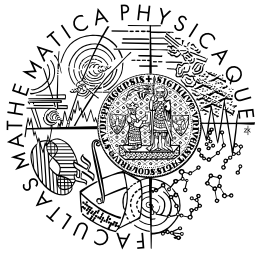

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Dmitry Simonov

Portfolio performance evaluation

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Filip Zavoral, Ph.D.

Study programme: Computer Science

Study branch: Programming and Software Development

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor RNDr. Filip Zavoral, Ph.D. for his time, patience, and help provided during implementation of this thesis.

Title: Portfolio performance evaluation

Author: Dmitry Simonov

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: The goal of this work is to develop a tool for tracking and analysis of investment portfolios, with support for automatic financial data retrieval and highly customizable chart creation. In this thesis, we research existing investment tracking tools, comparing them based on their supported features. We also analyze different methodologies of evaluating performance of an investment. Following this, we develop a web application allowing the user to import, track, and chart their investments. Financial data for tracked securities is automatically fetched from various data sources using a handcrafted library, which can be used for retrieval of any kind of data from different sources, while supporting easy addition or replacement of such sources. Finally, we evaluate the implemented application and suggest possible extensions and improvements.

Keywords: investment performance, web application, react, .net

Contents

Introduction

Investment tracking and evaluation are some of the most important activities pertaining to investment management. With the rise of online trading platforms and increased access to financial markets, more individuals are seeking to invest their financial resources to achieve short-term or long-term gains. However, managing these investments can be a complex and time-consuming process, more so if they are spread across different portfolios. Furthermore, evaluating their performance is challenging without the proper tools and knowledge.

While numerous tools exist to assist investors in making investment decisions, many such tools are either proprietary or narrowly specialized. This indicates a need for a generic application which would allow investors to aggregate and track their investments in a single place, while providing key performance metrics assisting them in their decision-making.

This work aims to provide investors with such an application. It should be as easy to use as possible, while being flexible enough to accomodate the more specific use cases needed by advanced investors.

To achieve this, the application should be able to automatically retrieve and maintain financial data, ensuring up-to-date information on the performance of tracked investments. Furthermore, as one of the key tools investors use to measure their investments are charts, the application should allow the user to chart their investments using highly configurable multi-line charts, enabling comparison between the performance of different investments or securities. The user should then be able to display multiple such charts in a single view, allowing them to see relevant information at a single glance.

This work consists of three key parts. First, we analyze the specific requirements stemming from the goals of this work, while also discussing the methodology of evaluating an investment's performance and existing solutions.

Then, we design, develop, and test the application according to requirements discovered in the analysis phase.

Finally, we evaluate the application and suggest possible extensions, while also describing how these extensions could be implemented in the application's architecture.

Glossary

Before delving into analysis and development of the application, it would be beneficial to establish basic terminology which is commonly used throughout this work:

- **Instrument**: A priced and typically tradable asset, such as company stocks, cryptocurrencies, exchange-traded funds (ETFs), commodities, or indexes. These assets typically have a name and a symbol, such as *Apple Inc.* with an *AAPL* ticker symbol.
- **Instrument price**: The price at which the instrument was traded at a certain time.

- **Position**: The amount of a particular instrument held by the investor.
- **Transaction**: An increase or a decrease of a position, executed at a certain time for a certain price.
- **Portfolio**: A collection of positions.
- **Value**: Instrument value refers to the price of the instrument at a specific time, position or portfolio value refers to current market value of assets held in the investment.
- **Profit**: Instrument profit refers to the change of instrument's price over a specific time period, position or portfolio profit refers to the monetary return on investment.
- **Performance**: Instrument performance refers to percentage change of instrument's price over a specific time period, position or portfolio performance refers to percentage return on investment.
- **Chart**: A multi-line chart consisting of instrument, portfolio, or position lines.
  - **Value chart**: Charts the value of selected financial entities.
  - **Profit chart**: Charts the profit of selected financial entities between the start and end of the chart date range.
  - **Performance chart**: Charts the performance of selected financial entities between the start and end of the chart date range.
  - **Aggregated profit chart**: Charts the profit of selected financial entities between each two points of the chart.
  - **Aggregated performance chart**: Charts the performance of selected financial entities between each two points of the chart.

1. Analysis

In the following chapter, we further analyze the goals described in *Introduction*. We start by discussing the different methodologies for investment performance evaluation.

Following this, we analyze available financial data sources and existing solutions.

Finally, based on the results of the analysis, we formulate the functional and non-functional requirements. Additionally, we describe the use cases of the application, providing example use case scenarios for several of them.

1.1 Evaluating portfolio performance

In this section, we examine the methods available for measuring the performance of an investment and select the approach that best fits the needs of the application.

1.1.1 Problem

As was indicated in the goals of this work, the primary metric in which we are interested is the returns earned by portfolios and their individual positions based on market changes. This means that this analysis will focus on techniques which can be used to measure such returns, and not techniques for measuring metrics such as risks or return efficiency.

The *return* of an investment can generally be defined as a ratio relating *how much was gained or lost* given *how much was risked* [Feibel, 2003]. For example, if we invest $10 in a fund and we get $15 back, we have gained $5. To express this return as a percentage, we can use the following calculation:

$$\frac{5}{10} \times 100 = 50\%$$

Such calculation is enough to measure the performance of individual instruments, where gains or losses are triggered by changes in the market value of the instrument. However, for investment portfolios this simplified approach does not accurately reflect reality due to several complications [Feibel, 2003]:

- Investors may contribute and withdraw from their portfolios at arbitrary times.
- Returns produced by the investment manager and returns experienced by the investor need to be evaluated differently.
- Returns spanning multiple valuation periods may need to be compounded.

To account for these complications, two different measures of return were developed:

- Money-Weighted Return (MWR)

- Time-Weighted Return (TWR)

In the following subsections, we examine both of these measures and the methods used to calculate them.

1.1.2 Money-Weighted Return

The Money-Weighted Return is a statistic reflecting how much money was earned during the measurement period, and can be described as a measurement reflecting the performance as experienced by the investor [Feibel, 2003]. Additionally, it accounts for timing and size of the cash flows inbetween. It does so by calculating a weighting adjustment for each cash flow, representing the proportion of the period during which the cash flow is available to be invested.

There are two common methods of calculating MWR - **internal rate of return (IRR)** and **Modified Dietz method**.

Internal Rate of Return

The internal rate of return is defined as *the rate of interest at which the present value of net cash flows from the investment is equal to the present value of net cash flows into the investment* [Kellison, 2008, p.252]. Essentially, it is the value r which satisfies the following formula:

$$\sum_{n=0}^{N} \frac{C_n}{(1+r)^n} = 0 \tag{1.1}$$

where:

- N - the total number of subperiods
- C_n - cash flow (positive or negative) in subperiod n

The resulting r represents the rate of return on each cash flow in and out of the portfolio, compounded over the time during which the cash flow was available for investment.

Alternatively, by substituting t for $1 + r$ and multiplying both sides of the equation by t^N, the formula can be rewritten as follows:

$$\sum_{n=0}^{N} (C_n t^{N-n}) = 0 \tag{1.2}$$

While IRR is a fairly intuitive measure, the formula above is a high degree polynomial, which is computationally expensive to solve.

Modified Dietz method

The modified Dietz method can be defined as an approximation of MWR using simple interest rate principle (compared to IRR which uses a compounding principle). The Modified Dietz return R is calculated as follows [Feibel, 2003]:

$$R = \frac{S - E - \sum_{n=1}^{N} C_n}{S + \sum_{n=1}^{N} (C_n * \frac{N-n}{N})} \tag{1.3}$$

where:

- S - starting market value
- E - ending market value
- N - total number of subperiods
- C_n - cash flow (positive or negative) in subperiod n

The advantage of the modified Dietz method compared to IRR is ease of calculation, as the formula above is a closed-form solution.

1.1.3 Time-Weighted Return

Contrary to MWR, the Time-Weighted Return eliminates the effect of external flows, such as contributions or withdrawals from a portfolio. This means that this statistic only accounts for changes in value caused by market changes or a reallocation of assets. This is achieved by splitting the total measurement period into multiple subperiods, the boundaries of which are determined by the dates of each cash flow. The return for each subperiod is then calculated and compounded, resulting in the following formula for total return R [Feibel, 2003]:

$$R = \prod_{n=1}^{N} (1 + R_n) \tag{1.4}$$

where:

- N - total number of subperiods
- R_n - return in subperiod n

Because TWR does not account for timely contributions or withdrawals by the investor, it can be described as a metric measuring the performance of the investment manager, rather than the investor [Feibel, 2003].

