MetFinance

Sanna Jammeh - s354397
Alime Deniz Ølcek - 354399
Anders Håkonsen - s354375
Rebekka Svalland - s354345

Table of contents

Introduction

We've taken on the job of making our MVP application from the first assignment bigger and better. The app created is a fully functional and responsive crypto trading app with *JWT* authentication, simulated trades, a real-time chart using Tradingview, and close to real-time data and exchange rates using the CoinMarketCap API. It features a web client written in React 18 using Node version 18 as the runtime and a backend (***API***) written in Net Core 6.

Requirement Specification

Functional requirements

Prior to rewriting the MVP, we laid out the following functional requirements:

1. Authentication
    a. The user can create an account
    b. The user can login to their account
2. Facilitate a simulated virtual wallet
3. Facilitate purchases and sales of cryptocurrency positions
    a. The user can buy a position of a cryptocurrency
    b. The user can sell a currently held position of a cryptocurrency
4. Display currently held positions

Non-functional requirements

Prior to rewriting the MVP, we took time to consider these non-functional requirements:

1. Extensibility
    a. The frontend application should be easily expanded in future developments
    b. The backend service can be easily expanded to accommodate more functionality required by the frontend
2. Security
    a. All data traffic between the client and the API is through **HTTPS**
    b. Users should securely authenticated using stateless methods (**JWT**)
    c. Users should automatically be signed out after a set interval
3. Robustness
    a. Both the client and API should handle errors without impacting the user interface

System design & Architecture

Technologies

The application builds upon the modern approach used in the MVP. As a result of our decisions in developing the MVP, rewriting the application's front-end using React proved trivial.

Client side technologies

Vite
Vite is a modern javascript build tool using ES Modules to provide lightning-fast HMR (updating the client without refreshing) in development. This allowed us to iterate much faster and see the results of the code within milliseconds. Additionally, we extended the Vite bundler with SWC (a Javascript transpiler written in Rust) to improve the production build times of the project.

React
React is a UI library developed by Facebook that allows developers to create highly interactive user interfaces that depend on frequently changing data. This is the main driver of our client-side app; all user interface elements are written in JSX and rendered by React.

Tailwind CSS & Sass
Tailwind CSS is a CSS framework that compiles CSS utility classes to native CSS code ahead of time. This allows for iteration when designing the user interface. We paired Tailwind with Sass (a CSS superset and preprocessor) to improve the developer experience when styling the app.

Tw-Classed
Tw-Classed is a public React library built by one of the group members (Sanna) to create reusable React components that integrate directly with Tailwind CSS classes and provide automatic TypeScript intellisense. Using this library, we were able to produce a small design system to improve development speed and ensure a consistent user interface across many pages and components.

SWR
a React fetching library made by Vercel to simplify making requests using React Hooks. It implements a global cache, which can be updated ahead of time or re-fetched programmatically. Additionally, it incorporates the “Stale While Revalidate” cache technique to decide when to render a component (Posnick, 2019).

React Hook Form
a react library providing hooks to control form and input events. The library handles custom input validation as well as regular input validation with custom error messages. All inputs currently use this for validation and form submission.

Radix Colors
Radix Colors acts as our color system and consists of handcrafted colors with automatic dark mode support. built by WorkOS.

Other dependencies
1. CSS.GG - A CSS based Icon library
2. React-Icons - A React based icon library
3. Notyf - A tiny library to provide small toast messages
4. Valtio - A global state management solution for React built using ES Proxies to handle reactivity
5. react-tradingview-embed - A React library which integers Tradingview's embeddable chart into React components
6. ofetch - Minimal native Fetch API wrapper to simplify sending requests.
7. React Router dom - SPA routing library

API Technologies

.Net Core 6
.Net Core is a cross-platform framework that provides the core functionality needed to create robust APIs and web applications. It includes all the core functionality needed to build a web app. In addition to providing newer functionality, we primarily decided to use version 6 due to incompatibilities with many of the group members' M1 Macbooks.

SQLite
MetFinance makes use of SQLite and does not host a database separately. SQLite is an SQL-based database capable of persisting all its data into a single file. This makes debugging very simple, in addition to being the perfect database for a system that is not distributed across multiple servers.

Entity Framework Core 6
Entity Framework provides the primary database functionality needed for MetFinance to function. Entity Framework is an object-relational mapper, or ORM. It translates code into SQL queries that a database like SQLite can process.

Other packages
- Newtonsoft.Json - A library to serialize and deserialize JSON data
- BCrypt.Net - A library providing the popular bcrypt hashing algorithm (used to save a hashed password in db)
- DotNetEnv - A library to load .env files (commonly used with api keys and secrets)

Data models

Our MVP already had the concept of *“users”* built in (without authentication). So, we only had to make a few small changes to our database schema to make it work with a system that uses email and a password to log in. The Users table’s “**name**” column was renamed to "**username**,” and “**email**” and “**password**” columns were added to the Users table. Additionally, “**createdAt**” and “**updatedAt**” columns were added to the Transactions table to improve ordering.

