

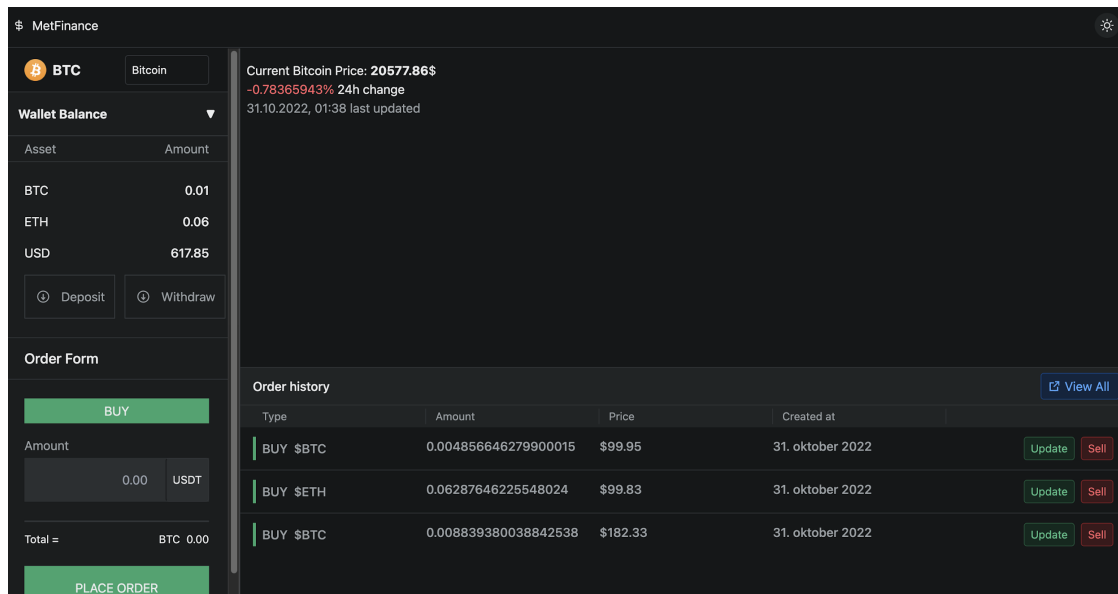
Webapplikasjoner - MetFinance

MetFinance is a crypto platform simulating the sale and purchase of “positions.” A position signifies a purchase of a certain stock (in this case, cryptocurrency). A position can be bought, updated, or sold off entirely. The current iteration of the application is classified as an MVP and consists of two front-facing endpoints.

