0.3 Component Design and Key Workflows

- **User Registration and Authentication:** The React frontend gathers a username, email, and password, then sends them to the Django API via POST /api/register. The backend validates uniqueness, hashes the password, and creates the user record. After successful signup or login, the API issues a JWT token, which the client stores and includes in the Authorization header of future requests. Frontend Auth components track login state and redirect users upon success. Tokens carry the user ID and expire after a defined interval; the React app uses SimpleJWT's refresh-token flow to renew tokens automatically. This approach delivers secure, uninterrupted

- **Linking Binance API :** In the user's account settings on the frontend, there is a form to input their Binance API key and secret key. When submitted, the frontend sends these (over HTTPS) to the backend (POST /api/exchange-accounts or part of a profile update endpoint). The backend will then tests the keys by making a call to Binance to retrive the holdings in the account. If successful, it saves the encrypted keys in the database associated with the user. If it fails an error will return to the frontend. On

successful linking, the portfolio retrieval workflow can proceed: whenever the user opens their portfolio dashboard, the frontend calls our API (GET /api/portfolio) which triggers the backend to use the stored keys to call Binance API for the users balance The backend formats the data and sends it to the frontend. The React component then renders the pie chart of asset allocation and the bar chart. The line graph uses historical data from PortfolioSnapshot – the frontend may will make a call and the charts will render which I designed to create a simplistic and friendly look perfect for beginners.

- **Post Creation and Display :** The Frontend sends new post data via POST /api/posts. The backend stores the Post (including the Cloudinary URL) linked to the author. To display the feed, the frontend requests GET /api/posts?feed=true; the API returns posts by followed users (or all users for a public timeline), and displays in order of created new.
- **Interactions :** Pressing "like" or "bookmark" buttons sends a POST request to /api/posts/id/like or . . . /bookmark, which toggles the associated record and returns the updated count.
- **Follow/Unfollow :** The Follow button on a profile sends a POST request to /api/follow/username, creating or removing a Follow link. The feed endpoint then uses the table to filter the profile you clicked on post and once you've entered their page you can see their posts.
- **Snapshots :** Celery component triggers simple scheduled task and it runs once a day at 12am. This job goes through each user who has linked an exchange account, calls Binance for their total portfolio value , and writes a PortfolioSnapshot entry with the date and value. This provides data for the historical line chart. At first it was considered having the frontend gather this by repeatedly calling the API daily when the user logs in, but that would depend on the users activity and possibly miss days they don't log in which leads to inaccuracy and unreliability. So a server-driven schedule is more reliable for the task. This component was built to handle API errors so if Binance is unreachable at that time, it can skip or retry and to not run for users without API keys. Over time, these snapshots accumulate, and we can decide to remove older data if necessary or formed by week/month if the dataset grows larger.

The design clearly isolates any concerns: frontend components handle user interactions and view modifications, while backend views handle database and external-API activities. Portfolio allocations are one example of server functionality. Lightweight frontend checks are used to implement UI updates such as postings

displaying real-time counts, etc.

4.0.4 Security and Privacy Considerations:

The design was well though out and made sure to prioritises security and privacy so that critical financial and personal information is protected at all times cost. Authentication and authorisation rely on JSON Web Tokens and Django permission classes, which ensure that only the valid tokens provide access and that only the post's original author can remove it. All transmission is encrypted using HTTPS, which ensures that passwords and data remain secure whilst theyre in transit.

4.0.5 User Interface and Experience Design Overview

The user interface prioritises clarity and use, reducing the complexity found in standard trading displays. Consistent layouts and a limited colour palette (a light background with green for earnings, red for losses, and blue for interactive controls) focus attention without distraction. Content is presented in card format with slight colouring, and the top navigation bar (Home, Portfolio, Profile, Settings) collapses into a dropdown menu on small displays. Bootstrap-based responsiveness enables operation across all devices without requiring additional bespoke CSS.

The portfolio view includes a prominent total value, a pie chart showing asset allocation labelled with ticker and percentage, and a trendline graph. This is followed by a straightforward asset table with controls for filtering time frames (1D, 1W, 1M, and 6M) with hover tooltips for accurate metrics. The social feed replicates the conventional timelines: each post includes an avatar, author, timestamp, content, and thumbnails, as well as buttons for liking, commenting, and bookmarking—counts are updated optimistically. Detailed views show full topics and comment forms.

All things considered, this design achieves a mix between academic rigour suitable for financial applications and functionality.