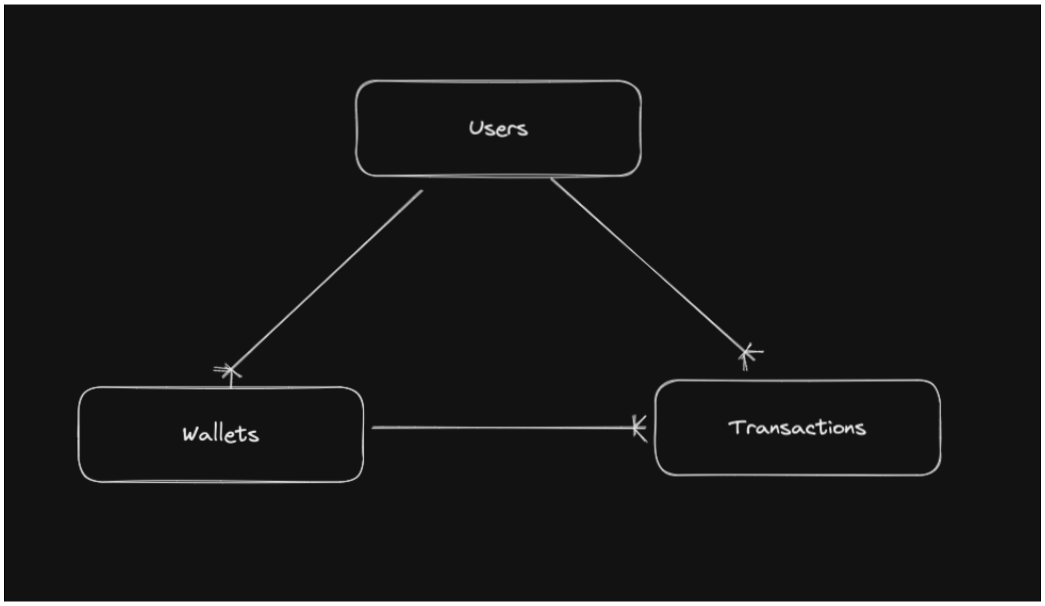


Figure 1: Relationship diagram

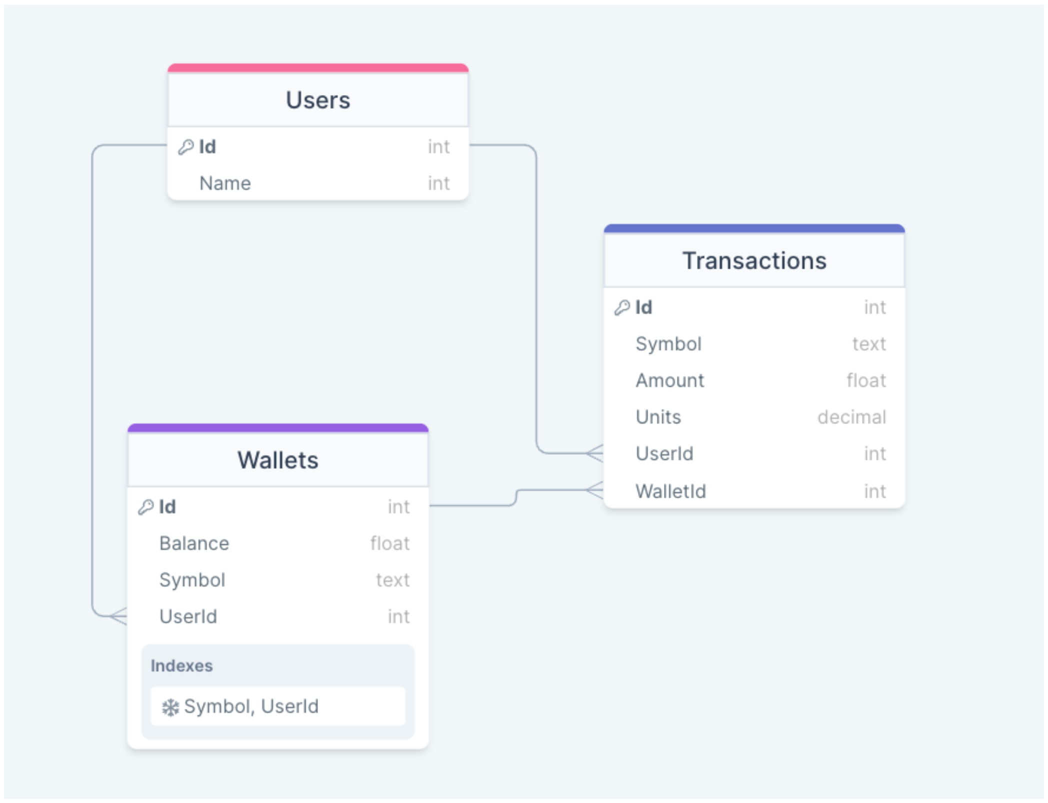


Figure 2: Original Database Schema of MVP

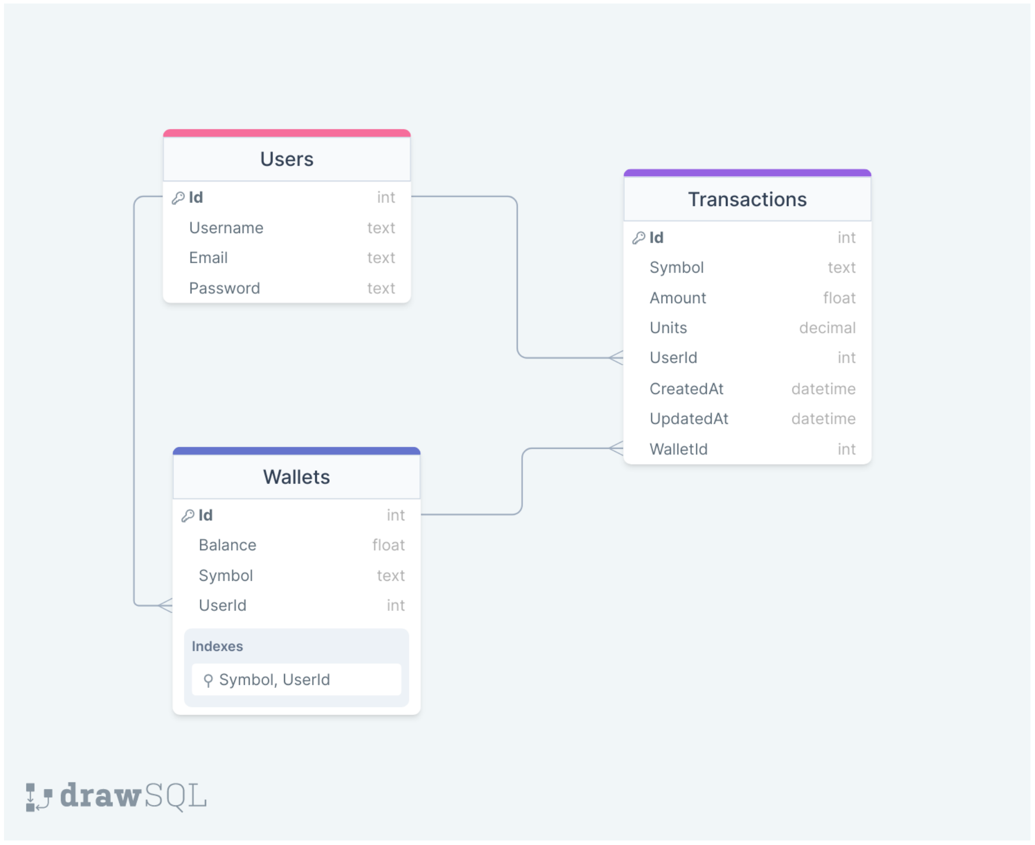


Figure 3: Final database schema for MetFinance

API Architecture

We decided to use the service pattern while developing the API. In the startup code, the services are dependency-injected into the controllers using .Net Core's Dependency Injection container.