1. The main app at “/”
2. The order history at “/history”

Visual overview

Dashboard

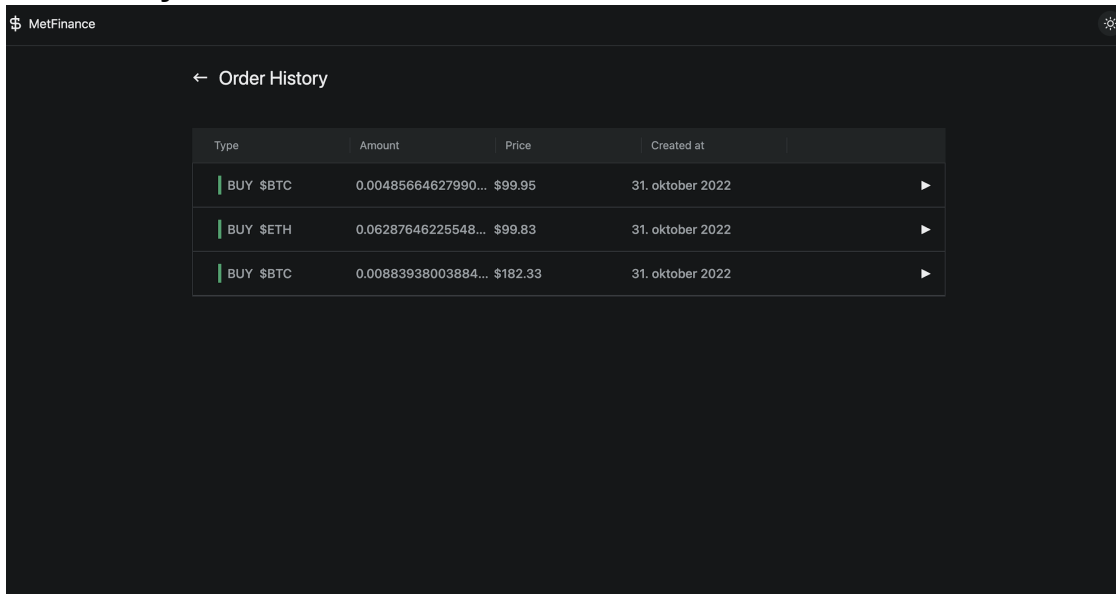


Dashboard: /

The main dashboard consists of three different sections

1. The sidebar. Containing a menu to select markets ranging from the top 10 crypto currencies, a sub-section with the user's current wallet balances, and the order form responsible for creating new orders.
2. The Main section; This is the largest section on the page and is responsible for displaying vital currency information for the selected market. We hope to expand this section in future editions of the application.
3. The order history; Responsible for displaying a list of all positions created by the user. Additionally, the user is able to update and sell their positions.

Full History



Type	Amount	Price	Created at
BUY \$BTC	0.00485664627990...	\$99.95	31. oktober 2022
BUY \$ETH	0.06287646225548...	\$99.83	31. oktober 2022
BUY \$BTC	0.00883938003884...	\$182.33	31. oktober 2022

Order history: `/history`

The order history page is responsible to display the full order data. Updates on the order's are not possible from this endpoint.

Technical overview

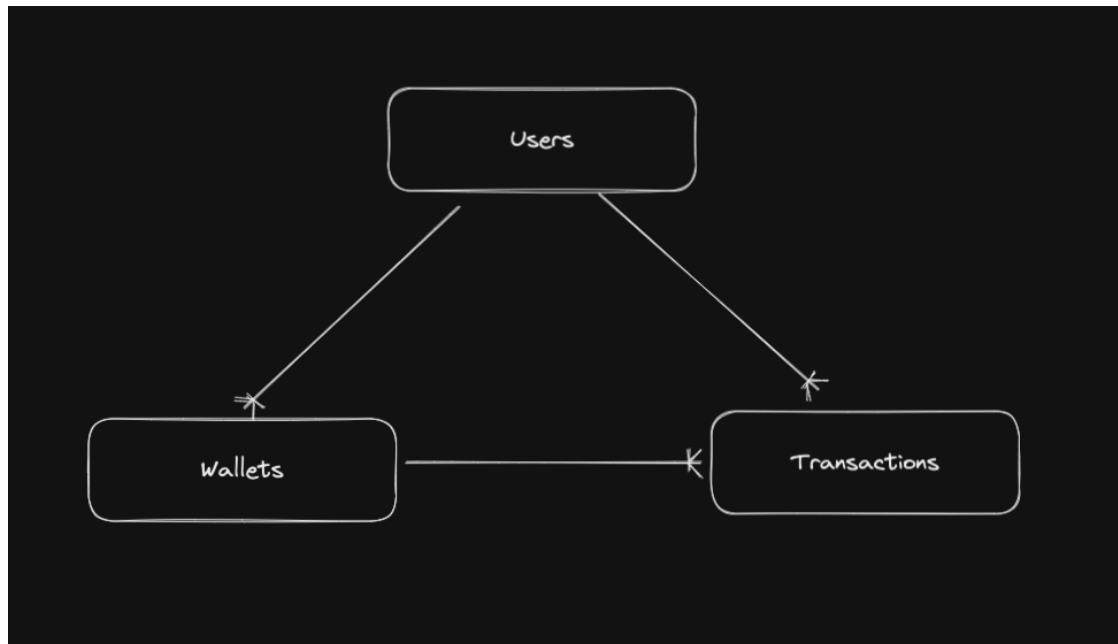
The Database

The database is modelled to closely resemble a real application with users and consists of 3 tables.