Additionally, to calculate the return for each subperiod, we need to know the portfolio valuation at the time of each cash flow, whereas for MWR only the valuation at the start and the end of the measurement period is necessary.

1.1.4 Conclusion

As the application is aimed towards investors measuring the performance of their own investments, MWR seems to be the more appropriate choice.

Out of two discussed methods for calculating MWR, IRR has the benefit of higher accuracy, while the modified Dietz method has the benefit of easier calculation. Due to the nature of this work, higher accuracy of IRR is perceived as more important than ease of calculation. For this reason, IRR was the methodology of choice for portfolio performance calculation.

To correctly use IRR in the application, two points need to be considered:

- IRR represents the average rate of return in each subperiod, not the return over the whole measurement period.

- IRR cannot be calculated directly, it requires an iterative trial-and-error approach.

Knowing IRR r, it is trivial to calculate the return over the whole measurement period. Assuming N subperiods, r can be compounded over these periods by calculating r^N.

However, with IRR formula being a high degree polynomial, finding its real roots algebraically might not be possible. For this reason, an approximation root-finding algorithm can be used. Several such algorithms exist, such as Newton's method, the secant method, or Horner's method [Atkinson, 1989].

Because of its relative simplicity, we utilize Newton's Method in this work. Additionally, to further simplify the calculation, formula 1.2 is used.

1.2 Financial data sources

As is evident from the goals of this work, the application should be able to automatically retrieve and maintain financial data, such as instrument prices, instrument splits, and currency exchange rates. In this section, we examine possible options and evaluate their suitability for the application's needs.

1.2.1 Evaluation criteria

Before evaluating individual data sources, it would be beneficial to establish some key criteria based on which the choice could be made.

- **Scope of data**: The number of different instruments, exchanges, or currencies supported, and the type of data available.
- **Rate limits**: Many public data sources limit the number of requests which a single IP address or a single token may use, which could reduce the availability of financial data in the application.
- **Ease of integration**: The number of requests needed to retrieve necessary data, or the perceived complexity of implementation.

1.2.2 Alpha Vantage

Alpha Vantage[1] is a popular freemium API (Application Programming Interface) providing stock market data.

- Scope of data - The API provides a wide range of financial data, with reported support for 100 000 instruments. The API provides historical adjusted EOD (end of day) prices, intraday adjusted prices, stock split information, and (crypto)currency exchange rates.
- Rate limits - 5 requests per minute and 500 requests per day for the free version, up to 1200 requests per minute and no daily limits for paid versions.

[1] `https://www.alphavantage.co/`

- Ease of integration - Alpha Vantage is a traditional RESTful API returning JSON (JavaScript Object Notation) data, allowing retrieval of complete price and exchange rate history with a single request.

Alpha Vantage can be considered a viable option, with its only drawback being its fairly strict rate limits.

1.2.3 Finnhub

Finnhub[2] is a freemium API which provides real-time and historical prices and cryptocurrency exchange rates.

- Scope of data - The API's primary focus is on US-based stocks, with several types of data (such as stock split data) being provided only as part of the premium package.
- Rate limits - 30 requests per second and 60 requests per minute for the free version, up to 900 requests per minute for the paid versions.
- Ease of integration - Similar to Alpha Vantage, Finnhub provides a RESTful API returning JSON data. In addition, it offers a WebSocket endpoint for real-time price updates.

While Finnhub generally seems like a valid choice, it is important to note its limitations with regard to its focus on US-based instruments and stocks in particular.

1.2.4 Tiingo

Tiingo[3] is a financial markets API providing intraday and historical instrument prices, as well as cryptocurrency exchange rates. The intraday prices are based on raw feeds provided by IEX exchange.

- Scope of data - Tiingo reports 84 000 supported securities with a 30-year price history. The API provides historical adjusted EOD (end of day) prices, intraday prices, stock split information, and (crypto)currency exchange rates.
- Rate limits - 50 requests per hour and 1000 requests per day for the free version, up to 5000 requests per hour and 50000 requests per day for the paid version.
- Ease of integration - Tiingo provides a RESTful API returning JSON data with a WebSocket endpoint enabling direct access to the IEX feed. However, due to the nature of the IEX feed, the intraday prices retrieved from this API will not be adjusted for stock splits.

While Tiingo does not provide adjusted intraday prices, its extensive price history can be used for downloading historical daily prices.

[2] `https://finnhub.io/`

[3] `https://www.tiingo.com/`

1.2.5 RapidAPI Mboum Finance

Mboum Finance[4] is a freemium financial data API provided by RapidAPI.

- Scope of data - Mboum Finance provides 10 years of historical EOD prices and 1 month of intraday prices. Additionally, it provides a stock split history for supported stocks. The total number of supported instruments is unknown, however, preliminary experiments showed support for certain European ETFs which were not available in previous sources.
- Rate limits - 10 requests per minute and 500 requests per month for the free version, up to 600 requests per minute and unlimited requests per month for the paid version.
- Ease of integration - Mboum Finance is a RESTful API returning JSON data. One advantage of this API is ease of retrieval of stock splits, as it provides a complete split history in a single request.

Mboum Finance has fairly strict rate limits, however, it seems to support more instruments than other considered data sources.

1.2.6 Yahoo Finance

Yahoo Finance[5] is a complete service providing financial data, reports, and news. It additionally offers various personal finance tracking tools, including portfolio tracking, which will be further discussed in the next section.

While Yahoo used to provide a direct access to their financial API, it was officially discontinued several years ago. Nevertheless, due to the fact that Yahoo Finance data is commonly used in similar applications, and due to the volume of data provided, it should be considered for this work as well.

Currently, the only two methods of retrieving data from Yahoo Finance are web scraping or their internal APIs[6].

- Scope of data - Yahoo Finance provides complete instrument price, stock split, and currency exchange rate histories for a large number of tickers and currencies.
- Rate limits - Unknown
- Ease of integration - Yahoo's internal APIs are not official, which means that there is no documentation available for them. It is also impossible to know when specific APIs will get discontinued. On top of that, Yahoo seems to be trying to prevent unauthorized access to such APIs[7], however,

[4] `https://rapidapi.com/sparior/api/mboum-finance`

[5] `https://finance.yahoo.com/`

[6] It is important to note that several organizations published their own unofficial APIs providing Yahoo Finance data retrieved using one of the methods above. However, their reliability is questionable, with one of the most popular APIs unexpectedly going out of service as recently as February 2023. For analysis purposes, we ignore such APIs and assume that a custom retrieval mechanism needs to be implemented.

[7] `https://github.com/ranaroussi/yfinance/issues/1407`

this does not seem to affect their quote and price history endpoints. It would also be possible to scrape the necessary data from Yahoo Finance website, however, it requires significant development time, while providing little reliability due to possible website changes.

One major advantage of Yahoo Finance is that it does not require registration or an API token to use, while also providing large volumes of data to the users. In the context of this work, Yahoo Finance could be implemented as a valid default source if the user does not want to register for other data sources.

1.2.7 Exchange Rate Host

ExchangeRate.host[8] is a free service providing current and historical currency and cryptocurrency exchange rates.

- Scope of data - Reports 170 supported currencies and 6000 supported cryptocurrencies with 20 years of historical data.
- Rate limits - The rate limits are IP-based, dynamic, and are reported in response headers. Preliminary experiments showed a consistent rate limit of 2000 requests per minute.
- Ease of integration - ExchangeRate.host provides a RESTful API returning JSON, comma-separated values (CSV), TSV (tab-separated values), or XML (Extensive Markup Language) data. Downloading historical data is limited to 366 days per request.

Similar to Yahoo Finance, ExchangeRate.host does not require any credentials to use, while also providing generous rate limits that are more than sufficient for the application's needs. The only drawback of this API is its limitation on the amount of historical data that can be downloaded per request, but this issue can be easily worked around.

1.2.8 Open Exchange Rates

Open Exchange Rates[9] is an API providing current and historical currency exchange rates.

- Scope of data - The API reports 200 currencies with 20 years of historical data. The free version is limited to exchange rates from USD only with hourly exchange rate updates.
- Rate limits - 1000 requests per month for the free version, up to unlimited requests for the paid version. However, for historical data, each day is counted as one request, so the free version only enables retrieval of 1000 days per month.

[8] `https://exchangerate.host/`
[9] `https://openexchangerates.org/`

- Ease of integration - The API provides RESTful endpoints for historical, time-series, and latest exchange rates. The time-series endpoint is limited to one month of data per request, which significantly increases the number of requests needed to download the complete history of exchange rates for a single currency.