1. Controller
    a. Validates client input: DTOs (data transfer objects) & route parameters
    b. Validates authentication and gets User
    c. Calls Service
    d. Responds to client with data from Service
2. Service
    a. Assumes inputs are valid
    b. Performs business logic (ex. Queries database, generates JWT Token, Requests external API)
    c. Returns data result of executed business logic
3. Data layer (***DataContext*** - Entity Framework Core)
    a. Performs SQL query
    b. Returns data result of SQL query

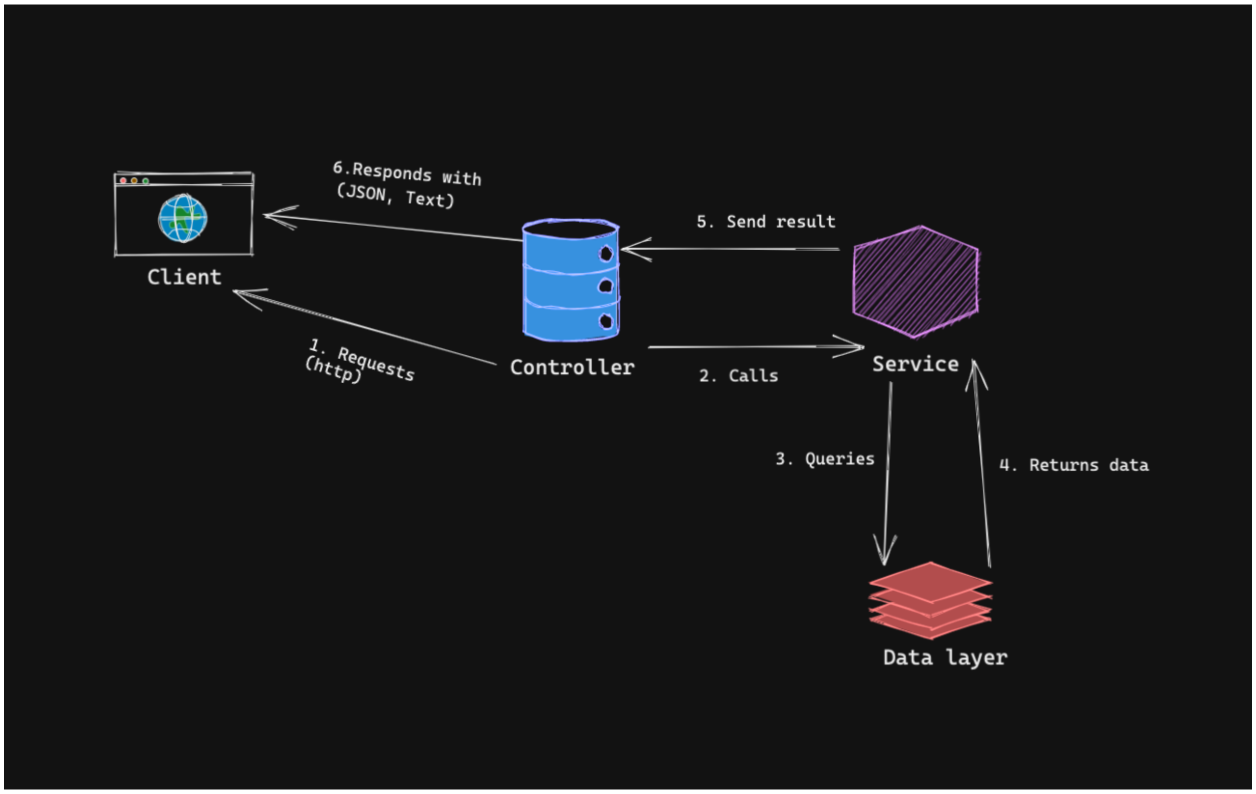


Figure 4: Client to Server request flow

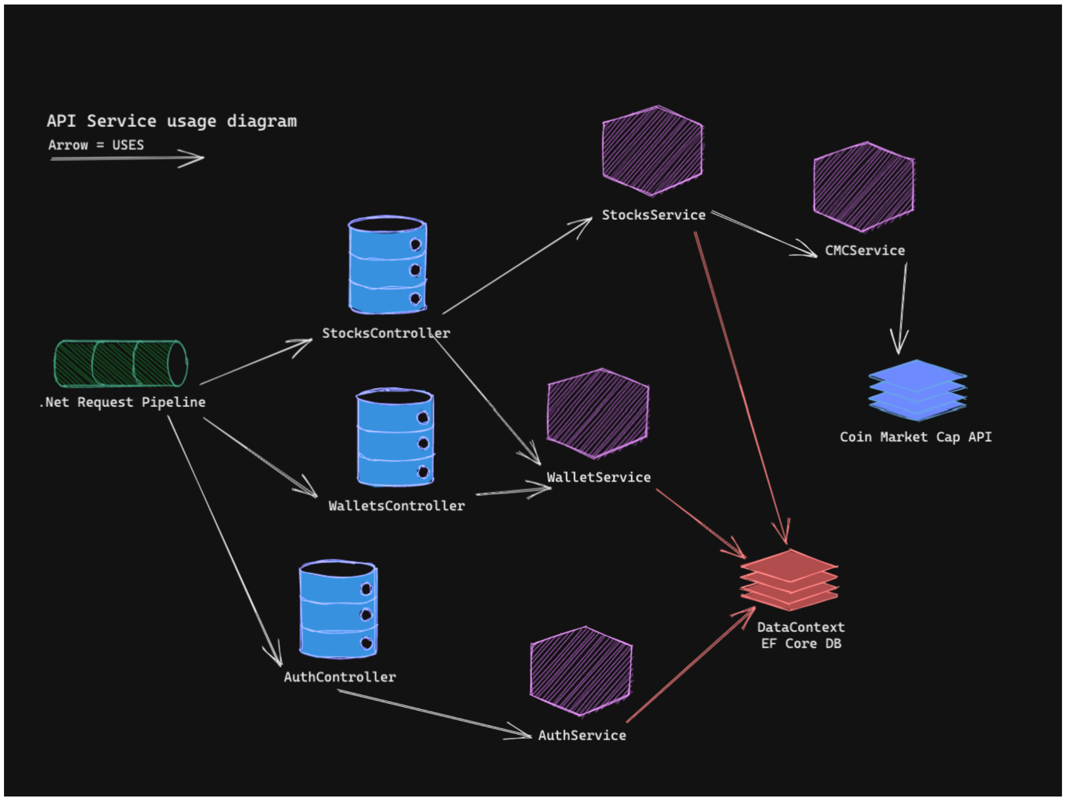


Figure 5: API service dependency diagram

Building the application

Authentication

Authentication was implemented through the use of JWTs (JSON Web Tokens). JSON Web Tokens are a way to keep the entire authentication process completely stateless. It has become the most frequently used authentication method due to its versatility. A JSON web token consists of a "***header***", "***payload***" and a "***signature***" **(auth0, n.d.).** The server is responsible for generating the JWT token; it usually includes a way to identify the user when the token is consumed at a later point. When the token is generated, its signature field is encrypted using a ***"secret key"*** provided by the API authors. The token is then sent back to the client, and the client is responsible for managing the token state. When the user requests a protected endpoint, it must provide the JWT token in the HTTP header. All the server needs to do to validate if the request is valid, is to verify that the ***"signature"*** matches the "***secret key***" used to create it. This ensures the authentication process is completely stateless, as the server no longer needs to keep track of any sessions.