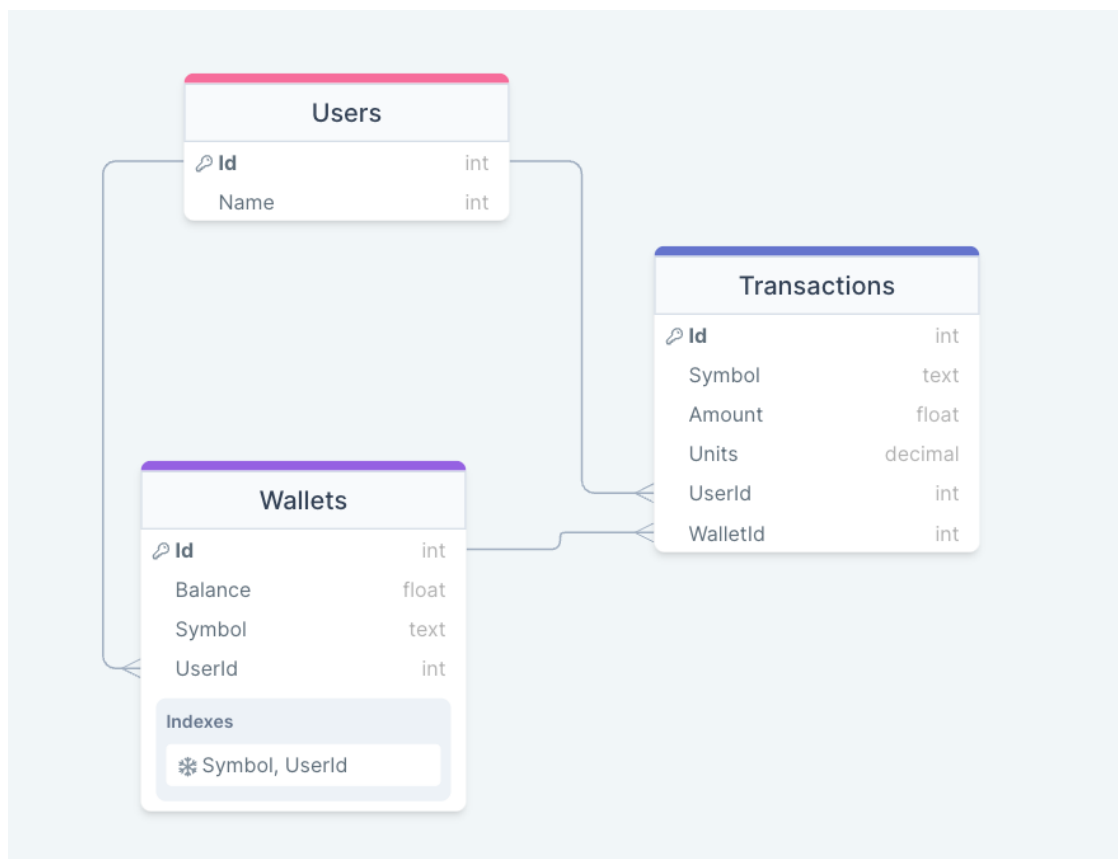
1. The Users table
2. The Wallets table - Containing the current balance of a given currency
3. The Transactions table - Contains data about a given position

The relationship between these can be described as follows:

One user can have multiple wallets, one wallet has multiple transactions, one transaction has a single wallet and one user can have multiple transactions.



Simple table diagram



Full Database Diagram

The application

Technologies

We decided to take a modern approach to during the development of the application. The application, consisting of the server and client, is built using the latest version of .net and modern frontend build systems. During development of the application we took consideration of several concepts like scalability, maintainability and developer experience. Therefore, we made the decision to use some of the latest tooling available to help efficiency in the development of the MVP and reduce time needed to perform future modifications to the application.

The list of technologies used is as follows:

- .Net Core 6
- Installed .Net packages:
 - Entity Framework Core 6
 - Newtonsoft.Json (for .net json conversion from external API's)
 - DotNetEnv (for loading .env files containing API keys etc..)
- Vite - [Vite](#) - Frontend build tooling enabling the use of the full ES2022 Spec
- Tailwind CSS & Sass - [Tailwind CSS](#) - Frontend CSS JIT compiler
- Radix Colors - [Colors – Radix UI](#) - A color system by Modulz to create accessible and good looking websites. Enabled through the [Radix Colors for Tailwind](#) plugin.
- [CSS.GG](#) - [CSS.gg](#) - CSS based Icon library
- Notyf - <https://github.com/caroso1222/notyf> - Tiny 3kb library to provide small toast messages

Using the latest release of .net

Several of the group's members had M1 chips and had previously encountered rosetta errors while executing .net 3 code. We therefore made the decision use the latest .net 6 release which has builtin arm64 support. This also allowed us to take advantage of HMR (hot module reload) inside the .net runtime.