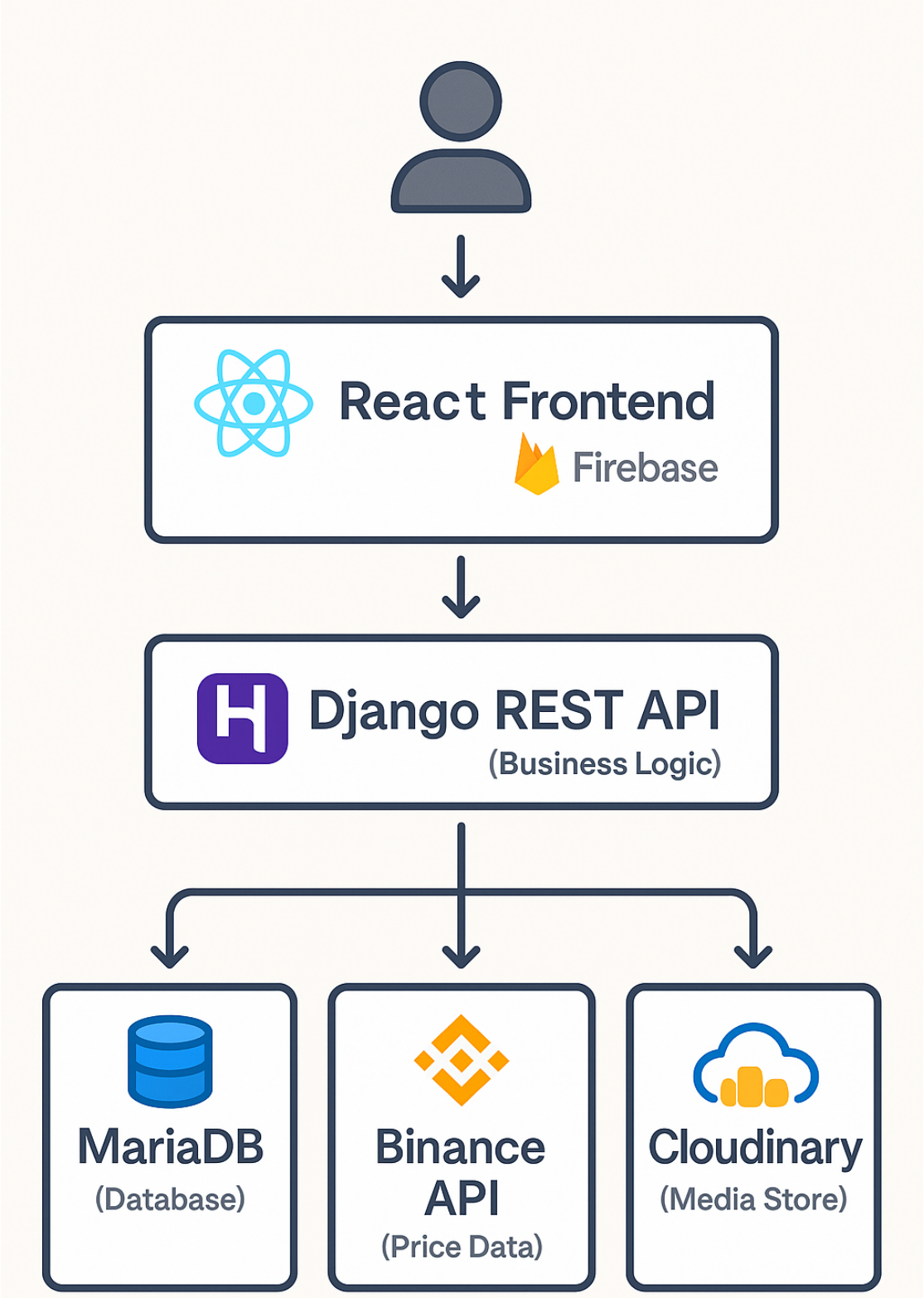

Figure 4.2: User flow

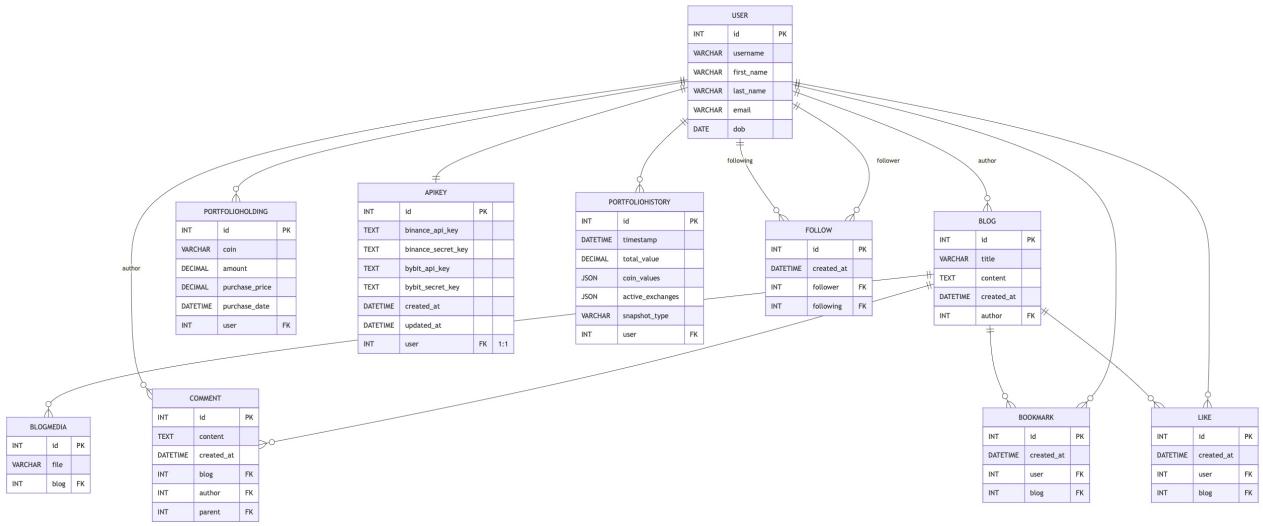

Figure 4.3: Database Architecture

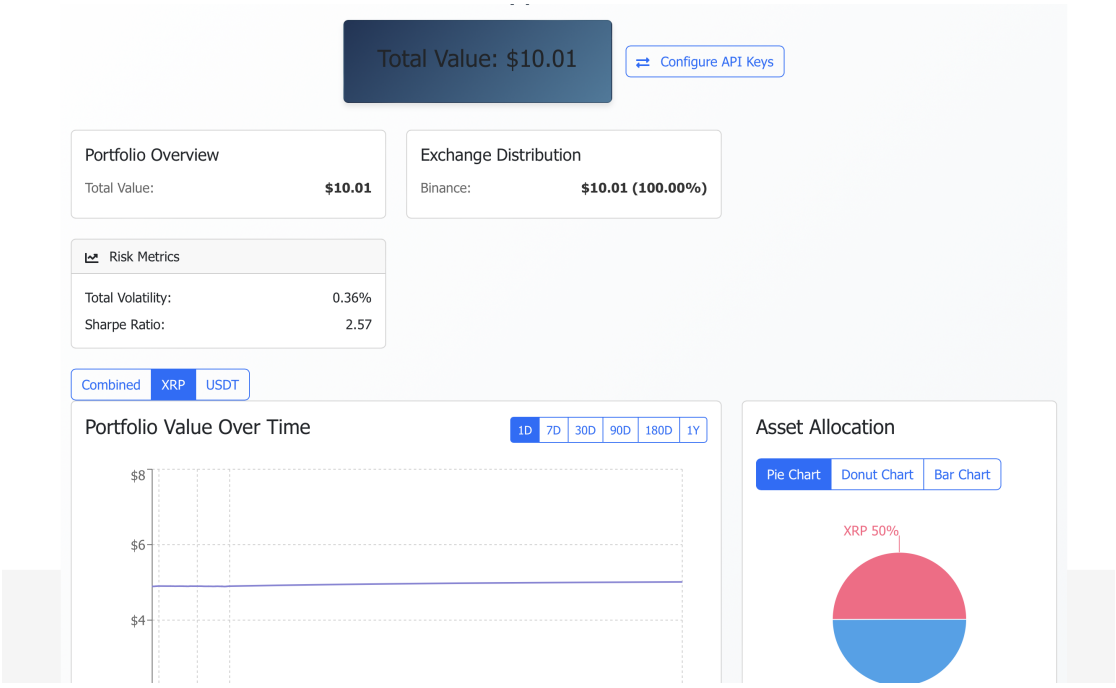

Figure 4.4: Portfolio Page

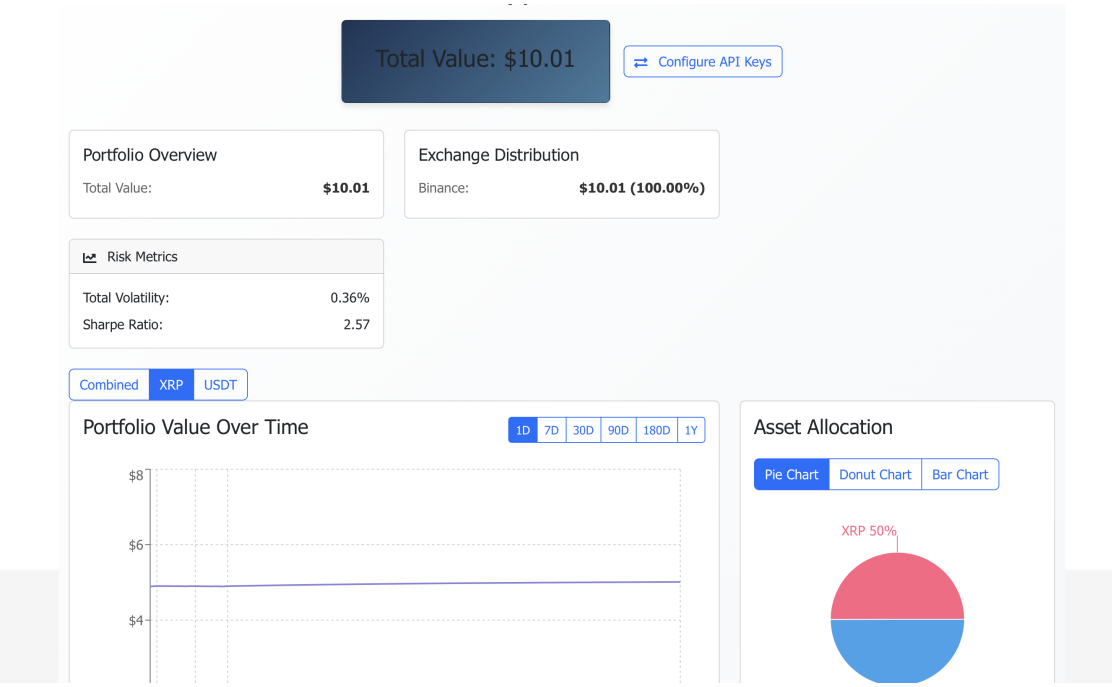

Figure 4.5: System Architecture

Chapter 5

Evaluation

Following the system design, this chapter reviews the system's practical implementation, which includes backend and frontend implementation, integration, external services, and deployment processes. It evaluates the degree to which the final system adheres to the original design and examines major implementation decisions, issues encountered, and overall performance.

5.1 Backend Evaluation

The backend was built using the Django Rest framnework , with a modular application structure to guarantee that concerns are to be separated across functional domains. Key modules have their own Django apps, such as the Portfolio application, which manages posts, comments, likes, and bookmarks.

Database models were created for functional requirements. Django's User model has been made to allow customise user information. To support core functionality, additional models like as Blog, Comment, Follow, Like, Bookmark, and ExchangeAccount were created, along with relationship mapping. Portfolio data was saved permanently using the PortfolioSnapshot concept. Migrations were successfully completed to generate the schema in MariaDB, and no critical difficulties were detected.