We implemented JWTs using "Microsoft.AspNetCore.Authentication.JwtBearer." A custom AuthController was created to handle user authentication and issue JWT tokens. We then added the validation logic needed inside the startup script and bound the “***User entity***” to .Net’s “***HttpContext***” if the token is valid. The Microsoft.AspNetCore.Authentication package provides middleware that handles the business logic for validating the token on protected controllers and endpoints.

Crud implementation

MetFinance implements CRUD (create, read, update and delete) on the **“Transaction”** entity. The service “***StockService***”, which is called by the “***StocksController***” is responsible for providing this functionality.

Create

StockService.CreateTransaction provides the create functionality required. It is responsible for creating a ***Transaction*** entity, getting the exchange rate for the cryptocurrency using the CMCService and populating the Wallet balances based on the transaction result.

Read

The StockService provides multiple read methods related to transactions. These include:

1. GetTransactionsByUser -> Gets all transactions with a given userId
2. GetTransactionById -> Gets a transaction with a given id
3. GetTransactionByUser -> Gets a transaction with a given id and userId

Update

When a position is purchased, it can be updated, however, only the units (amount of cryptocurrency) can be modified. The UpdateTransactionUnits method changes a transaction based on the new unit amount it gets. It will call the CMCService to get the difference between the “***Transaction***” entity’s current units and the incoming units. Based on a negative or positive unit difference, it will buy or sell the difference (updating the appropriate USD and crypto ***“Wallet”*** entities). Finally, it will update the transaction with the correct units and USD value.

Delete

Deletion occurs when a position is sold in its entirety. When a call to “***DeleteTransaction***” is made, the service will call CMCService to receive the latest USD exchange rate and sell off the entire position (updating the appropriate USD and crypto “***Wallet***” entities). Finally, it will delete the transaction from the database and commit the changes.

Additional functionality

In order to optimize the application, we added some additional functionality to the frontend application.

Responsivity: The client app is fully responsive down to the display size of an iPhone 12 Pro.
"A functional Tradingview" was added to the main section of the screen and will display the current real time chart of the selected currency compared to USD. This was achieved using the Tradingview API and React library called "react-tradingview-embed".

Exception handling and Logging

Local exception handling

All controllers in the app use local exception handlers if any of their services' methods are expected to throw an exception. Services will throw a user defined exception (Microsoft, 2022) if there's a user based reason for it not succeeding in performing their business logic. The exception and its message will be caught by the controllers, which will then send the right HTTP response. If a service's method does not explicitly throw an error, no local exception will be present in the controller's method. If these methods throw an exception, the "Global exception handler" middleware will catch it.

Global exception handling

Any uncaught exception will be caught by the "***ExceptionFilter***" middleware. This middleware is responsible for mapping the uncaught exception type to appropriate HTTP responses and serializing the error to JSON. Additionally, logging is performed here. As these errors are unexpected exceptions (not programmatically thrown), they will be printed to the console using the Logger service.
By using a mix of global and local exception filters, we are able to differentiate between critical errors and explicit errors and respond accordingly in a more centralized way.

Testing

Testing is implemented using "***Moq.EntityFrameworkCore***" and "***xUnit***". All controllers are fully tested with both failure and success cases. Mocked services either return the right data or throw explicit errors to test how exceptions are handled locally. Also, some controllers' HttpContexts will be mocked so that the "User" entity can be used to fake a valid JWT token.

Frontend user interface

When the user first visits the page, they will be redirected to “/login” and be shown the login UI. From this UI, they can decide if they’d like to register a new account or login to an existing one.