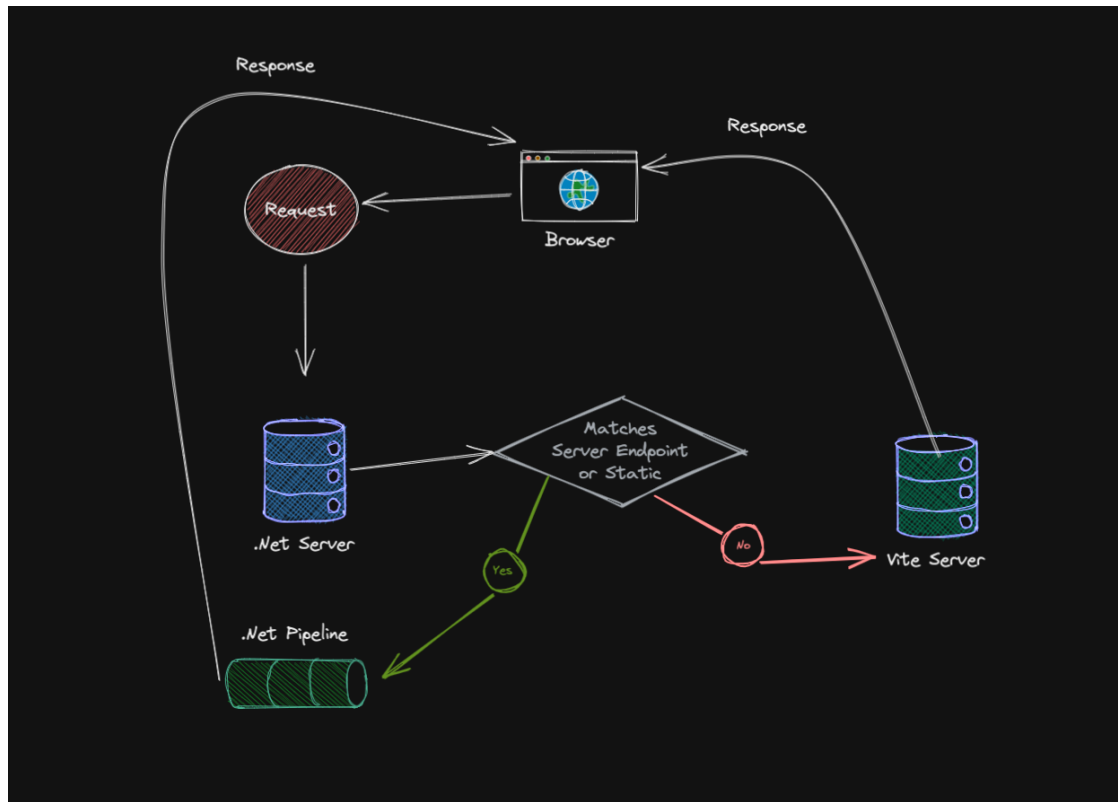
Using Vite as the preferred build tool

We decided to extend the default development environment provided by .net by utilising Vite, a modern front-end build tool for single and multi-page applications. Vite allowed us to iterate much faster as it uses native ES modules to provide HMR (hot module reloading), allowing us to see results within milliseconds of pressing save. Additionally, Vite has built in support for PostCSS allowing us to use Tailwind CSS's just in time compiler to generate CSS from regular html class-names before the app is built.

As Vite also compiles newer “not yet” browser-ready code down to ES6 we were able to drop the use of jQuery in favour of cleaner and newer code able to run on 95% of browsers with the exception of IE.

As Vite starts its own development server we had to make several changes to the configuration of our .net server. This includes making use of the

Microsoft.AspNetCore.SpaProxy package and proxying the requests from Vite to hit the .net server. This was performed following [this](#) guide, and making our own modifications. During development the Request → Response flow looks like this:



Development server request response pipeline

During production this changes, Vite is no longer needed and .Net will serve the static assets bundled by Vite from wwwroot.

The result of switching to modern tooling was a large boost in developer productivity and cooperation between the group members.

The Server

The .Net toolkit provides several ways to create Web API's. We decided to use the modified MVC approach which .Net provides when scaffolding a new SPA project. Controllers are set as the web facing interfaces and automatically created by the .Net runtime. Additionally controllers may use services and database contexts. As this is considered an MVP, we decided to not apply the repository pattern and rather directly dependency inject the DataContext in the controllers as it allowed for faster iteration and development. A helper class named ResponseData was also created to ease the front-to back-end integration providing a unified response object for all responses. Additionally, endpoints requiring request bodies use the DTO (Data Transfer Object) pattern to de-serialize the JSON string into classes.

The application contains two controllers which provide all the functionality needed for the application.

The StocksController

This controller handles the following requests:

- Requesting CoinMarketCap to obtain a list of the 10 most popular crypto currencies
- Listing all created positions for a given user
- Creating a new position - (Creates a crypto wallet if it doesn't exist)
- Updating a position (selling parts of or buying more units of a given currency)
- Selling the entire position

In order to enforce DRY code we decided to create a helper service class called CMCSERVICE. This class contains methods for fetching the latest stocks and getting the USD price for a given amount of a cryptocurrency.

