Project Report: Level 4 Software Developer EPA

Jordan McTaggart

Contents

[Introduction 2](#_Toc202172207)

[Planning/ Scope/Feasibility Analysis 3](#_Toc202172208)

[Aims and Objectives 3](#_Toc202172209)

[Scope/Key Performance Indicators (KPIs) and functional specifications 3](#_Toc202172210)

[User Stories 3](#_Toc202172211)

[Feasibility analysis 3](#_Toc202172212)

[Consideration of Legislation, Regulation, and Policies 3](#_Toc202172213)

[Maintaining a secure productive and professional Work Environment 4](#_Toc202172214)

[Constraints and Stakeholders 4](#_Toc202172215)

[Functional Requirements: 4](#_Toc202172216)

[Non-Functional Requirements: 5](#_Toc202172217)

[Research and Findings 5](#_Toc202172218)

[Design 6](#_Toc202172219)

[Development 6](#_Toc202172220)

[Testing 8](#_Toc202172221)

[Deployment 9](#_Toc202172222)

[Maintenance 10](#_Toc202172223)

[Technical Challenges & Solutions 10](#_Toc202172224)

[Project Outcome/Conclusion and Artefact Explanation 12](#_Toc202172225)

[Artefact Explanation 12](#_Toc202172226)

[Conclusion: 13](#_Toc202172227)

[What Went Well: 13](#_Toc202172228)

[Areas for Improvement: 13](#_Toc202172229)

[References 14](#_Toc202172230)

[Appendix 14](#_Toc202172231)

# Introduction

This report documents the development and implementation of a stock market ‘mock’ simulation dashboard designed and built as part of my Level 4 Software Developer apprenticeship. The project was delivered under simulated conditions, due to redundancy from my previous workplace. This project has used each stage of the software development life cycle and has been done end-to-end. I chose software’s, languages, libraries and frameworks that are commercially used to create as professional a project as possible.

The main goal of the project was to enable users to simulate stock market investments using virtual currency, thereby helping them understand trading dynamics in a low-risk educational setting. The dashboard integrates with real-time market data (using finnhub’s API), supports secure user authentication, and enables mock trading, portfolio tracking, and data visualisation. It can be accessed here: <https://stock-dashboard-drab.vercel.app/register> , and the code can be accessed here: <https://github.com/Jordan73737/Stock-Dashboard>

**Technologies used:**

Planning/Ticketing: ClickUp (Agile methodology for incremental/iterative development)

Design: Figma, LucidChart

Frontend: React with Vite build tool, Tailwind CSS, Redux Toolkit, react-router-dom

Backend: Node.js, Express.js, PostgreSQL, axios, pg, cors, bcrypt, jsonwebtoken

APIs and Tools: Finnhub (stock data), SendGrid (email), Docker, GitHub, Vercel, and Render

Testing: Jest, Postman

Throughout this project, I was solely responsible for requirements gathering, design, development, testing, deployment, and maintenance. This allowed me to apply a wide range of skills mapped to the Software Developer standard. However, in a typical software development lifecycle (SDLC), different roles collaborate complete tasks. For example, upper management, stakeholders, and business analysts might carry out the planning, scope and some parts of the feasibility analysis. And you might have designers create wireframes and user journeys. Testers verify that the system meets expectations. Developers can sometimes end up having a hand in feasibility analyses, parts of the design process (like class diagrams, design patterns, and ERD’s), development, testing, deployment, and maintenance. For this project, I took on each role as a full stack developer, with the idea of keeping good commercial protocols like ticketing, version control, security standards etc.

Furthermore, I made sure I had access to tools such as GitHub, Docker, and deployment platforms like Vercel and Render. Externally, I made use of open-source libraries like React and Redux, as well as public APIs like Finnhub. I regularly looked at the apprenticeship KSBs make sure I had a project plan that would cover those off.

# Planning/ Scope/Feasibility Analysis

## Aims and Objectives

The aim was to develop a secure, interactive, and responsive stock investment simulation platform. It’s intended to help users learn how trading works without using real money.

## Scope/Key Performance Indicators (KPIs) and functional specifications

* Register/Login/Logout using secure authentication
* Access protected dashboard content when authenticated
* Save, view and remove their favourite stocks from dashboard table
* Reset their passwords securely using verification email
* Track mock investment value over time via interactive charts and statistics
* Real-time stock search functionality
* State persists and restores after login/logout
* Investment logic works with mock user inputted balance

## User Stories

I gave a questionnaire (See figure 1) to people I knew who were into trading/investing as they are my target audience to gather feedback on what features they would find useful. Like tracking investments, viewing live data, and being able to simulate trades. After building the platform, I gathered feedback from them to improve usability and ensure it met their expectations.