The biggest disadvantage of Open Exchange Rates is its restriction to USD exchange rates for the free version.

1.2.9 Conclusion

While there are numerous financial data sources available, none of them completely cover the application's needs. Furthermore, due to fairly strict rate limits, the application cannot be reasonably expected to retrieve necessary information in real time.

This means that the application should maintain its own financial data, which would be composed from data retrieved from different data sources. Additionally, to maximize reliability, the application should be able to switch between data sources in case of their unavailability, or in case a data source's rate limit is exceeded.

For the purposes of this application, we do not need to maintain complete and continuous price and exchange rate histories for each instrument and currency, which would require a significant amount of storage space. Instead, the following intervals were deemed to be enough:

- Five minutes for instrument prices in the past day.
- One hour for instrument prices in the past five days.
- One day for instrument prices older than five days.
- One day for currency exchange rates from user-selected default currency.

  If the application requires currency conversion between non-default currencies, the appropriate exchange rate can be estimated using exchange rates from the default currency.

Based on the information provided in the previous subsections, we have opted for the following data sources and data types:

- Yahoo Finance
- Alpha Vantage
- Tiingo
- RapidAPI Mboum Finance
- Exchange Rate Host
- Open Exchange Rates

1.3 Related work

In this section, we examine existing portfolio tracking solutions and compare them to the application being built.

1.3.1 Evaluation criteria

With regards to this work's goals, we examine related solutions based on the following set of criteria:

- **Custom data support**: The level at which the solution supports tracking of custom instruments or positions.
- **Tracking capabilities**: Financial metrics which can be tracked in the application and scope of available market data.
- **Charting capabilities**: Configurability and usability of charts.
- **User experience**: How easy it is to learn and use the application. This criterion is primarily subjective.
- **Business model**: Whether the application is free to use, and whether its source code is publicly available.

1.3.2 Yahoo Finance

Yahoo Finance was first discussed in the *Financial data sources* section in the context of their API, however, they also provide a web-based portfolio tracking tool, which is the target of this analysis.

- Custom data support - Yahoo Finance enables tracking custom positions, however it does not support maintaining custom price or split histories, so the analysis is limited only to returns based on entered transactions.
- Tracking capabilities - Yahoo Finance offers an incredibly large volume of data with real-time updates, while also enabling tracking of multiple portfolios, their returns, and various financial metrics of individual instruments.
- Charting capabilities - Yahoo Finance enables creation of highly configurable charts. However, these charts cannot be saved or laid out in a single view.
- User experience - The interface of Yahoo Finance is unreasonably cluttered to the level where it is hard to find necessary information at a single glance. On the other hand, its chart configuration functionality is very flexible and easy to use.
- Business model - All the tools provided by Yahoo Finance are free to use, but closed-source.

Yahoo Finance provides a large volume of data and very powerful charting and analysis capabilities, with its drawbacks being lack of support for custom data and inability to save charts.

1.3.3 Digrin

Digrin[10] is a web-based portfolio manager with focus on dividend returns. This project was originally created as a bachelor's thesis[11] at Masaryk University, with further improvements being made in a master's thesis[12] by the same author.

- Custom data support - Digrin maintains its own list of supported instruments, which means that there is no option to create and track custom instruments or positions. However, the users can attempt to add an unsupported symbol on a separate page, after which the application will attempt to load the data for that symbol from Yahoo Finance.
- Tracking capabilities - Digrin's primary focus is tracking of dividend growth, however, it also reports changes in portfolio value based on market changes.
- Charting capabilities - Digrin renders basic charts for portfolio value, position value, and dividend statistics. However, there is no option to create custom charts.
- User experience - Generally Digrin's interface is well structured and easy to use.
- Business model - Digrin operates on a freemium model, with some features only being available to paid subscribers. Free users are also limited in the number of portfolios or transactions they can enter into the application. The application is closed-source.

Digrin's approach to portfolio tracking is quite similar to this work, with the main difference being its focus on dividends, rather than market-based performance and customized charting.

1.3.4 Portfolio Performance

Similar to this work, Portfolio Performance[13] is an investment portfolio manager with focus on portfolio performance tracking. It is a desktop application available on Windows, Linux, and macOS.

- Custom data support - Portfolio Performance fully supports tracking custom instruments and positions.
- Tracking capabilities - Portfolio Performance retrieves its data from multiple data sources, and allows tracking of multiple key metrics, such as IRR and risk indicators. However, it only maintains daily historical prices of instruments.
- Charting capabilities - Portfolio Performance supports custom charts, however, their configurability is limited to date range and data interval. Custom charts can be saved as separate views or exported to a file, with no option to lay multiple charts in a single view.

[10] `https://www.digrin.com/`
[11] `https://is.muni.cz/th/ia1o6/?lang=en`
[12] `https://is.muni.cz/th/gve1x/?lang=en`
[13] `https://www.portfolio-performance.info/en/`

- User experience - The user interface of the application can be described as outdated with a steep learning curve.
- Business model - The application is free to use and is open-source. Its source code is available on GitHub[14].

Out of all projects discussed in this section, Portfolio Performance's functionality and goals align the most with this work. Its main drawbacks are its unintuitive user interface and limited chart configurability.

1.3.5 Sharesight

Sharesight[15] is a popular commercial web application for portfolio tracking and analysis, reporting both capital gains and dividends.

- Custom data support - Sharesight supports tracking custom instruments and positions with the exception of instrument splits, which are managed on the level of individual positions.
- Tracking capabilities - Sharesight offers a vast number of supported instruments from different regions and exchanges, and is capable of tracking capital gains, dividends, and currency gains of whole portfolios and individual positions.
- Charting capabilities - Sharesight renders simple charts displaying changes in instrument price, portfolio performance, or position performance. While these charts support basic configurability, such as the date range of the chart or its type, there is no option to build custom multi-line charts, save them, or display them in a single view.
- User experience - Sharesight offers a modern and clean user interface which is easy to learn.
- Business model - Sharesight is a freemium project with severe limitations imposed on free users, such as limited number of portfolios and positions. The application is closed-source.

One of Sharesight's biggest advantages is its integration with over 180 brokers, which enables the application to automatically import transactions from supported brokers. However, it is a commercial project which is hardly usable with a free plan. Additionally, it lacks support for custom charts.

1.3.6 Conclusion

As can be seen in the previous sections, there are many different approaches to tracking investments. With that in mind, we can summarize the goals of this work (referred to as *PortEval*) using the same evaluation criteria:

[14] `https://github.com/buchen/portfolio`
[15] `https://www.sharesight.com/`

- Custom data support - PortEval offers complete support for custom instruments and positions, including instrument prices and splits.
- Tracking capabilities - PortEval automatically retrieves market data from multiple different sources. It is capable of tracking monetary and percentage capital gains of portfolios and individual positions. Reported gains are automatically adjusted for currency exchange rate movements.
- Charting capabilities - PortEval supports creation of custom multi-line charts, enabling comparisons between different portfolios, positions, or instruments. Multiple types of charts are supported, with configurable date range, currency, data intervals, aggregation frequencies, and line styles. Charts are automatically saved on each change, and the application supports laying out saved charts in a custom grid.
- User experience - PortEval offers a simple and modern user interface, providing a good balance between the scope of data displayed in a single view, and the complexity of the interface.
- Business model - PortEval is free to use and open-source.

1.4 Requirements

Based on conducted analysis, we can formulate the requirements of the application. The following section describes the functional and non-functional requirements of the application, as well as use cases generated from these requirements including sample use case scenarios.

The functional requirements describe the concrete functionality which the application must offer to its users. These requirements are constructed as high-level statements which outline the specific tasks that the application must be able to perform.

In contrast, the non-functional requirements characterize the properties and limitations of the system. They provide the constraints under which the functional requirements must be fulfilled.

1.4.1 Functional requirements

FR1 The application must allow the user to track various investment instruments, including their prices and splits.

FR2 The application must allow the user to create investment portfolios.

FR3 The application must allow the user to open positions in their portfolios.

FR4 The application must allow the user to enter new transactions executed within a position.

FR5 The application must allow the user to see the value, profit, and performance of instruments, portfolios, and positions.

FR6 The application must allow the user to create and save custom multi-line charts showing the value, profit, performance, aggregated profit, or aggregated performance of selected instruments, portfolios, or positions.

FR7 Chart lines rendered by the application must additionally display small icons indicating the transactions performed on the underlying instrument, portfolio, or position.

FR8 The application must allow the user to display saved charts in a custom grid.