The WalletsController

This controller handles the following requests:

- Getting a wallet for a given userId query parameter
- Depositing USD only - (Also creates the USD wallet)
- Withdrawing USD only

Database integration

We opted to not use a hosted database solution and rather provide a single SQLite file named LocalDatabase.db

Using the Entity Framework we created the 3 models following the ER diagram provided above. We have CRUD support on the “*Transaction*” model through the calls made in the StocksController. As the database is modelled to resemble the final product as close as possible we decided to seed the database during the OnModelCreating stage of the Entity Framework db migrations. A user with the “UserId” of 1 will be present when applying the database migrations. As no login functionality is provided in this iteration, this was an important step to making the other functionality work.

Frontend Development

When developing the frontend we used several concepts to make the page as interactive as possible with Vanilla javascript. We made our own implementation of the Observable pattern <https://www.dofactory.com/javascript/design-patterns/observer> to allow individual objects to act as global state which other related or unrelated code can “subscribe” to. This allows us to perform updates in one place and react to the updates in another without having to think about how we connect the two unrelated actions.

During development of the Order lists we found that looping over, and rendering our lists required applying event listeners and handlers outside of the list items in order to make

the “Sell” and “Update” buttons work. This made two very related actions decoupled from each other, resulting in confusing code. We therefore decided make use of native Web Components to handle this specific case.

https://developer.mozilla.org/en-US/docs/Web/Web_Components These are part of the Living HTML Standard and have wide browser support. This allowed us to hook into the element’s lifecycle and apply business logic in a much more approachable way.

Usage guide

Development

1. Make sure the Node.js runtime installed - vital for Vite to run
2. Unzip the project - The zip provides the full source code and the SQLite db
3. Rename .env.copy to .env
4. Cd into “./ClientApp” and run “npm install” from the command line.
5. Cd into root and run the project (“dotnet watch” in the command line for VSCode)
6. A fully functional app should now be available at “localhost:5001”

Production

1. Follow step 1-3 in the development guide
2. Run ‘dotnet publish -c Release -o “<OUT_DIR>”’ or build using Visual Studio
NOTE: the “dotnet publish” CLI command will ignore wwwroot inside of the release folder if executed from the project folder and show the fallback index.html instead. To prevent this either cd into the folder where the crypto_stocks.dll is located or build to external folder and run from there.
3. Execute the application with “dotnet <PATH_TO_FILE>”

Interacting

- Select different markets using the “Select Market” select menu in the top left sidebar
- Deposit some USD into your wallet using the Deposit button
- Create a buy order for the current market in the bottom of the sidebar
- Attempt to update or sell the entire order from the “Order History” tab in the bottom center
- View all the orders, and their JSON data) by pressing “View All” in the “Order history” list

References

Anderson, R., & Larkin, K. (2022, June 27). *Response caching in ASP.NET Core*.

Microsoft Learn. Retrieved October 31, 2022, from

<https://learn.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-6.0>

caroso1222/notyf: A minimalistic, responsive, vanilla JavaScript library to show toast notifications. (n.d.). GitHub. Retrieved October 31, 2022, from

<https://github.com/caroso1222/notyf>

CoinMarketCap API Documentation. (n.d.). CoinMarketCap. Retrieved October 31, 2022, from <https://coinmarketcap.com/api/documentation/v1>

Colors – Radix UI. (n.d.). Radix UI. Retrieved October 31, 2022, from

<https://www.radix-ui.com/colors>

CSS.GG. (n.d.). 700+ CSS Icons, Customizable, Retina Ready & API. Retrieved October 31, 2022, from <https://css.gg/>

JavaScript Observer Design Pattern. (n.d.). Dofactory. Retrieved October 31, 2022, from <https://www.dofactory.com/javascript/design-patterns/observer>

Keithn. (n.d.). *keithn/dotnetviteguide: Guide to getting started with .NET 6, Vite, and Vue 3*. GitHub. Retrieved October 31, 2022, from

<https://github.com/keithn/dotnetviteguide>

Tailwind CSS. (n.d.). Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. Retrieved October 31, 2022, from <https://tailwindcss.com/>

Vite.js. (n.d.). Vite | Next Generation Frontend Tooling. Retrieved October 31, 2022, from <https://vitejs.dev/>

Web Components | MDN. (2022, September 8). MDN Web Docs. Retrieved October 31, 2022, from https://developer.mozilla.org/en-US/docs/Web/Web_Components