## Feasibility analysis

The project is technically feasible with the tech stack being the frontend in React and hosted on Vercel, while the backend is developed using Node.js and Express.js, hosted on Render. PostgreSQL will be used as the relational database and hosted on Render. Using Finnhub API enables real-time stock data access. These tools are well-documented, supported and within the scope of my technical skillset. The project is also economically viable with my current situation and can be done in the timeframe given.

## Consideration of Legislation, Regulation, and Policies

**GDPR Compliance**: No personal sensitive/financial data was collected. Passwords were encrypted with bcrypt, and user tokens stored in localStorage with expiration logic.

**Data Protection Act 2018**: Email addresses and passwords were never exposed in plaintext. All secrets were managed using .env files and excluded from version control.

**Computer Misuse Act 1990**: Only authenticated users can access investment-related routes. API endpoints were secured using middleware and token validation.

**Security Best Practices**: I have used HTTPS enforced production environments with JWTs signed and verified on every request. I have also got rate limiting and validation implemented on APIs. As well as following commercial best practices such as Git version control, secure API key management, and use of Docker containers for consistent environments.

## Maintaining a secure productive and professional Work Environment

I maintained consistent development progress by using ClickUp ticketing to track workload. I followed a structured plan and committed code in meaningful increments with descriptive Git messages. Making sure to abide by coding standard, comments, neat file structure. As well as using security best practice like .gitignore (See Figure 35), .env files, bcrypt password hashing, JWT-based auth, parametized queries, secure SSL secured endpoints, cors, csrf, token expiration, and only HTTPs connections.

## Constraints and Stakeholders

This is a solo project so not much in the way of stakeholders but there are constraints regarding the collection of stock data from an API. Some are paid for services whilst some are free. I was limited to how much I am willing or able to pay for Finnhubs (in this case) API services/upgrades. I was limited to using the free or low paid tier API rate limits. With Render and Vercel (my choice of hosting platforms), I used the same tiers for services, so I ended up receiving the lower end hosting limits like bandwidth, build time, memory limits. There is also the time constraint of the deadline itself, and of course the resources I could use and expertise/manpower available.

Security and compliance restraints mean that the site must be HTTPS as to not allow for interception and tampering of the un-encrypted data by people. Use of hashing and salting user passwords and other sensitive data, and upon login there should be a JWT token generated. User data like holdings, balance etc must be stored in a secure database endpoint (using authentication, SSL, role-based access, password encryption, Secrets, parametized queries, and firewall rules).

Functional Requirements:  
The stock dashboard must enable users to view real-time stock data, search for specific stocks, and monitor changes in price, percentage change, and buy/sell options. Authenticated users should be able to manage their virtual portfolio by buying/ selling stocks, which then will update their balance and holdings accordingly. The system must track each user’s transactions and display current holdings, including quantity owned and profit/loss per stock. Users should also be able to view a detailed breakdown of their investment value over time through graphs and tables. Additionally, the platform must support persistent authentication, account registration, and features like profile transaction history, and withdraw/deposit functions.

Non-Functional Requirements:  
The application must be responsive and performant, providing a seamless experience across modern browsers and devices. It should maintain high availability and reliability, with the backend hosted on Render and the frontend on Vercel, leveraging appropriate caching and error handling to reduce API rate limits (e.g., from Finnhub). Security is critical; data must be securely transmitted over HTTPS, and user authentication should follow best practices, including token-based login and protected endpoints. The system should scale to handle multiple users concurrently and store historical data efficiently, with regular CRON jobs ensuring scheduled tasks (e.g., calculating daily portfolio value) run reliably. The codebase should be maintainable, well-documented, and follow consistent development practices for ease of future updates.

## Research and Findings

Before development, I conducted technical research to inform decisions such as tools to use as well as frameworks and libraries. I also thought about coding paradigms like procedural, event-driven, and object-oriented programming (OOP). For this project I chose OOP and component-based structure with React for the frontend and Express for the backend. React allowed for reusable UI modules and then Express had a clear routing and middleware pattern for requests. I based my design choices on the functional and technical specifications which required secure authentication, reusable components, API integration, and clear separation of concerns.