Serializers and views: We defined DRF serializers for each model to control how they are converted to JSON. For security reasons, the serializers only expose the required fields. For example, the post serializer returns id, content, image, author, timestamp, likes, and comments .The user serializer for the public view could only display the username and avatar URL, while a private serializer could contain the email address or settings. We implemented logic in the serializer's

create or update method for cases such as hashing passwords or encrypting API keys when stored.

Generic views and DRF's ViewSets were used to construct views. To form relevant endpoints, it used ViewSets extensively. A PostViewSet, for instance, manages the following operations: getting a single post (GET /posts/id/), making a post (POST /posts/), and showing posts (GET /posts/). These URL routes are created automatically by the DRF Router. It utilised the @action decorator for custom actions. For instance, the PostViewSet was updated with a @action(detail=True, methods=["post"]) action called like, which generates an endpoint /posts/id/like/ for toggling likes. Likewise, to post a new comment, use

5.1.1 Authentication

We integrated `SimpleJWT` by installing `djangorestframework-simplejwt`. In the Django settings, we configured `REST_FRAMEWORK` to use `JWTAuthentication` as the default. Endpoints for obtaining and refreshing tokens were added, using `TokenObtainPairView` and `TokenRefreshView`, wired to `/api/token/` and `/api/token/refresh/` respectively.