FR9 The application must allow the user to set different currencies for portfolios and charts, executing currency conversion automatically where needed.

FR10 The application must allow the user to import instruments, portfolios, positions, prices, and transactions in CSV format.

FR11 The application must allow the user to export available data in CSV format.

FR12 The application must be able to download instrument prices automatically from external sources.

FR13 The application must be able to download currency exchange rates automatically from external sources.

FR14 The application must be able to download stock splits automatically from external sources.

FR15 The application must allow the user to manually add prices and splits to an instrument.

FR16 The application must allow the user to set an application-wide default currency, which will then be automatically pre-set for portfolios, instruments, and charts.

1.4.2 Non-functional requirements

NFR1 The application must attempt to maintain five-minute intervals between instrument prices in the past day.

NFR2 The application must attempt to maintain one-hour intervals between instrument prices in the past five days.

NFR3 The application must attempt to maintain one day intervals between instrument prices older than five days.

NFR4 The application must attempt to maintain one day intervals between currency exchange rates from the application-wide default currency to at least 100 other currencies.

NFR5 The application must prevent uncontrolled growth of its storage size by cleaning up overabundant data.

NFR6 The application must allow integration of new data sources without any changes to application logic.

NFR7 The application must support at least two different external instrument price sources, and two different external currency exchange rate sources.

NFR8 The application must be able to render any chart containing 10 lines in under two seconds.

NFR9 The application must enable drag-and-drop functionality for the custom chart grid.

NFR10 The application must allow the user to configure date and number formats.

1.4.3 Use cases

In this subsection, we define the use cases of the application, drawing upon the requirements specified in the previous section. For this purpose, five UML (Unified Modeling Language) diagrams were created using PlantUML[16], each covering one part of the system's expected functionality. These diagrams are contained in figures 1.1, 1.2, 1.3, 1.4, and 1.5.

Additionally, we describe several such use cases using use case scenarios. Simple use cases were omitted for brevity.

Actors

In the context of the system, we will define two actors:

- **User**: The human user of the system, typically an investor who wants to track the performance of their investments.
- **Timer**: A time-based scheduling mechanism which triggers certain actions within the system.

Use case: Configure chart

- **Goal**: A user should be able to configure the chart's type, date range, currency, frequency, and lines.
- **Initial state**: The user has the chart editing view open for a specific chart.
- **Normal flow**:
  1. The user sets the chart type to *price*, *profit*, *performance*, *aggregated profit*, or *aggregated performance*
  2. If the chart type is set to *price*, *profit*, or *aggregated profit*, the user may change the currency of the chart. By default, the application-wide default currency is set.

[16] `https://plantuml.com/`

3. If the chart type is set to *aggregated profit* or *aggregated performance*, the user may change the aggregation frequency. By default, weekly aggregation frequency is used.
4. The user may set a specific start and end date of the chart, or they may use one of the pre-defined ranges, such as *5 days*, *1 month*, *3 months*, or *1 year*.
5. The user may add lines to the chart or remove lines from the chart. Each line corresponds to an existing instrument, portfolio, or position.
6. The user may change the color, dash style, or width of each line added to the chart.
7. The user may rename the chart.

- **Final state**: The chart is saved in the application under the specified name.

Use case: Add an instrument

- **Goal**: A user should be able to add a new instrument to the application.
- **Initial state**: The user has the instrument creation form open.
- **Normal flow**:

  1. The user fills in the name of the instrument, its symbol, its currency, and its type.
  2. Optionally, the user may fill in the exchange where the instrument is traded or a note.
  3. The user submits the instrument for creation.

- **Exception**: An instrument with the specified symbol already exists in the system.

  - **Exception flow**: The instrument is not created and an error message is displayed to the user.

- **Final state**: The instrument is created in the system.

Use case: Download missing instrument prices

- **Goal**: The system should automatically download price history for created instruments. Additionally, it should periodically attempt to download any missing prices of existing instruments.
- **Initial state**: A new instrument was created or 24 hours have elapsed since the last missing price download for an existing instrument.
- **Normal flow**:

  1. The system analyzes the existing prices of the instrument, finding periods with missing prices.

2. The system attempts to find missing prices for such periods using external data sources.
3. If any prices were found, then these prices are saved into the system. If there are missing intervals in the downloaded prices, for example during times when the instrument is not traded, then prices in these intervals are extrapolated based on the last known price before the missing interval.

- **Exception**: Communication with an external data source fails in step 2.
  - **Exception flow**: The system retries the data source, while additionally attempting to download the data from an alternative source. If no data source is responsive after multiple retries, then the normal flow continues as if no prices were found.
- **Final state**: Downloaded missing prices are saved into the system.

Use case: Validate position

- **Goal**: The system should not allow changes in transactions which result in a position's size falling below zero.
- **Initial state**: A transaction was created, modified, or deleted.
- **Normal flow**:
  1. The system calculates the size of the position after each known transaction.
  2. The change to the position's transactions is allowed to continue.
- **Exception**: Position size falls below zero at any point in step 1.
  - **Exception flow**: The system prevents the change from being executed, displaying an error message to the user.
- **Final state**: The change is successfully executed.

Use case: Open a position

- **Goal**: The user should be able to open positions in portfolios.
- **Initial state**: The user has the position creation form open.
- **Normal flow**:
  1. The user selects an instrument from the list of instruments available in the application.
  2. The user enters the price and the size of the initial transaction.
  3. Optionally, the user may enter a note to be displayed in the position.
  4. The user saves the position.

- **Exception**: The user enters an invalid price or size.
  - **Exception flow**: The position is not created and an error message is displayed to the user.
- **Final state**: The position is successfully created and is visible to the user.

Use case: View portfolios

- **Goal**: The user should be able to view all their portfolios, including their positions and transactions.
- **Initial state**: The user has the portfolios view open.
- **Normal flow**:

  1. The system displays all the user's portfolios including their name, currency, profit, performance, and note.
  2. The user may expand any given portfolio to see the positions belonging to the portfolio, including their instrument symbol, current size, profit, performance, break-even point, and note.
  3. The user may expand any given position to see the transactions executed within that position, including its time, size, price per unit, and note.
  4. The user may expand all portfolios and positions by pressing a single *Expand all* button.
  5. The user may collapse all portfolios and positions by pressing a single *Collapse all* button.