**React vs Angular**: I chose React as it is easier for junior developers and has a large community and great documentation.

**Redux Toolkit vs Context API**: Went with Redux Toolkit as its used for cleaner async logic and simplified state slices

**PostgreSQL vs MongoDB**: PostgreSQL selected due to structured data (users, holdings, transactions)

**Security Research**:

* Bcrypts blowfish cipher used for password hashing and salting
* JWTs signed with environment variable secrets
* SendGrid templates used for secure, branded email flows
* This research helped me build a secure, scalable, and user-friendly product.

# Design

I started by making wireframes in Figma to visualize what components should go where on each page and how I’d structure them (See Figures 2, 3 & 4). From these I developed my React component hierarchy to break the UI into manageable/reusable parts like the Navbar, TradeSidebar, ValueHistoryChart). Therefore, the design was a component-based architecture and had separation of concerns. For the data flow and state management I’ve used Redux. But other than Redux’s global state management, data should flow only from parent to child through props (with Redux global states preventing prop drilling). Additionally, I used react-router for routing, making sure only authenticated users could access certain pages by using JWT tokens and CSRF tokens. I used Docker to containerize the frontend, backend and database so that development and deployment is more consistent and reliable across different environments. I can also start the entire stack locally or any other machine quickly using docker-compose up (which reads the docker-compose.yaml file for instructions on how to build each image, network between containers and run them) instead of running separate commands for front and backend.

For my Postgres (pgadmin 4) database I decided to create an entity relationship diagram using the Barker style relationship notation (See Figure 5).

# Development

My code has the following file format (See Figure 6). Main.jsx is the React starting point as it tells react to look for the html element with the ‘root’ ID from index.html and renders the entire app there. The app is wrapped with <Provideer store={store}> tag to connect the entire app to the Redux store. This means any component can now access/update the global state. <React.StrictMode> tags were also used to highlight further problems during development.

App.jsx defines all the main routes/pages using react-router-dom. Checking if a user is authenticated and decides which pages they can access. For example, if the front endpoint /dashboard is accessed it will control the flow to the <Dashboard /> component.

**Backend Node.js and config files**

Essentially, the backend folder holds its own .env file containing - DB URL, JWT secret, Port number, SendGrid API key, the from\_email with my GoDaddy Domain address, and lastly the client URL link to my frontend Vercel hosted site. The backend folder also contains the Dockerfile, package-lock.json, package.json and server.js file. The server.js holds most of the backend API endpoints and brings together the middleware, routes, environment variables, and server startup logic. Without it, the backend wouldn't run/respond properly. It ties into separation of concerns, because actual route logic and database handling are in separate modules. It imports express.js and declares allowed cors origins domains. It initializes the database with initDatabase() - I realize initDatabase() is not optimal because having schema creation inside app logic means there is a lack of version control or migration tracking. It also doesn’t have error handling or multiple environments. If I were to scale this product and have it as a product, I would certainly move the SQL to a separate script file like initDB.sql along with a node script like initDb.js and then use migrations via node-pg-migrate. Continuing, server.js contains auth routes like /api/login, /api/register (which is proceeded by a function to set the users money value to 0 for the valuehistorychart graph) and an authenticate JWT token function.

It then has the get, post and delete endpoints for favourites, and the buy, sell and get endpoints for holdings. Popular stocks table logic is done with a get (for the top 10 most popular stocks). Following that is the balance and holdings logic to set, get, buy, sell and update the holdings and balance. Then the daily value calculations done by recordDailyUserValues() does just that, it stores the amount of money in the account using the available balance, and money in each of the stocks, to calculate this, and then plots it on the table. The value recalculates with recordSnapshotForUser() which reruns every day via api/run-daily-snapshot and api/user-history endpoints. (See Figure 29, 30 & 31)

Finally, server.js handles password reset and server startup.

The backend has the auth.js file which is in backend/routes/auth.js, unlike server.js which is in the root of the backend folder. Auth.js handles sendgrid password reset (using the sgMail library), including the construction of the password reset URL, and builds the email message to be sent and generates the JWT token. Inside backend/db/users.js has functions to get user by email from the database using the Pool class from the pg library. It can also update user passwords with the bcrypt library again that uses a blowfish cipher.