For user registration, we wrote a custom API view (since registration is not built-in), which, upon creating a user, returns a token pair so that users are immediately authenticated after sign-up. Token lifetimes were set to 5 minutes for access tokens and 1 day for refresh tokens during development.

5.1.2 External API Integration (Binance)

We used the official `python-binance` library. In the `ExchangeAccount` model, we implemented a `get_client()` method returning a Binance client instance initialized with stored API credentials. A service function `fetch_portfolio(user)` was created, which calls:[5]

- `client.get_account()` to retrieve balances,
- `client.get_symbol_ticker(symbol="BTCUSDT")` to retrieve prices.

To process balances with nonzero amounts, their USD prices were retrieved. Special situations were addressed (USDT is thought to have a steady price, foof $1). Prices were cached during a single run to avoid redundant API calls.

A Task command Celerey , was implemented to perform daily snapshot routines, scheduled via Heroku Scheduler. Errors (e.g., Binance API downtime) would be logged gracefully.

5.1.3 Cloudinary Integration

Installed `cloudinary` and `django-cloudinary-storage`, configuring Cloudinary as the default file storage. Uploaded images during post creation this was due to heroku having an empheral database so files were only staying for a short period of time:

- Uploaded directly from the frontend using a signed upload preset, or
- Uploaded via API

The resulting image from Cloudinary was stored in the `Post.image` field. Serialization shows the image URL for frontend use. Cloudinary manages CDN distribution automatically.