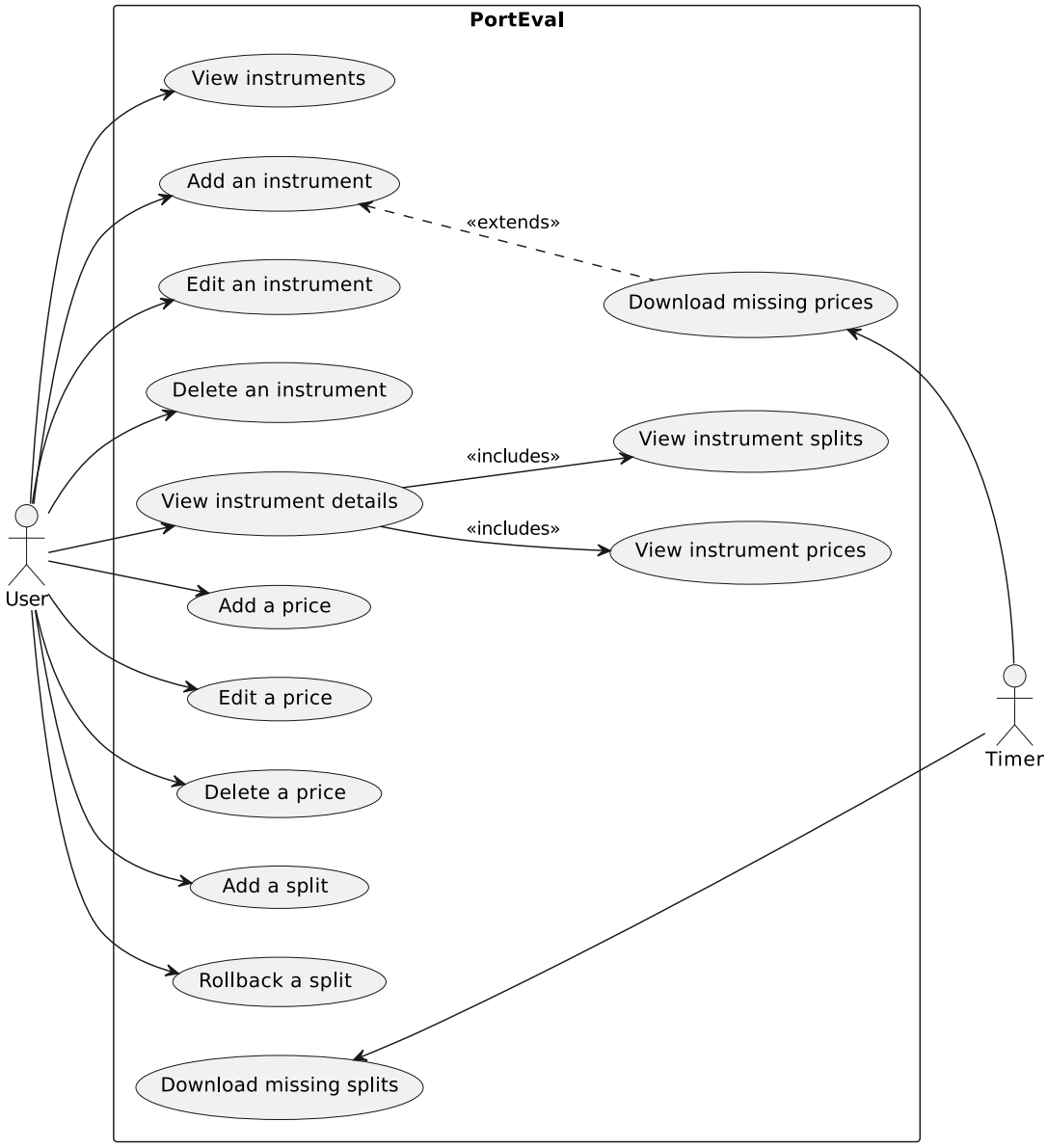

Figure 1.1: Use case diagram - instrument management

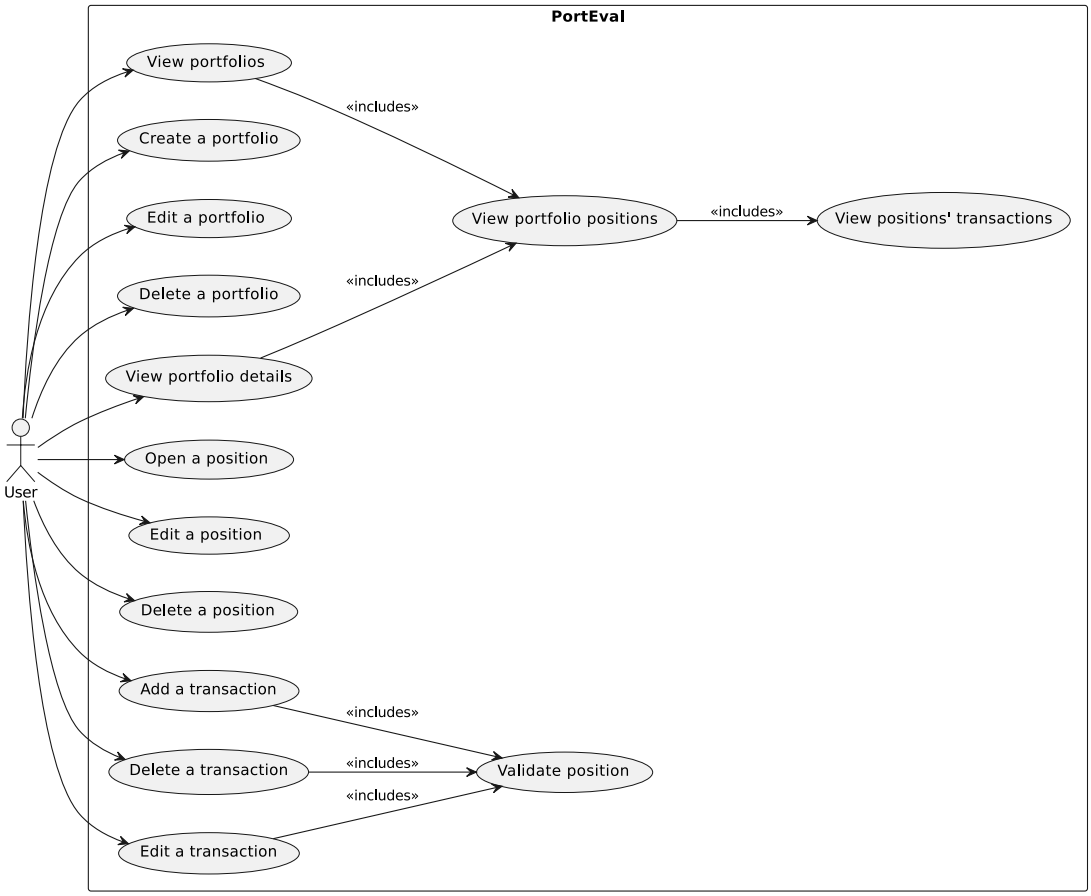

Figure 1.2: Use case diagram - portfolio management

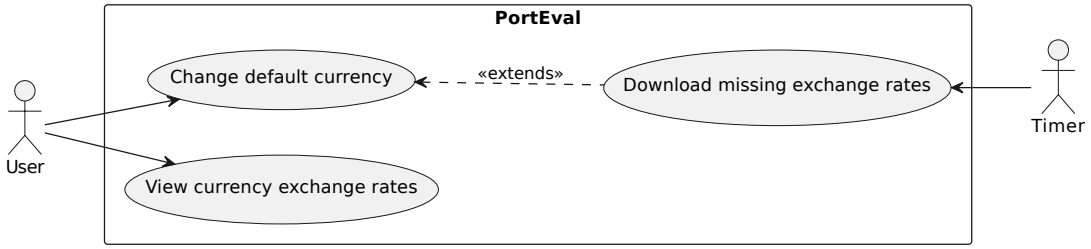

Figure 1.3: Use case diagram - currency management

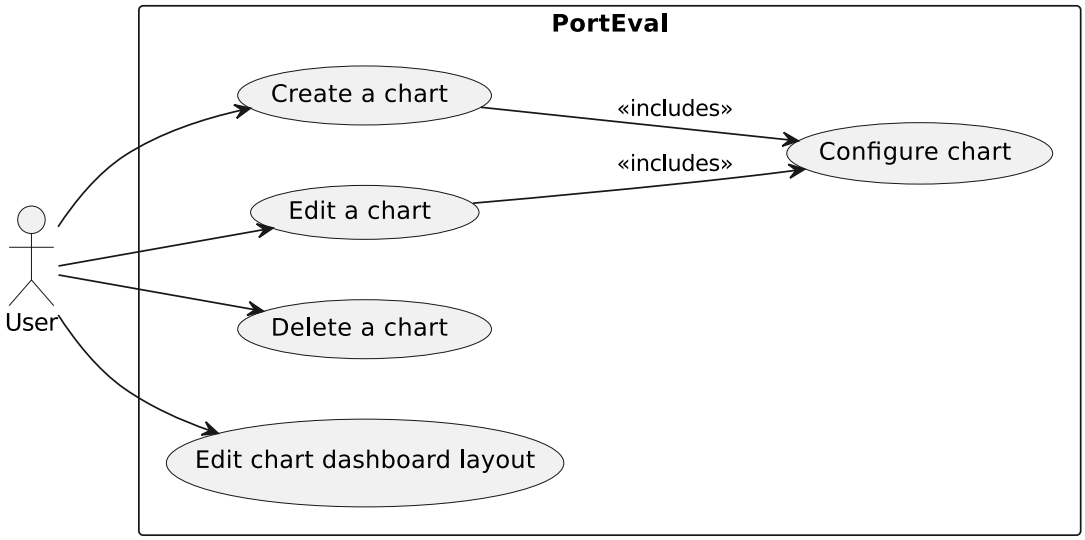

Figure 1.4: Use case diagram - chart management

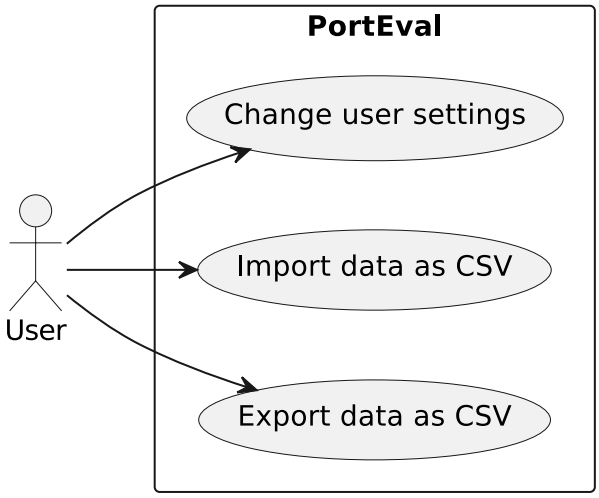

Figure 1.5: Use case diagram - application management

2. Design

In the following chapter, we describe the architecture and key decisions made during the project design phase.