**React Components**

Moving onto frontend, all the react component jsx files inside of src/components (See Figure 7). Each file is a piece of UI logic. For example, Navbar.jsx is at the top every page after you’ve logged in and handles links based on login status using localStorage. TradeSidebar.jsx is one of the more interactive components since it slides out to allow the user to buy or sell stocks. It uses the useState hook and Redux selectors to pull in current balance and holdings, and dispatch to trigger actions like buying or selling stocks (as well as useEffect).

ValueHistoryChart.jsx renders a line graph of the user’s portfolio value history using the Recharts library. Other components include Profile.jsx, which shows a user’s current holdings, their balance, portfolio value chart and allows deposits or withdrawals. It also reuses TradeSidebar (like home.jsx does) to keep the layout consistent.

The components work together in a hierarchy managed by App.jsx, that handles routing using react-router-dom. I’ve written the components in a way that minimizes prop drilling by using Redux for global state where needed (such as balance, holdings, and auth). Redux slices havve asynchronous logic (e.g., fetching or updating data) using thunks and reducers to update state in a predictable way.

Each component uses React Hooks like useState, useEffect, and Redux’s useSelector/useDispatch. Styling is handled using Tailwind CSS classes inside the JSX.

**Redux Store and Slices**

The slices folder contains authSlice.js and stockSlice.js. The authSlice.js manages login state and JWT tokens so that components can respond to whether a user is logged in or not. The stockSlice.js handles the user’s portfolio – including balance, stock holdings, favourites, and portfolio history. Using Redux thunks allowed me to do async API calls (like fetching holdings or updating the balance), while reducers update. This made the app more maintainable and prevented repetitive prop passing (prop drilling) between deeply nested components.

Store.js is outside the slices folder but still inside the src folder. It imports the reducers from the slices and then Redux uses this store.js file to figure out which slice/reducers should handle certain actions.

# Testing

Throughout this project I utilized console.log/.warn/.error (see Figure 34), breakpoints, and test-driven debugging for many different types of errors – here are the major types of testing I’ve used.

**Smoke Testing**

I was doing constant smoke tests during development to make sure each new feature didn’t break anything major. This included running the frontend and backend locally after changes to check that login, stock search, buy/sell, and dashboard rendering all worked as expected.

**Regression Testing**

I performed manual regression tests after making changes that would have a cascading impact on different components, functions etc. For example, when changing balance update logic I had to make sure that buying, selling via tradesidebar.jsx worked via profile.jsx or home.jsx. And that balance changes that were from deposit and withdraw were not impact by this.

**End-to-End Testing**

While I didn’t use automated E2E tools like Cypress, I manually tested full the system from front to back. For example, for the register process, I made sure it registered users in the database correctly, that it logged you in automatically after this, and you could carry out primary functionality like depositing, buying, selling, refreshing, withdrawing all worked from front to back. Including the reset password functionality.

**Unit Testing**

I unit tested handleAddStock inside Dashboard.jsx, which adds a stock to the dashboard and favourites. The test checks that duplicate stocks can’t be added, and that correct Redux actions are dispatched when a new stock is added successfully (See Figure 8). To simulate failure I used Jest to mock the axios post request and forced it to reject with an error. And then I confirmed the error was correctly caught and that the alert function was triggered with the appropriate error message. I also verified that no Redux actions were dispatched in case of failing. Basically making sure invalid state changes were avoided. These tests helped validate and debug both the positive and negative paths for stock addition.

**Postman API Testing**

I used **Postman** to test my backend endpoints separately from the frontend. This helped confirm that the APIs worked and returned correct responses. One example was testing the route for recording the daily value of a user’s holdings (/api/run-daily-snapshot).

This endpoint runs a function called recordDailyUserValues() that calculates the total portfolio value for each user and stores it in the database. I sent a request in Postman and confirmed it returned (See Figure 9).

# Deployment

For deployment, I hosted the frontend on Vercel and the backend (including the PostgreSQL database) on Render. This is to take advantage of their platform-specific strengths. As Vercel is best for frontend frameworks like React and has fast global CDN, automatic static optimization, and Git integration. Meanwhile, Render is better suited for backend services, Separating the two allows for more clear separation of concerns and independent scaling.

So, I push my code to GitHub, and both Vercel and Render pulled the code directly from my repository whenever I pushed changes. Vercel handled the React frontend by building it from my package.json and automatically serving the compiled files. However, for Render, I created separate web services for the backend and database. And even if I deployed to github the Render logic would need redeployment to see changes. I configured environment variables (like JWT secrets, database URLs, and API keys) in Render (See Figure 10) . I made sure to include a vercel.json file to handle client-side routing in production. That was essentially my deployment pipeline which worked well for what I needed.