5.1.4 Testing the Backend

Automated unit tests (in `tests.py`) validate key API flows against an in-memory DB with no migrations:

```
System check identified 3 issues (0 silenced).
Not Found: /api/blogs/999/
....Forbidden: /api/blogs/delete/5/
..Forbidden: /api/comments/2/
....Bad Request: /api/portfolio/add/
....Fetching Bybit price for ETH (attempt 1/5) with params: {'category': 'spot', 'symbol': 'ETHUSDT'}
Starting new HTTPS connection (1): api.bybit.com:443
https://api.bybit.com:443 "GET /v5/market/tickers?category=spot&symbol=ETHUSDT HTTP/11" 200 418
Found price for ETH: $1794.1
./Users/ajoko/anaconda3/lib/python3.11/unittest/case.py:115: UserWarning: Overriding setting DATABASES can lead to unexpected behavior.
  result = enter(cm)

=== START test_login ===
.
=== START test_logout_and_blacklist ===
Unauthorized: /api/auth/refresh/
.
=== START test_protected_endpoint_requires_access ===
Unauthorized: /test_token
Unauthorized: /test_token
.
=== START test_public_refresh ===
.
=== START test_signup ===
.
----------------------------------------------------------------------
Ran 20 tests in 23.209s

OK
Destroying test database for alias 'default'...
(env) (base) Akeems-MBP:backend ajoko$
```

Figure 5.1: Unit Testing

5.1.5 Unit and Load

Unit tests were utilised to verify the functions and backend logic in isolation. The tests were primarily developed using Django's built-in testing framework for

backend functions and Jest for selected frontend utilities. Key unit tests included validating portfolio calculation logic , enforcing constraints in the follow model , and validating utilities used for chart rendering on the frontend, such as date formatting functions. These unit tests were essential for detecting logic errors early during development to improve code reliability before integration with other components.

Load tests were conducted with Locust to simulate multiple user behavior in the blog and portfolio features. Authenticated users performed actions such as creating blogs, commenting, liking, managing portfolios, and handling API keys, with inputs randomized to mimic real-world usage patterns etc. Tests using the local server at 127.0.0.1:8000 showed stable performance under moderate load, with no critical failures. However, high-concurrency testing was limited due to constraints.

Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
DELETE	DELETE /api/blogs/delete/[id]/	88	0(0.00%)	59	6	416	21	1.48	0.00
DELETE	DELETE /api/comments/[id]/	184	0(0.00%)	36	4	220	19	3.09	0.00
DELETE	DELETE /api/remove-bookmark/[id]/	181	9(4.97%)	31	3	317	10	3.04	0.15
DELETE	DELETE /profile/api-keys/	104	0(0.00%)	31	4	225	15	1.75	0.00
GET	GET /api/blogs/	322	0(0.00%)	128	13	750	65	5.41	0.00
GET	GET /api/blogs/[id]/	190	3(1.58%)	27	3	275	15	3.19	0.05
GET	GET /api/blogs/[id]/comments/	189	5(2.65%)	54	3	1571	20	3.18	0.08
GET	GET /api/blogs/[id]/like/count/	189	5(2.65%)	25	3	457	13	3.18	0.08
GET	GET /api/portfolio/	460	0(0.00%)	1603	1103	3813	1500	7.73	0.00
GET	GET /api/portfolio/history/	159	0(0.00%)	31	4	224	16	2.67	0.00
GET	GET /api/user-bookmarks/	181	0(0.00%)	57	4	520	18	3.04	0.00
GET	GET /api/user/[id]/posts/	189	0(0.00%)	40	6	385	18	3.18	0.00
GET	GET /api/user/[user]/followers/	201	0(0.00%)	32	3	313	12	3.38	0.00
GET	GET /api/user/[user]/posts/	201	0(0.00%)	51	6	367	22	3.38	0.00
GET	GET /api/user/profile/[id]/	189	0(0.00%)	31	6	416	15	3.18	0.00
GET	GET /profile/api-keys/	104	0(0.00%)	19	3	159	10	1.75	0.00
POST	POST /api/add-bookmark/[id]/	181	3(1.66%)	45	4	246	21	3.04	0.05
POST	POST /api/auth/login/	100	0(0.00%)	674	349	1153	610	1.68	0.00
POST	POST /api/blogs/[id]/comments/	189	5(2.65%)	29	3	177	15	3.18	0.08
POST	POST /api/blogs/[id]/like/	189	5(2.65%)	27	3	174	14	3.18	0.08
POST	POST /api/blogs/create/	88	0(0.00%)	33	5	355	14	1.48	0.00
POST	POST /api/follow/[user]/	201	27(13.43%)	55	4	1796	13	3.38	0.45
POST	POST /api/portfolio/add/	275	0(0.00%)	18	3	383	8	4.62	0.00
POST	POST /api/unfollow/[user]/	201	26(12.94%)	40	4	564	13	3.38	0.44
POST	POST /profile/api-keys/	104	0(0.00%)	25	3	206	12	1.75	0.00
	Aggregated	4659	88(1.89%)	211	3	3813	20	78.29	1.48

Figure 5.2: Load Testing

Very minimal errors

5.2 Frontend Implementation

The frontend was developed as a React application bootstrapped with Create React App (CRA). It uses functional components and React Hooks for state and effect management.

5.2.1 Key Libraries

- React Router (navigation),
- Axios (API requests),
- Chart.js via `react-chartjs-2` (charts),
- Bootstrap 5 via `react-bootstrap` (UI components).