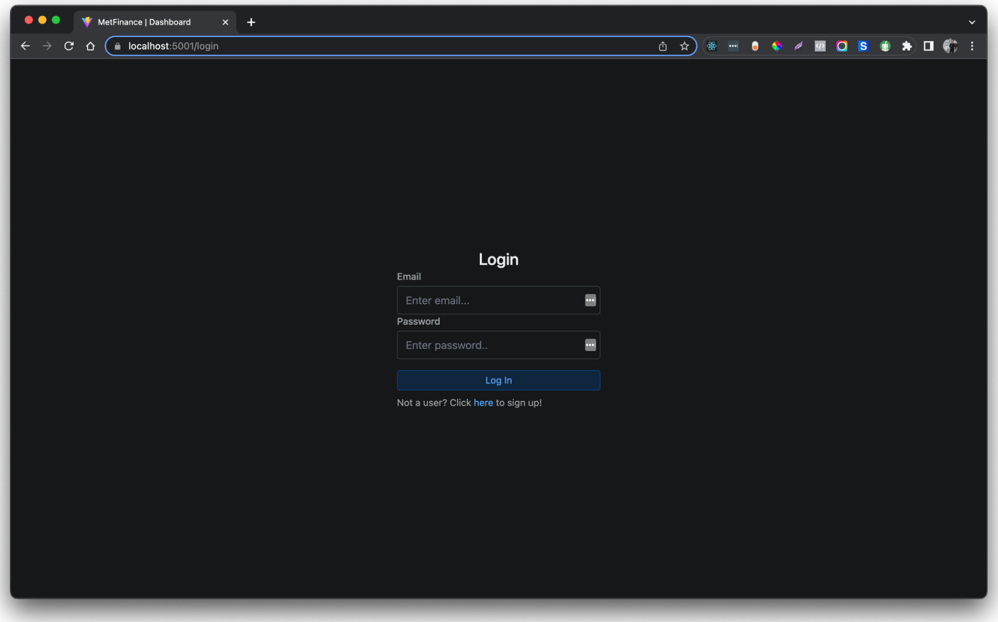


Figure 6: /login - Login Mode

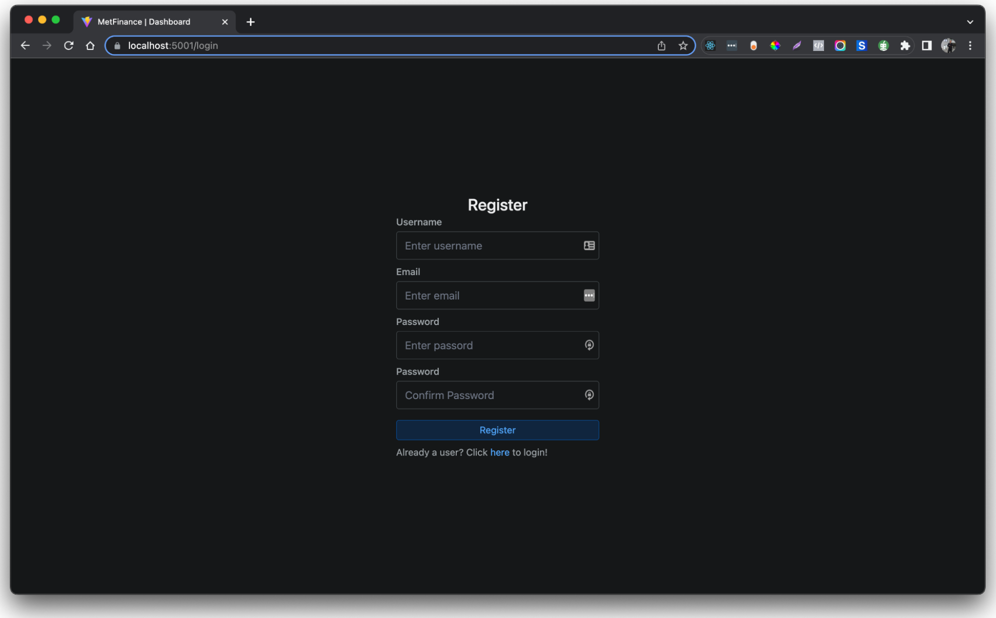
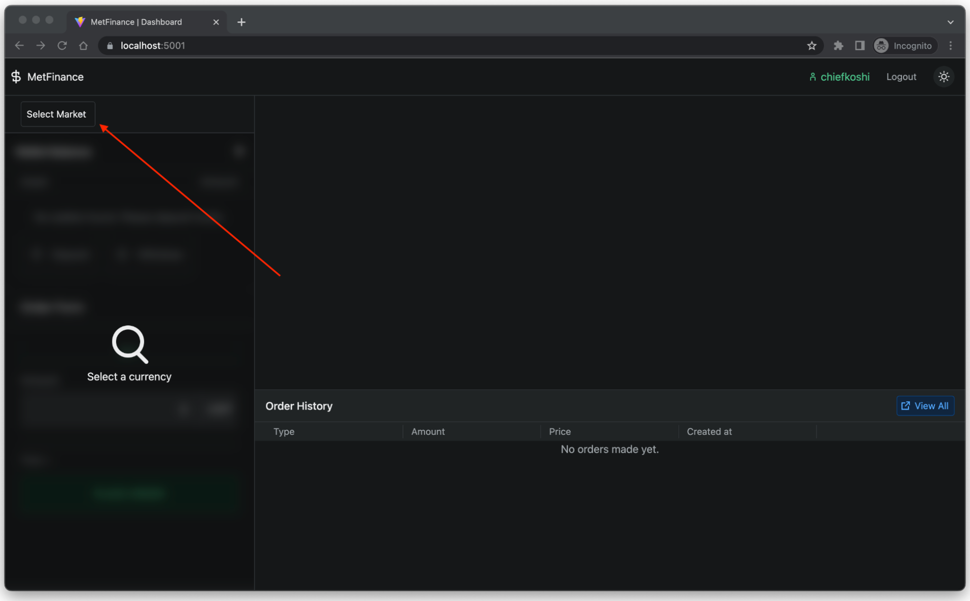


Figure 7: /login - Register Mode

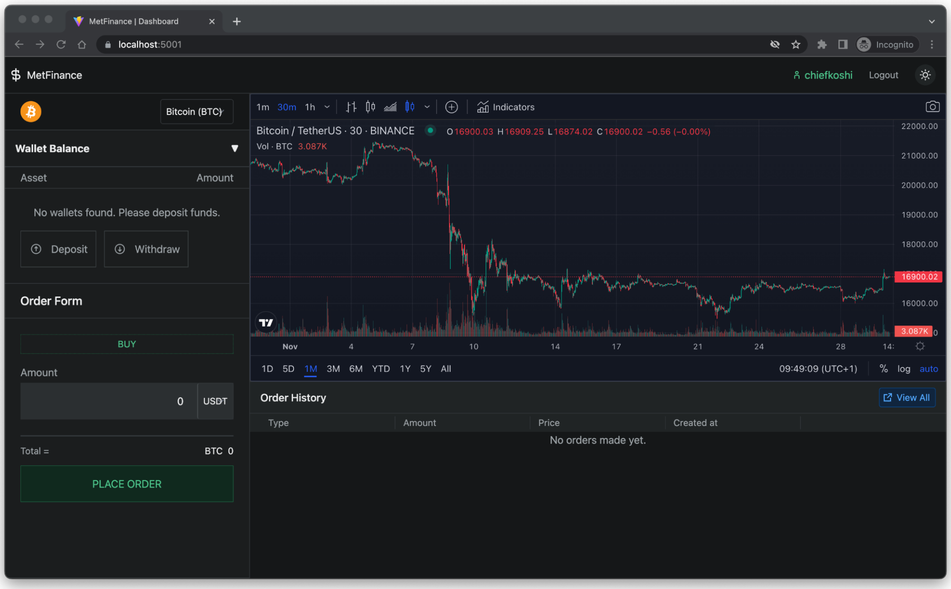
Once the user has successfully logged in or registered they will be redirected to the Dashboard at ***"/".*** If it's the user's first visit to the application, they will be asked to select a market (cryptocurrency) by pressing the "***Select market***" button. Once a currency is selected, the full dashboard will display.



Type	Amount	Price	Created at

No orders made yet.

Figure 8: / - Dashboard page - no currency selected



Asset	Amount

No wallets found. Please deposit funds.

Amount: 0 USDT

Total = BTC 0

Type	Amount	Price	Created at

No orders made yet.

Figure 9: / - Dashboard page - currency selected

By pressing either deposit or withdraw, a popup will show where the user can enter the amount they'd like to deposit or withdraw. Purchases of crypto are positions; therefore, the user cannot deposit or withdraw any cryptocurrencies, only USD.