# Maintenance

I have produced some documentation for the project (See Figure 11 and Figure 32) and the ticketing system is accessible via ClickUp (See Figure 12). The code also has appropriately naming conventions for variables etc. Along with comments throughout all of the files to guide potential handovers through the logic.

## Technical Challenges & Solutions

**CORS Setup**

I ran into a couple of cors issues along the way because needed my frontend to securely communicate with the backend. CORS is basically a security protocol that’s configured in the backend (with cors headers like access-control-allow-origin) but is enforced in the browser - to make sure that the domains, ports and protocols sending requests are allowed and valid to send their POST requests through to the backend (CORS preflight checks do this after request is sent). If for example the browser sends a request from an un-allowed origin domain then the browser console will show a CORS error message. If the backend response doesn’t have the right cors headers, then the browser will block access to the response.

**Docker Containerization Setup**

I had to write a Dockerfile in the backend and the root project folder to define how the app should be built and run. The frontend had a first stage that installed dependencies and then built the files using Vite, and the second stage served packages to host the compiled files. The backend Dockerfile installs dependencies and starts the Express server. I then had to create a docker-compose.yaml file at the root level to define and manage multiple services: the frontend, the backend, and the postgres database. In this file, I specified the ports and loaded the .env variables. I also used depends\_on to make sure the backend waited for the database to be ready. Finally, I ran the entire app stack locally using the docker-compose up --build command, to build all the images and start up the containers to run everything in a consistent/reliable isolated environment (See Figure 13).

**Deployment Pipelines and Key Files – VS Code + Postgres - Git/Github – Vercel – Render**

I had to make sure environment variables like the API base URL and database credentials were set up properly in both platforms. For frontend I used the VITE\_ in front of both the Finnhub api key and api base url variable.

And for backend I had to make sure Renders .env variables that matched up with my .env file variables. Using Database Url, JWT secret, Port number, sendgrid api key, from\_email, and client url link to my frontend Vercel hosted site (See Figure 14).

I also had to make sure I committed and pushed important files like package.json and package-lock.json as the former tells the hosting site what packages to install. And the latter ensures that Render installs the particular versions of each package (this is important as major package upgrades may break a server). Package.json is more of a guideline of what must be installed, whereas package-lock.json forces the versions of those packages you want to be installed. On this topic, I also had to make sure to create .gitignore to avoid committing and pushing sensitive data in .env files, or huge files like node modules folder to github. Similarly .dockerignore does this same thing for the docker containers, but docker also has docker-compose.yml file which basically handles the starting and connection of my frontend, backend and database and controls what order they start in. The Dockerfile does the image building for the project which has two stages, build and production. For the build stage it just uses node.js to install dependencies and build the frontend with Vite. And for production stage it just serves a more lightweight image of the already built files to the host site, whilst exposing the 3000 port to let you visit it in the browser.   
Other important files were Vercel.json, tailwind.config.js, postcss.config.js and vite.config.js. Vercel.json basically tells Tailwind.config.js how to handle the routing, tailwind config customizes the CSS settings like which files to scan for the class names. Postcss config sets up postcss plugins to process CSS automatically during build time. And Vite config configures how the Vite build tool is run.

**Finnhub API Rate Limit (See Figure 15 & 16)**

I had an issue were triggering too many rapid requests to Finnhub via page refreshes or just navigating back and forth from profile to home pages caused my rate limit to reach its limit and give me a 429 error. This was due to the Popular stocks table fetching the stock data every time the page refreshed. And the profile page would fetch the holdings stock data every time it refreshed as well. To resolve this, I used functionality to cache the stored values and only trigger a refresh if it has been more than 2 minutes since data was last cached. I also filtered out unsupported symbols (e.g. those containing dots like "BRK.B") before making requests, as these would often fail or waste an API call. These both helped me stay under Finnhubs free membership rate limits.

**Finnhub API Rate Limit Preventing Live Stock Polling/Updates (For favourites table)**

I wanted to create the illusion of real-time stock updates on the dashboard, so I implemented a simulation using Math.random() to generate small fluctuations in stock prices. While this added a sense of interactivity and visual interest, I now recognise that it's misleading since the changes do not reflect actual market data. This approach was chosen because of Finnhub restricting free users from accessing live real-time requests. In a production environment, I would instead use a websocket connection or server-side polling to stream or periodically update genuine price data.