2.1 Choice of technologies

In this section we examine the platforms, programming languages, and frameworks we can use to build the application.

2.1.1 Platform

One of the critical decisions which needs to be made early is the platform the application will run on. The three most common options are desktop, web, and mobile. Out of these, web offers the most versatility, such as the ability to potentially expand into a Software as a service (SaaS) model, lower hardware requirements for end-users, and easier scalability. For these reasons, the application was developed as a web application, consisting of the following layers:

- Frontend
- Backend
- Database

In the following subsections, we explore the available technologies for each of these layers.

2.1.2 Backend

There is a wide array of choices available for the backend layer. Some of the most popular combinations include Node.js with Express[1], Java with Spring[2], Python with Django[3] or Flask[4], and C# with ASP.NET Core[5] [Stack Overflow, 2022]. Due to prior experience with .NET, the extensive ecosystem of the framework, its native support for multi-threading, and high performance in common scenarios [TechEmpower, 2022], ASP.NET Core was the framework of choice for the backend layer of the application.

2.1.3 Frontend

While it would be possible to build the frontend using only vanilla JavaScript or libraries such as jQuery, this could result in longer development time, as the application is expected to require a large number of Document Object Model (DOM)

[1] https://expressjs.com/
[2] https://spring.io/
[3] https://www.djangoproject.com/
[4] https://flask.palletsprojects.com/
[5] https://learn.microsoft.com/en-us/aspnet/core/

manipulations and complex state management. For this reason, we instead only consider modern frontend frameworks, which can significantly simplify development of the application.

This presents us with a smaller pool of viable options compared to the backend layer, with the most widely used frameworks being React[6], Vue.js[7], and Angular[8] [Stack Overflow, 2022]. All three frameworks offer a large ecosystem of libraries and modules, so the decision ultimately hinges on performance and familiarity with the framework.

To compare performance between these frameworks, prototype versions of the application were developed using each. These prototypes incorporated the following functionality, which was expected to be the most performance-intensive:

- Three-level expandable table containing portfolios, positions, and transactions
- SVG (Scalable Vector Graphics) multi-line chart rendering using d3.js[9]

The performance was then measured using Time to Interactive (TTI) metric[10], which represents the time needed for the application to render useful content and be able to respond to user interactions.

The following operations were benchmarked:

- Three-level expandable table comprising 10 portfolios, 10 positions each, and 100 transactions each
  - Expand all items
  - Collapse all items
  - Expand one item
  - Collapse one item
- Chart rendering of 18 charts with 50 lines each

The benchmarks were done over 10 iterations in Google Chrome version 91 running on Windows 10, on a desktop computer with an AMD Ryzen 5 3600 CPU and 16GB of RAM.

The averaged results can be seen in Table 2.1. The frameworks produced similar results, with React slightly outperforming Vue and Angular in all operations except "collapse all". For this reason, and based on personal preference, React was chosen for the frontend layer of the application.

[6] `https://reactjs.org/`
[7] `https://vuejs.org/`
[8] `https://angular.io/`
[9] `https://d3js.org/`
[10] `https://github.com/WICG/time-to-interactive`

Prototype	Table - expand all	Table - collapse all	Table - expand one	Table - collapse one	Charts
React 17	1497.14ms	268.04ms	318.92ms	305.22ms	547.46ms
Angular 11	1818.56ms	221.74ms	347.24ms	331.38ms	819.50ms
Vue 4	1617.26ms	230.52ms	385.34ms	381.68ms	635.14ms

Table 2.1: Frontend framework performance comparison

2.1.4 Database

The application needs to be backed by a data storage to persist user and financial data.

The first decision to be made is whether to choose a relational or a non-relational (NoSQL) database. Relational databases, such as MySQL[11], PostgreSQL[12], Oracle[13], or Microsoft SQL Server[14], are based on a relational model, which organizes data into tables with well-defined relationships between them. Generally, the benefit of the relational model is its rigid structure, allowing for stricter validation of data consistency and integrity.

NoSQL databases, on the other hand, do not use a relational model. Instead, they utilize structures such as key-value pairs, graphs, or objects. Examples of such databases include MongoDB[15], Cassandra[16], Neo4J[17], or Couchbase[18]. NoSQL databases typically offer higher schema flexibility and easier horizontal scalability compared to relational databases, at the cost of lack of standardization, and in some cases, violation of ACID (atomicity, consistency, isolation, durability) properties.

As this work is expected to have a fairly consistent and well-defined data model, and currently there are no requirements necessitating horizontal scalability, we use the relational model for this application.

The second decision is the database management system (DBMS) to use. The most popular options for relational databases are MySQL, PostgreSQL, Microsoft SQL Server, and Oracle [Stack Overflow, 2022]. Due to its well-supported interoperability with ASP.NET Core, Microsoft SQL Server seemed like the natural choice for the database layer.

2.2 Conceptual model

Before discussing the design of the individual parts of the application, it would be beneficial to define the domain of the application. To achieve that, we have built a conceptual model using UML class diagrams.

[11] `https://www.mysql.com/`
[12] `https://www.postgresql.org/`
[13] `https://www.oracle.com/database/`
[14] `https://www.microsoft.com/en-us/sql-server/`
[15] `https://www.mongodb.com/`
[16] `https://cassandra.apache.org/_/index.html`
[17] `https://neo4j.com/`
[18] `https://www.couchbase.com/`

In the model, we use the following stereotypes:

- **Entity**: A mutable, long-lived concept or object, the equality of which is based on identity.
- **Interface**: A base element defining attributes which all its implementations must contain.
- **Enum**: A set of named identifiers.
- **Value object**: An immutable concept or object, the equality of which is not based on identity.

For clarity, the model will be described using two diagrams. The first diagram outlines the core model of the application, which includes entities such as portfolios and instruments. The second diagram covers the chart model. These diagrams can be seen in figures 2.1 and 2.2.