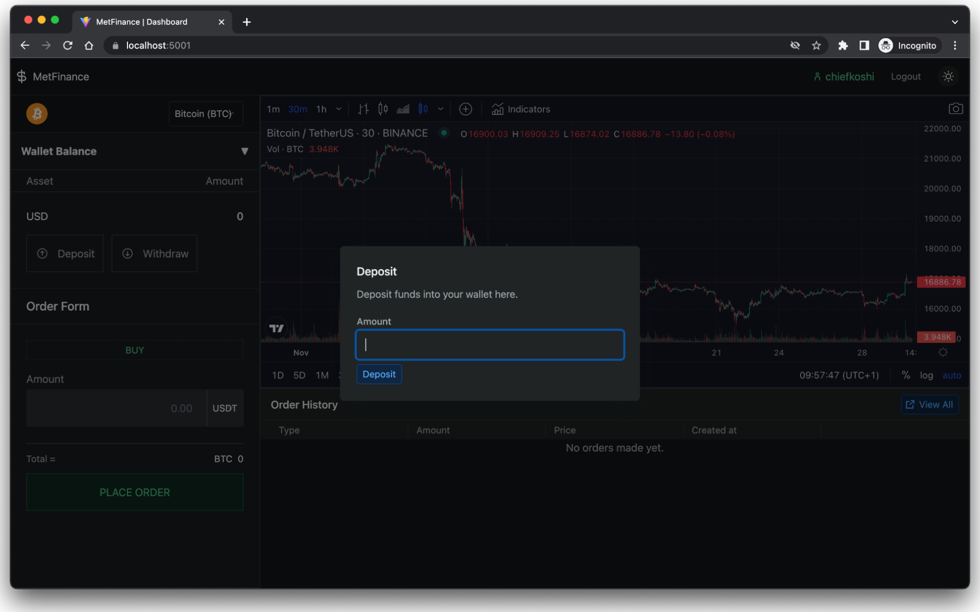


Figure 10: Dashboard page - Deposit or withdrawal popup

To purchase a cryptocurrency, the user must enter an amount. By pressing the currency symbol in the input, they can choose whether to enter a number in USDT or BTC.

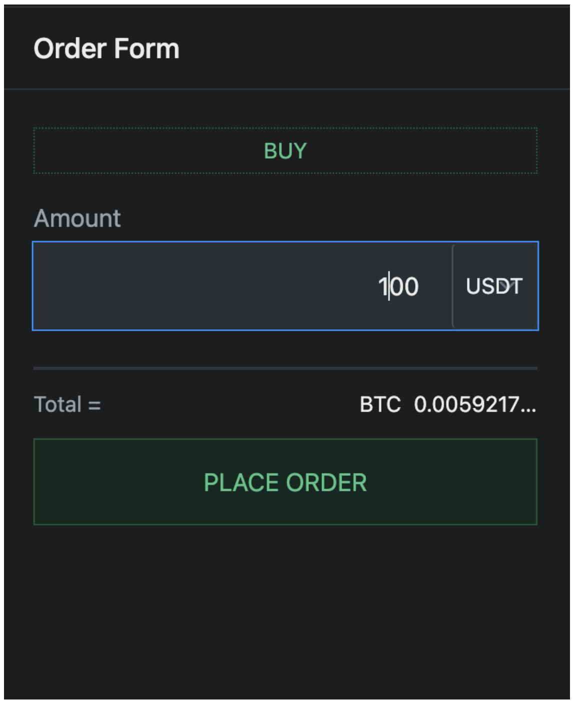
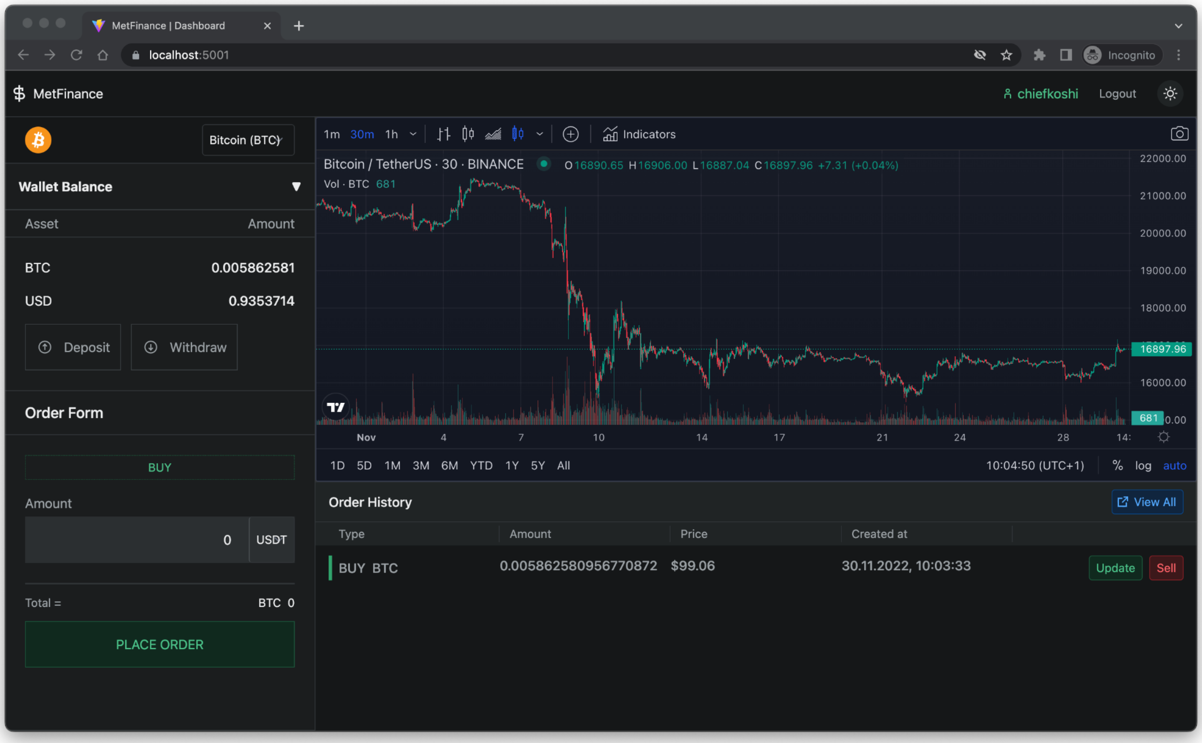


Figure 11: Order Form

When a position has been created, it will display in the “Order History” section. Additionally, the wallet balances will be automatically updated. To update the position (either sell or buy more units), the user must press the “Update” button on the position item. If the user wishes to sell the entire position, they must press the sell button. This will immediately sell off the position and update the wallet balances.