**Preventing unnecessary re-renders (See Figure 17)**

For my tradesidebar.jsx I have various calculation functions for raw quantity, rounded quantity, maxslidervalue, and sliderstep. To do this I had to implement the useMemo hook to cache the result of these calculations unless its dependency changed. As without this, the app would slow down and continue to cause unnecessary re-renders .

# Project Outcome/Conclusion and Artefact Explanation

## Artefact Explanation

**Supporting Documentation:** Maintenance document, Entity Relationship Diagrams (ERD), Figma wireframes, and user questionnaire.

**Frontend:** A responsive React-based UI (See Figure 18) featuring authentication forms, interactive dashboard, charts, real-time stock search functionality, and stock search into favourites functionality (See Figure 33).

**Backend:** An Express-based REST API secured with JWT and bcrypt (See Figure 19 & 20) , handling user auth, portfolio logic, and stock data.

**Database:** PostgreSQL schema with tables for users (See Figure 21), holdings, favourites, and token handling. Includes .env config for environment security and bcrypt password hashing (See Figure 22).

**Deployment Setup:** Production deployment to Vercel (frontend) and Render (backend) using Docker where applicable.

**SendGrid Setup:** (See Figure 23) with official domain to send the reset email (See Figure 24).

**Version Control:** GitHub repo (See Figure 25 & 26) with meaningful commit messages, organized feature branches, and reviewed PRs for all major milestones.

**Links:**

<https://stock-dashboard-drab.vercel.app/register>

<https://github.com/Jordan73737/Stock-Dashboard>

**KPIs Met**

* Register/Login/Logout using secure authentication
* Access protected dashboard content when authenticated
* Can search for any stock and add (or remove it) from your favourites
* Can add a popular stock to your favourites
* Reset their passwords securely using verification email
* Track mock investment value over time via interactive charts and statistics
* State persists and restores after login/logout
* Investment logic works with mock user inputted balance
* Deposit and withdraw functionality with balance updating

## Conclusion:

This project provided hands-on experience across the full stack. The finished product achieved all KPIs, reflected SDLC best practices and covered off all the KSB’s.

## What Went Well:

* Frontend and backend integrated smoothly and had docker containers as well
* Hosting Vercel and both Render hosted logic and database works (See figure 27)
* Project components are separated clearly and most code is commented where necessary for potential colleague handovers
* Password reset functionality and email works quickly and easily
* Tailwind allowed fast and easy styling
* Deployment is reliable and used environment-specific configurations.

I successfully used JavaScript array methods such as .map(), .filter() and .sort() to render and manage stock data based on user interaction. In the backend, user authentication was successfully in storing hashed and salted passwords with bcrypt, which was then used in comparison functions to validate credentials securely. Generating JWT tokens conditionally using a header, a payload, and a signature. The header and payload are Base64Url-encoded JSON objects containing metadata and claims (such as user ID and expiry), while the signature is generated by hashing the encoded header and payload with a secret key using a specified algorithm usually HMAC SHA256.

## Areas for Improvement:

My ClickUp tickets still have many small improvements that would polish the site (like a sell all button), as well as larger features that would drastically level it up (like clicking a stock row and getting a pop-up candlestick chart). My ClickUp also contains small bug fixes like changing when you purchase your entire balance worth of stock you can be left with -0.01 due to a rounding error.

O-AUTH would be a great security measure as it would integrate with platforms like Microsoft and Google for example to have them verify on their side which increases the security as these mega corporations have more thorough methods of verification. It would help my platform appear as more reliable and up to date which gives users more confidence and trust.

A feature for deleting your account and its associated data is certainly something I will implement as soon as possible as I feel this is important even for a small application.

I could convert the project into a typescript syntax for better type safety, linting, and overall maintenance. In future if I wanted faster loading of static pages and dynamic pages alike, I could implement the usage of static site generation (SSG) and server-side rendering (SSR) methods respectively.

Ideally in hindsight I would have used clearer branching strategy for bug fixes vs features.

# References

* <https://react.dev/>
* <https://render.com/docs/render-vs-vercel-comparison>
* <https://tailwindcss.com/>
* <https://nodejs.org/>
* <https://www.postgresql.org/>
* https://finnhub.io/docs
* <https://sendgrid.com/>

# Appendix

See attached document titled – Mapping Document & Appendix