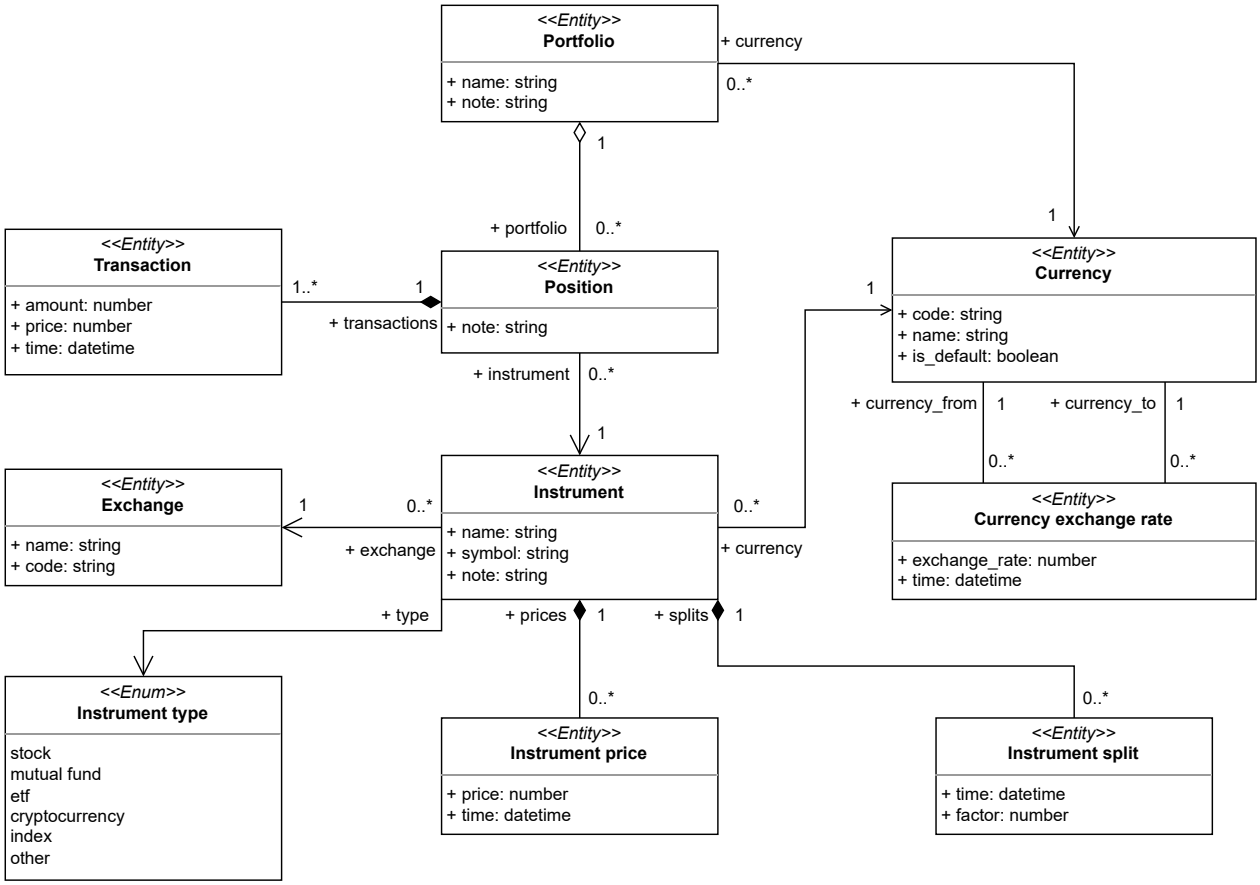

Figure 2.1: Conceptual model - core

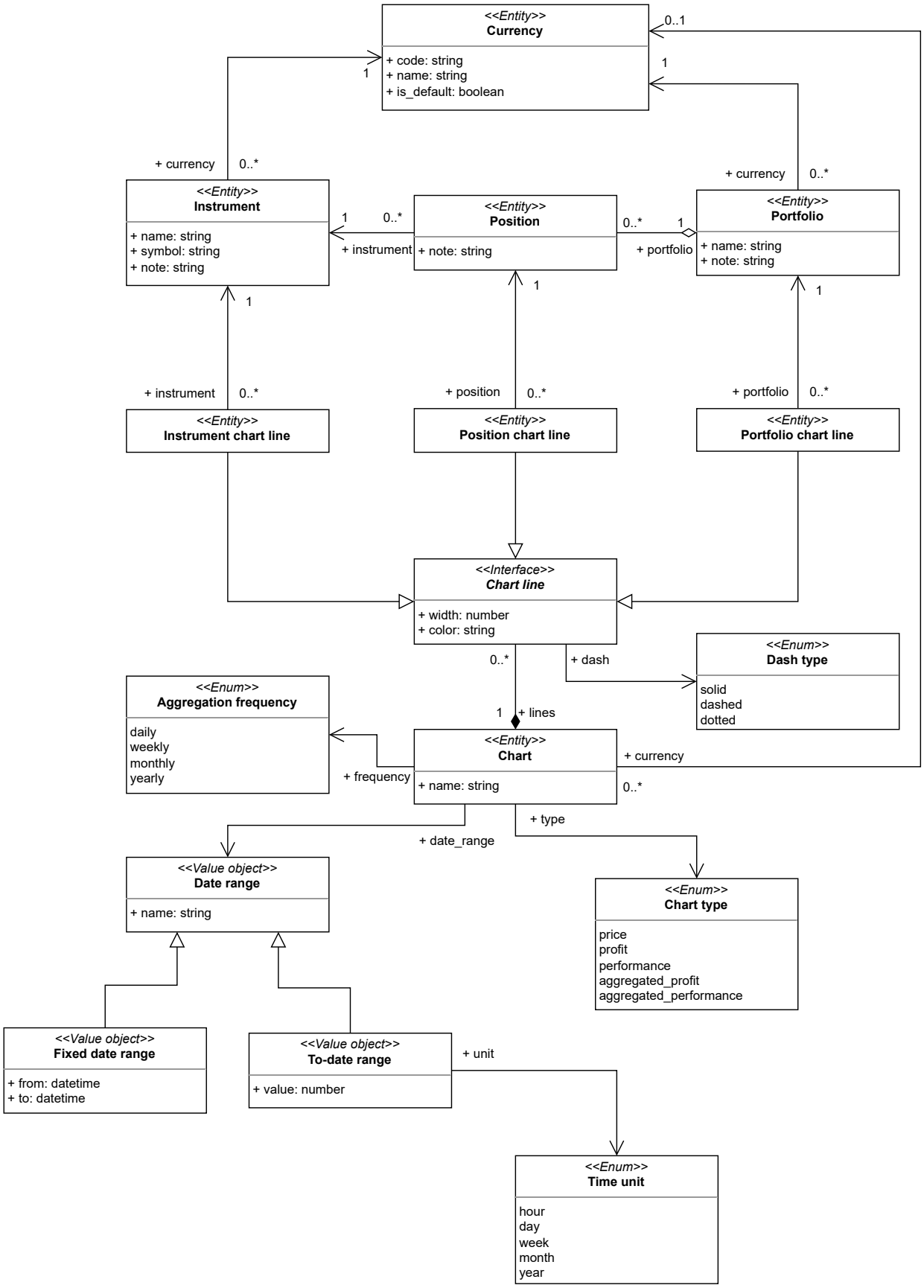

Figure 2.2: Conceptual model - charts

2.3 Backend

As is evident from the requirements of this work, the backend of the application is expected to have a large number of responsibilities, such as:

- Retrieval and storage of financial markets data.
- Management of user data, such as portfolios or charts.
- Calculation of financial metrics, and consequently, chart data generation.
- Provision of an API enabling access to the backend's functionality.

The design choices described in this section should reflect these responsibilities, while fulfilling the non-functional requirements described in the previous chapter.

2.3.1 Domain

The domain of the application was first introduced in the *Conceptual model* section. Due to the complexity of the domain and the expected complexity of application logic, it would be beneficial to separate the two, implementing the domain as a separate independent layer.

To implement the domain, we took inspiration from several principles of domain-driven design (DDD), such as [Evans, 2003]:

- **Aggregates**: Entities are contained in aggregate trees representing the scope of business invariants related to these entities, where each operation on an entity in the tree will need to be executed through the root of the tree.
- **Domain services**: Domain services implement operations spanning multiple aggregates.
- **Domain events**: Operations emit domain events, which can then be handled by the application logic.

Aggregates of this application are quite simple, typically consisting of a single entity, with two exceptions:

- Position: The domain should not allow a position's amount to fall below zero at any time. For this reason, *Transaction* entities are part of the *Position* aggregate, where *Position* is responsible for validating entered transactions.
- Chart: The domain should not contain duplicate chart lines. For this reason, *Chart line* entities are part of the *Chart* aggregate, where *Chart* is responsible for validating its chart lines.

2.3.2 Persistence

As the application is backed by an SQL Server database, the backend needs to be able to communicate with that database, ideally in the language of entities described in the previous section.

For this reason, it makes sense to use an object-relational mapping (ORM) framework to facilitate such communication. The obvious choice in the .NET ecosystem is Entity Framework (EF), which is an ORM framework developed by Microsoft. It supports LINQ[19] queries, automatic change tracking, and schema migrations. Additionally, it allows for a Code First approach, where the database schema is generated from the domain entities defined in the application, which helps us focus on the business logic of the application, rather than persistence concerns.

While relying on a fully fledged ORM for all persistence-related functionality may simplify development, it poses a performance problem, especially if we are to rely on domain aggregates for all read and write operations. It can be expected that the number of read operations will be substantially higher than the number of write operations. In this context, it might make sense to use a different method of reading data from the database, one which minimizes the overhead caused by EF or domain concerns. This approach also makes sense considering that the backend does not necessarily return domain entities to its clients. Instead, it will typically use Data Transfer Objects (DTOs), which may be comprised of data from multiple entities, or data calculated on-the-fly.

For these reasons, the read operations are instead implemented using Dapper[20], which can be described as a micro-ORM providing only the mapping between object models and relational structures. With Dapper, we can build custom SQL queries retrieving only the data necessary to fulfill a client's request.

2.3.3 Financial data sources

According to the requirements analyzed in the previous chapter, the application needs to be able to download and store instrument prices, instrument splits, and currency exchange rates. We have also identified several data sources which provide such data.

One of the important properties of the data retrieval mechanism is the ability to use any of the implemented data sources to retrieve necessary data, while possibly switching to a different one in case of failure. Additionally, according to non-functional requirement **NFR6**, data source implementations should be independent of this ability, allowing for easy replacement or extension of the existing sources.

For this reason, the data retrieval functionality can be viewed as two separate concerns:

- Data retrieval mechanism - Responsible for delegating the requests for specific types of data to data sources supporting retrieval of such data.
- Data source implementations - Responsible for providing integrations with the data sources.