Wallet Balance

Asset	Amount
BTC	0.005862581
USD	0.9353714

Order Form

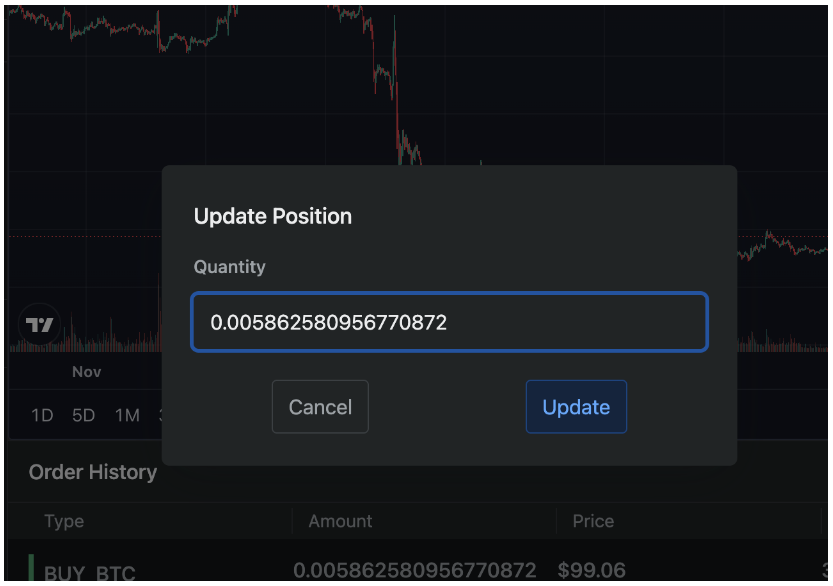
Amount 0 USDT

Total = BTC 0

Order History

Type	Amount	Price	Created at
BUY BTC	0.005862580956770872	$99.06	30.11.2022, 10:03:33

Figure 12: Dashboard with purchased currency



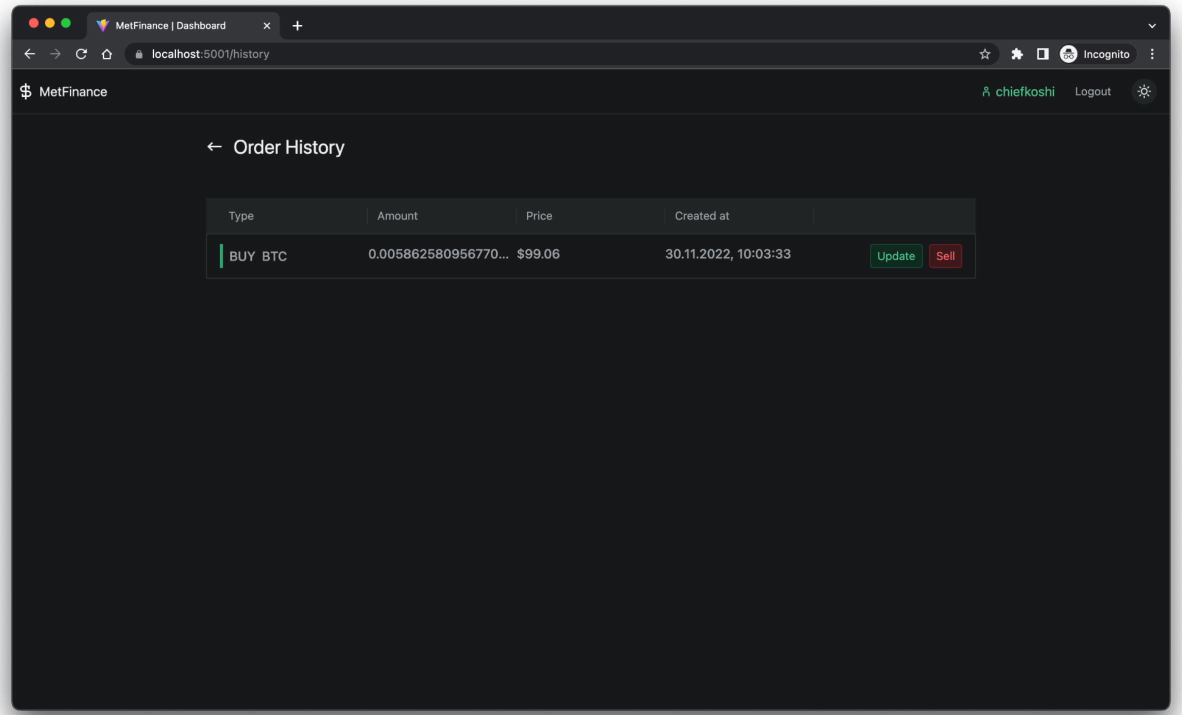
Update Position

Quantity

0.005862580956770872

Figure 13: Update Position Popup

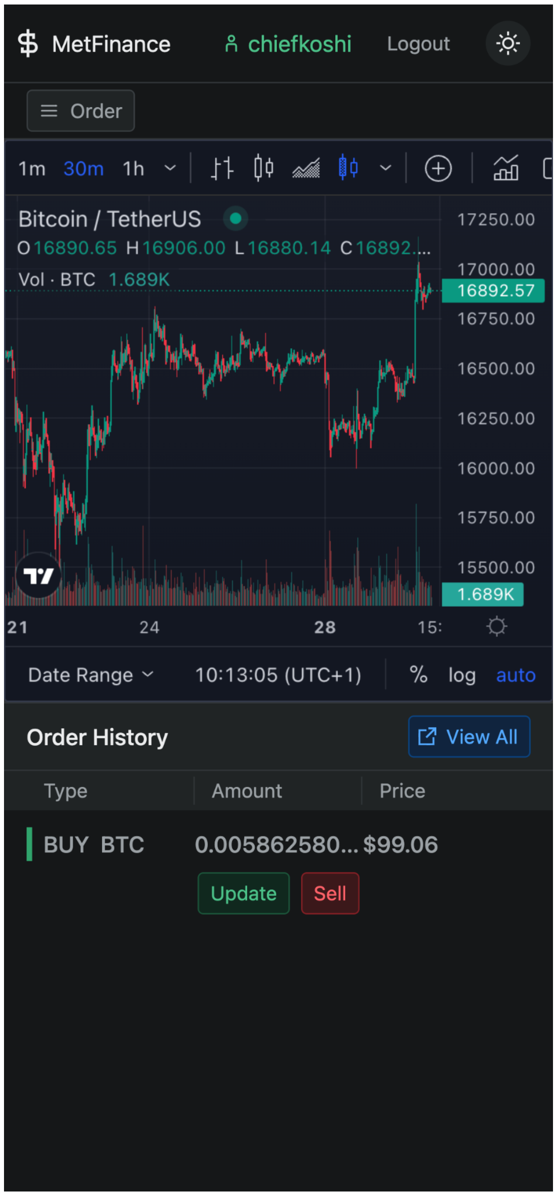
By pressing "View all" in the “Order History” section, the user is able to view all positions in large mode. If there are more than three positions, this is useful.