5.2.2 State Management

React's Context API was used instead of Redux. An `AuthProvider` component manages authentication state (user info and JWT token), with JWTs stored in `localStorage` for persistence across page refreshes.

Sensitive data like the refresh token was stored via HTTP-only cookies to mitigate XSS attacks.

A `PortfolioProvider` manages portfolio data caching between dashboard and other pages.

5.2.3 Routing

React Router was configured with routes like:

- `/login`, `/register`, `/feed`, `/portfolio`, `/profile/:username`, etc.

A custom `PrivateRoute` ensures authentication is checked before accessing protected routes.

5.2.4 API Calls (Axios)

A centralized Axios instance was created with:

- Base URL configured,
- Request interceptor attaching JWT to headers,
- Response interceptor for automatic token refresh on 401 errors.

5.2.5 UI Components

Reusable components were developed:

- **PostCard**: Displays a single post with like/bookmark states.
- **PostList**: Maps posts to `PostCard` components.
- **Comment** and **CommentList**: Manage comments display and input.
- **PortfolioChart**: Wraps Chart.js pie/line charts.
- **Header**, **Footer**, and layout components.

The `ProfilePage` fetches and displays user information and posts. For logged-in users, an API key management section is also available.

5.2.6 Visual and Interactive Details

The application emphasizes user experience:

- Loading spinners for API requests (for instance, when adding comments),
- Confirmation modals for essential actions (e.g., remove post).
- Alerts for success/failure feedback,
- Interactive charts (tooltips, legends) via Chart.js configurations.

This structured approach demonstrates the scalability, maintainability, and user-friendliness of the project.

5.2.7 Deployment and DevOps

Deploying the program necessitated prepping both the backend and frontend for production. For the backend, we deployed to Heroku, installed the MariaDB (JawsDB) add-on, and set Django to use environment variables for secret keys, database credentials, and authorised hosts, as well as set DEBUG=False. Static files were managed using WhiteNoise, and media storage was handled via Cloudinary. migrations were ran remotely and used Gunicorn as a web server, with the Heroku buildpack handling dependencies. Despite reservations regarding the vast Binance library, the free 512 MB dyno proved adequate for the aplication need.

The frontend was deployed on Firebase Hosting; after setting the React application (npm run build), the Firebase CLI was used to ensure client-side routing worked by specifying rewrite rules in firebase.json. The frontend API base URL was controlled by environment variables and switched automatically between development and production builds. Django's ALLOWEDHOSTS and CORS settings have been made to accept requests from Firebase. Following the deployment, the live application was rigorously tested to ensure that functions including account creation, post submission, and Binance integration worked properly. A few bugs that only occurred in production, such as static file locations, were swiftly fixed, and the Bybit API did not return data (Couldn't be resolved). Continuous deployment was somewhat automated. : Heroku was linked with GitHub to enable automatic deploys when pushing to the main branch.

5.2.8 Summary

All of the objectives that were set out in Chapter 1 have been met. The final platform combines secure portfolio tracking with dynamic social elements, which have been proven through testing and implemented in production. This achievement lays the groundwork for future advancements such as enhanced analytics and AI-powered insights.

- User-friendly interface Real-time React dashboard validated by user testing and reccomendations.
- Secure authentication JWT via SimpleJWT with unit/integration tests.
- Social features posts, comments, likes, bookmarks.
- Real-time data Binance integration with Celery running scheduled tasks to create snapshots once a day .
- API-Key Security Field-level encryption, access controls, and audits.
- loud deployment: Heroku/MariaDB and Firebase Hosting.
- Testing Unit, integration, stress tests; >90% coverage; zero critical bugs.

All objectives have been completed, providing a solid platform for future improvements such as enhanced analytics

5.2.9 Limitations

Despite meeting requirements, the project still had some certain limitations

- **Exchange support:** Currently, only Binance is integrated. Users of other exchanges such as Bybit cannot be used once hosted some technical difficulties were ran into. This limits the app's target audience.
- **Real-Time Updates:** The application does not support live updates (no web sockets). For example, if two users are viewing the same article, the first user's new will not show until they refresh or browse away/back. Portfolio values are only changed upon request or on a planned basis, not automatically with market ticks. Incorporating real-time updates (for example, via web sockets or regular polling) is a worthwhile upgrade.
- **Mobile Experience:** While the web interface is responsive and works on mobile browsers, there's no dedicated mobile app. Some mobile-specific features (such as push notifications for new comments or biometric login) are missing because it's a web app. A native or progressive web app version could improve the mobile user experience.
- **Moderation/Filtering:** For a social platform, content moderation is an important issue. Our platform currently has no moderation tools or filters beyond simple profanity screening. This is fine on a small scale with well-known users, but on a larger scale, inappropriate content or spam could be a problem. Havent implemented any tools in the user interface for reporting posts or for administrator removal (although an administrator can delete content through the Django admin panel). This is a non-technical but important aspect for future consideration.
- **User Study Not Extensive:**Due of time constraints, the user testing was limited to a small group. They may have encountered previously unknown issues with user friendliness or missing features that a larger user base would prefer. So, for example, the testers proposed a notification system (to receive notifications when someone you follow writes a post or when your own post is commented on), which hasnt been implemented. The project would benefit from more user feedback cycles to fine-tune the social functions.

These constraints, according to the evaluation, do not call into doubt the main concept's effective implementation, but rather emphasise the areas that need to be improved in order to advance the project from a working prototype to a production-ready product.

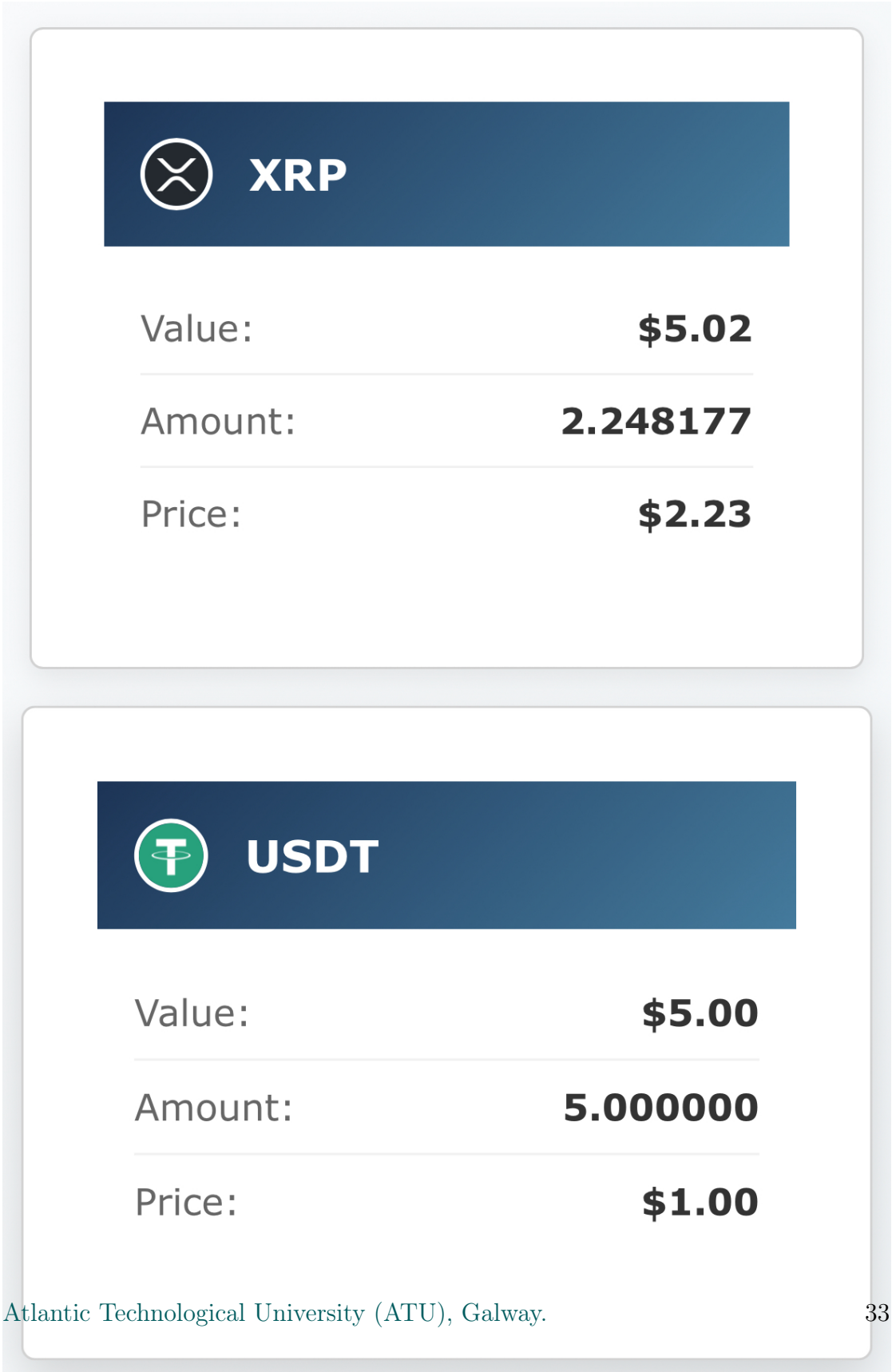

Figure 5.3: Load Testing

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This project required the design and development of a full online application that integrates real-time cryptocurrency portfolio tracking with a social blogging platform. Through a thorough process of research, design, implementation, and testing, the project established that such integration is both practical and successful in enhancing user engagement. The finished solution enables users to monitor their bitcoin assets via an easy-to-use dashboard while actively participating in a community by publishing updates, exchanging insights, and learning from others.