Type	Amount	Price	Created at
BUY BTC	0.005862580956770...	$99.06	30.11.2022, 10:03:33

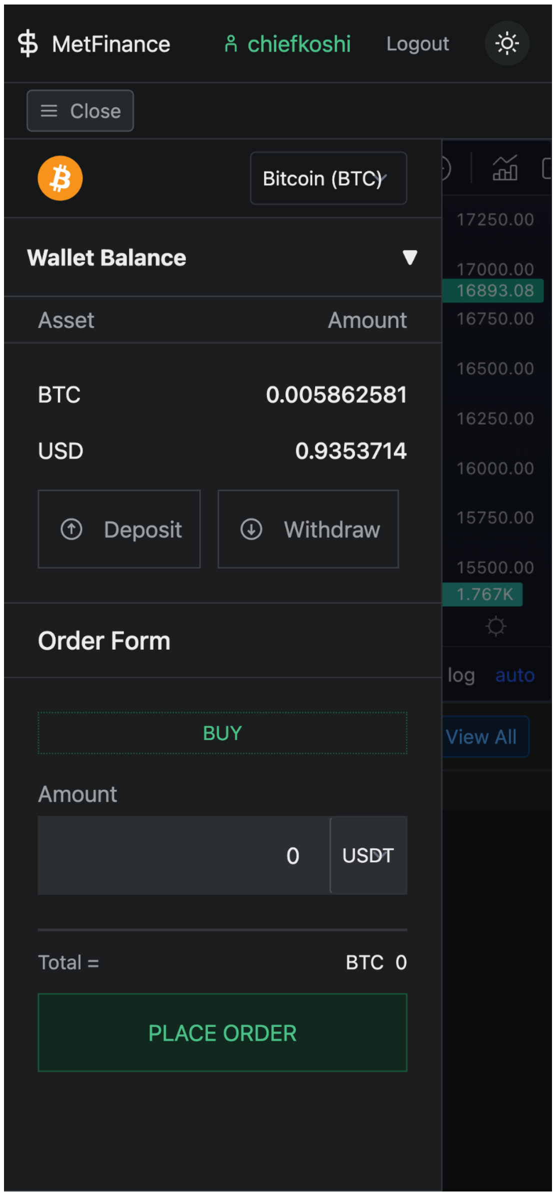
Figure 14: View all orders using /history.

The mobile version of the app performs similarly to the desktop version; however, it will display the OrderForm and wallets in a sidebar that can be toggled.



Order History

Type	Amount	Price
BUY BTC	0.005862580...	$99.06



Wallet Balance

Asset	Amount
BTC	0.005862581
USD	0.9353714

Order Form

Amount: 0 USDT

Total = BTC 0

Figures 15 & 16: Mobile version

Reflection of work

Implementing testing was perhaps the greatest difficulty we faced while developing this project. There are many ways to perform mocking and unit tests, and they all differ based on application architecture. Before we started rewriting the codebase, we attempted to implement tests on our current architecture. This proved incredibly difficult as we had not used any service or repository patterns on the initial MVP. Mocking the database directly should rarely be done and results in messy code. We therefore decided to implement the service pattern in order to mock the services when unit testing the controllers. This simplified the testing process greatly.

Rewriting the front-end codebase to use React was easy because we had already set up the front-end build tools in the MVP version, which made it easy to switch without any special configuration.

Conclusion

In this project, we were tasked with extending our prior MVP application. The created application is a fully functioning and responsive crypto trading application with JWT authentication, simulated trades, a real-time chart using Tradingview, and near real-time data and exchange rates using the CoinMarketCap API. The web client is written in React 18, and the backend (API) is written in ".Net Core 6." The assignment requirements, including the implementation of testing, logging, and a focus on Entity Framework core and CRUD, as well as all the functional requirements, have been met.

Attachment - Development guide

Test email: john@doe.com
Test password: johndoe1

Development

1. Make sure Node.js version 18 is installed
2. Unzip the project
3. cd into ./crypto_stocks
4. Rename .env.copy to .env
5. cd into ./ClientApp and run "npm install"
6. cd out of ClientApp and run the project (VSCode: "dotnet watch")
7. A fully functional app should now be available on "localhost:5001"

Production

1. Follow step 1-3 in the development guide
2. Run "dotnet publish -c Release -o "../dist" or build using Visual Studio
3. cd "../dist" or run with visual studio
4. Exec "./crypto_stocks" from dist folder

References

auth0. (n.d.). *JSON Web Token Introduction - jwt.io*. JWT.io. Retrieved November 30, 2022, from https://jwt.io/introduction

CoinMarketCap. (n.d.). *CoinMarketCap API Documentation*. CoinMarketCap. Retrieved November 30, 2022, from https://coinmarketcap.com/api/documentation/v1/#section/Introduction

ExecuteAutomation. (n.d.). *ASP.NET Core 6.0 Minimal API with Entity framework core*. Medium. Retrieved November 30, 2022, from https://medium.com/executeautomation/asp-net-core-6-0-minimal-api-with-entity-framework-core-69d0c13ba9ab

Jammeh, S. (n.d.). *Tw Classed*. TW-Classed | Tailwind with the DX of CSS in JS - TwClassed. Retrieved November 30, 2022, from https://tw-classed.vercel.app/

Kanjilal, J. (2022, August 11). *How to implement JWT authentication in ASP.NET Core 6*. InfoWorld. Retrieved November 30, 2022, from https://www.infoworld.com/article/3669188/how-to-implement-jwt-authentication-in-aspnet-core-6.html

Meta. (n.d.). *React Docs*. React Docs Beta. Retrieved November 30, 2022, from https://beta.reactjs.org/

Microsoft. (2022, August 12). *How to: Create User-Defined Exceptions*. Microsoft Learn. Retrieved November 30, 2022, from https://learn.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-user-defined-exceptions

NewtonSoft. (n.d.). Json.NET - Newtonsoft. Retrieved November 30, 2022, from https://www.newtonsoft.com/json

Posnick, J. (2019, July 18). *Keeping things fresh with stale-while-revalidate*. web.dev. Retrieved November 30, 2022, from https://web.dev/stale-while-revalidate/

Watmore, J. (2022, January 17). *.NET 6.0 - Global Error Handler Tutorial with Example*. Jason Watmore's. Retrieved November 30, 2022, from https://jasonwatmore.com/post/2022/01/17/net-6-global-error-handler-tutorial-with-example