This dual capability is consistent with the initial motivation: cryptocurrency aficionados no longer need to hop between a portfolio app and social media to discuss their assets; instead, they can use a single platform that supports both. Importantly, the study has shown that integrating these domains is doable while preserving performance and security levels. We created a seamless user experience by using current frameworks (React and Django REST) and following best practices. For example, a user may log in, see current portfolio data right once, and then read community posts or ask a question—all without being confused by the design.

From a technical standpoint, the project was valuable in full-stack development. Solved multiple challenges such as authenticating across a distributed system, aggregating data from external APIs, and designing a relational schema for a social network. Each feature built was linked back to concepts discussed in literature: for example, implementing the follow system and observing increased engagement aligns with studies on social networks driving user. The successful deployment on cloud platform further indicates the development and scalability of the system

architecture. Overall, the project meets its objectives and contributes a prototype to the "social fintech" domain – an emerging area where financial and social technologies converge.

Beyond functionality, this project provided insights into user behavior and needs. The enthusiastic feedback from test users – who reported spending more time exploring others' posts after checking their portfolio – supports the claim that combining financial tracking with social features can increase engagement and knowledge sharing, as posited in our introduction and consistent with trends noted in the literature in academic terms, the work adds a case study to the fintech field demonstrating how community elements can be integrated into personal finance tools, and the trade-offs involved (e.g. the need to secure personal data while fostering openness in social content).

This project was developed to gives insights into user behaviour and demands. The positive feedback from test users who reported spending more time investigating others' contributions after findings of their portfolio—backs up our contention that combining financial tracking with social features might improve engagement and information exchange, as we suggested in our introduction. Consistent with academic literature trends, This paper contributes to the fintech industry by providing a case study that demonstrates how community features can be included into personal finance applications, as well as the trade-offs involved.

Of course, there were many problems that had been ran into and lessons to be learnt the hard way. One of the most essential concerns i combining two critical features into a single project required balancing the time requirements for each. Some enchanced features in both areas had to be removed and planned to be done at a later in order to deliver a consistent product on time. This required making realistic judgements, such as focusing on strong, secure API integration above supporting every possible exchange, or implementing basic social media actions Another problem was maintaining consistency throughout the application by matching the data model with the API architecture and UI expectations. Debugging issues highlighted the necessity for thorough work on both back and frontnends

6.1.1 Future Work

Real-time features: Using WebSockets (e.g., with Django Channels or a Node.js socket server) allows for live updates. This might enable live chat or notifications when a tracked user posts, as well as real-time portfolio value updates when market prices fluctuate. Real-time price data streaming (maybe over Binance's

WebSocket) could keep the portfolio display up to date even if the user does not refresh it.

Support for multiple exchanges and wallets: Expanding beyond Binance to other exchanges (Coinbase, Kraken) and even blockchain wallets (via services like Ethereum or Solana APIs) would broaden the user base. Each source has its own API nuances, so a layer for normalizing asset data from different sources would be required. The application could aggregate a user's entire crypto holdings across platforms—a very compelling feature.

6.1.2 Mobile application:

Creating a native mobile app (Android/iOS) or a progressive web app may improve accessibility. A mobile app can send push notifications (for social interactions or pricing warnings) and use device features for security (fingerprint login). Minor changes could be made to the existing REST API to serve a mobile client. Some of the code and design patterns from the online app might be reused using technologies such as React Native.

6.1.3 Research Opportunities:

Opportunities for them future the data thats coming from the platform could prove to one day be useful for research. An instance analyzing the sentiment of the users posts in relation to how the market is currently moving could showcase some insights into how the sentiment around the crypto community affects the price of the coins, another benefit is such a feature could provide very fast and efficient insights to users on their holdings keeping them ahead of the curve on various projects.

Opportunities for them future the data thats coming from the platform could prove to one day be useful for research. An instance analyzing the sentiment of the users posts in relation to how the market is currently moving could showcase some insights into how the sentiment around the crypto community affects the price of the coins, another benefit is such a feature could provide very fast and efficient insights to users on their holdings keeping them ahead of the curve on various projects.

Appendix A

GitHub Repository

Repository: `https://github.com/keemo01/portfolio`
Screencast: `https://youtu.be/P5pLcX_f9Ns?si=LvNli7Myp1QieENt`

Bibliography

[1] What is agile software development? | agile alliance. https://www.agilealliance.org/agile101/. (Accessed on 04/15/2021).

[2] Brett Scott. How can cryptocurrency and blockchain technology play a role in building social and solidarity finance? Technical report, UNRISD working paper, 2016.

[3] Cory Gackenheimer and Cory Gackenheimer. What is react? *Introduction to React*, pages 1–20, 2015.

[4] Jeff Forcier, Paul Bissex, and Wesley J Chun. *Python web development with Django*. Addison-Wesley Professional, 2008.

[5] Jakob Albers, Mihai Cucuringu, Sam Howison, and Alexander Y Shestopaloff. The good, the bad, and latency: Exploratory trading on bybit and binance. *Available at SSRN 4677989*, 2024.