[19] LINQ - Language-Integrated Query is a data querying technology integrated into .NET.
[20] `https://github.com/DapperLib/Dapper`

In .NET context, it makes sense to implement the data retrieval logic as a class library. This library should not rely on any specific data sources, instead allowing the user to plug their own implementations into the library, assuming that the implementation fulfills the interface defined by the library. This approach can be achieved using reflection.

The integrations with specific data sources can then be implemented as part of the application itself.

2.3.4 Jobs

In the previous chapter we have outlined multiple requirements concerning asynchronous data processing, such as **FR12** or **NFR5**. These requirements can be fulfilled by having a layer responsible for scheduling and executing background jobs.

To achieve this, we can use Hangfire[21], which is a .NET library providing extensive background processing functionality. One key advantage of Hangfire is its ability to persist jobs, allowing them to be monitored or reattempted even after application restart. Additionally, it guarantees that a successfully created job will run at least once.

2.3.5 API

The backend needs to provide an API for the frontend to access its data. Commonly used API types include REST (Representational state transfer), GraphQL, or SOAP (Simple Objects Access Protocol). To minimize the size of transferred data, and to simplify the communication on the frontend side, we would like to avoid using XML-based APIs such as SOAP. Furthermore, while GraphQL provides a lot of flexibility to the clients of the API, implementing it optimally on the server side is challenging, especially considering file processing requirements **FR10** and **FR11**. Therefore, we have decided to utilize a RESTful JSON API.

2.3.6 Architecture

With the decisions made in previous sections, we can now describe the architecture of the application backend. We use a variant of the layered architecture, called the Onion Architecture, first described in a series of articles by Jeffrey Palermo [2008]. In the remainder of this section, we provide a brief overview of the backend layers and their respective responsibilities and design.

Domain

The *Domain* layer implements the domain model and the domain logic, either as part of the domain entities, or as part of domain services.

Application logic

The *Application logic* layer implements functionality which is not part of the domain logic. It means that it covers the following concerns:

[21] `https://www.hangfire.io/`

1. Read and write operations on application data
2. Asynchronous background jobs
3. CSV processing
4. Financial metrics calculation
5. Chart data generation

This logic is typically exposed by *services*, where each method of a service represents a specific use case.

Infrastructure

The *Infrastructure* layer implements concerns related to communication with the database and other external sources. In the context of this application, these concerns are:

1. Definition of the database schema based on domain entities
2. Access to the database and ORM functionality
3. Implementations of external data sources

In point 2, it is important to make the distinction between EF-based and Dapper-based communication as discussed in *Persistence*. For domain entities, we utilize the *Repository* pattern [Fowler, 2005] implemented with EF. Repositories provide a collection-like interface for storage of domain entities.

For simple data retrieval which does not need to work with domain entities, we develop *queries*, which utilize Dapper with handcrafted SQL to retrieve the necessary data as optimally as possible.

It is the responsibility of the aforementioned *services* to determine whether repositories or queries should be used to fulfill a specific use case.

Presentation

In the context of the backend, the *User interface* layer can instead be described as the *Presentation* layer, which is responsible for the implementation of the API endpoints. These endpoints are implemented in *controllers*, which use functionality provided by services from the *Application logic* layer.

2.4 Frontend

The frontend of the application is expected to provide an interactive graphical user interface (GUI), allowing users to view and manipulate application data. As was established in the previous chapter, the frontend is built as a single-page application (SPA) using React.

2.4.1 User interface

This subsection describes the user interface (UI) design of the web application by presenting several mockups. These mockups do not represent the final software, rather providing a general structure of the key views of the application and their content.

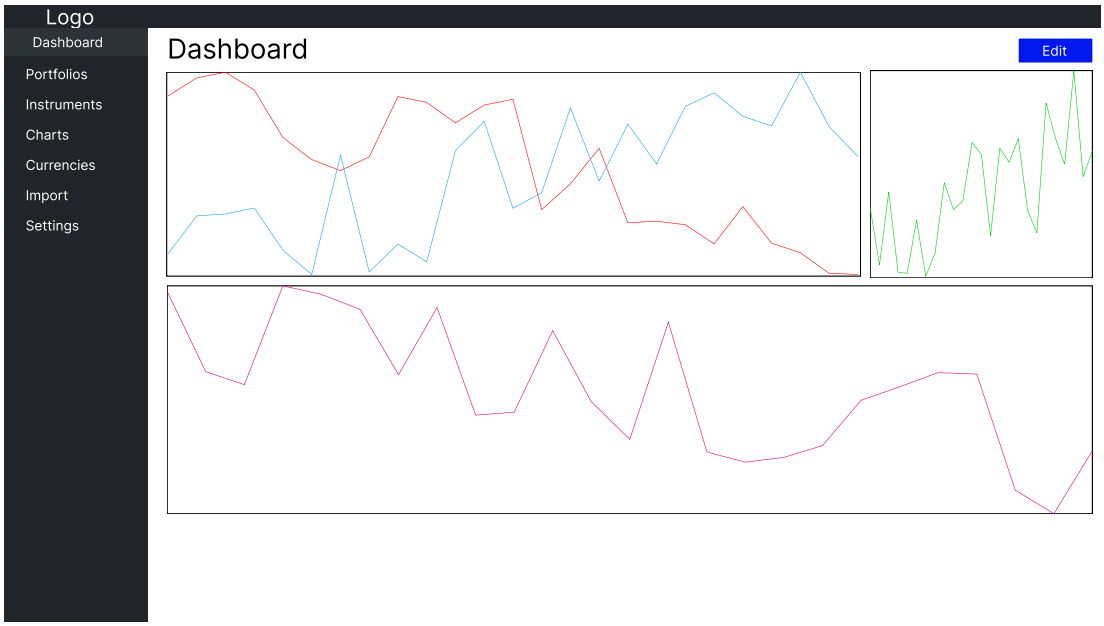

Figure 2.3: UI design - dashboard

The dashboard (Figure 2.3) is a page allowing the user to place their created charts into a custom grid. These charts can then be moved, resized, or removed from the dashboard.

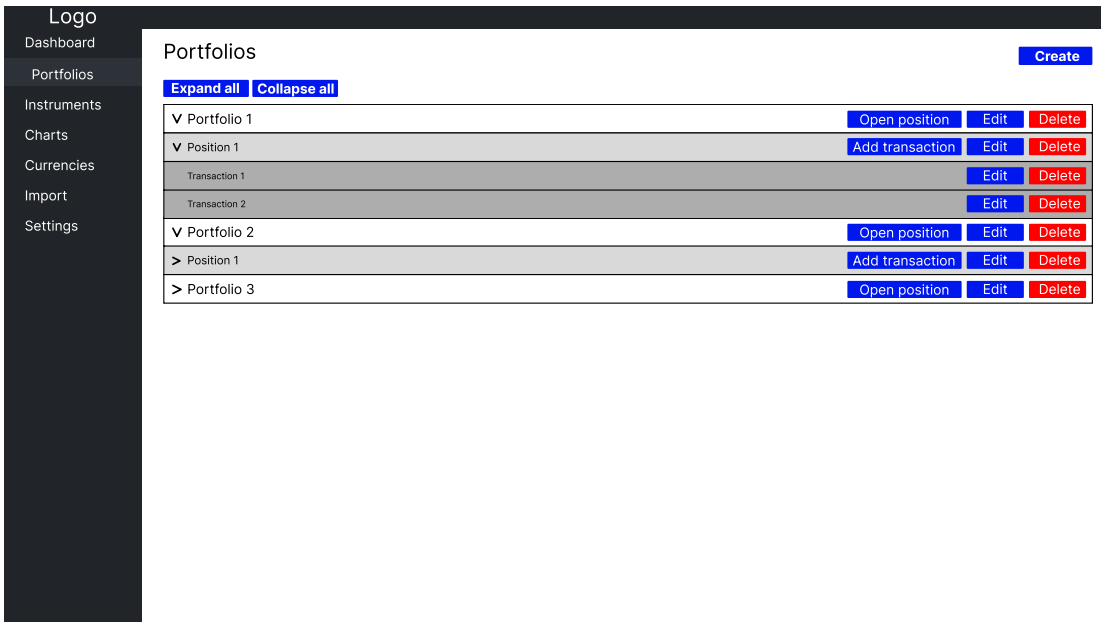

Figure 2.4: UI design - portfolios

The portfolios view (Figure 2.4) contains an expandable list of all user's portfo-

lios. When a given portfolio is expanded, it displays a list of positions contained in the portfolio, which can then be further expanded to display the position's transactions.

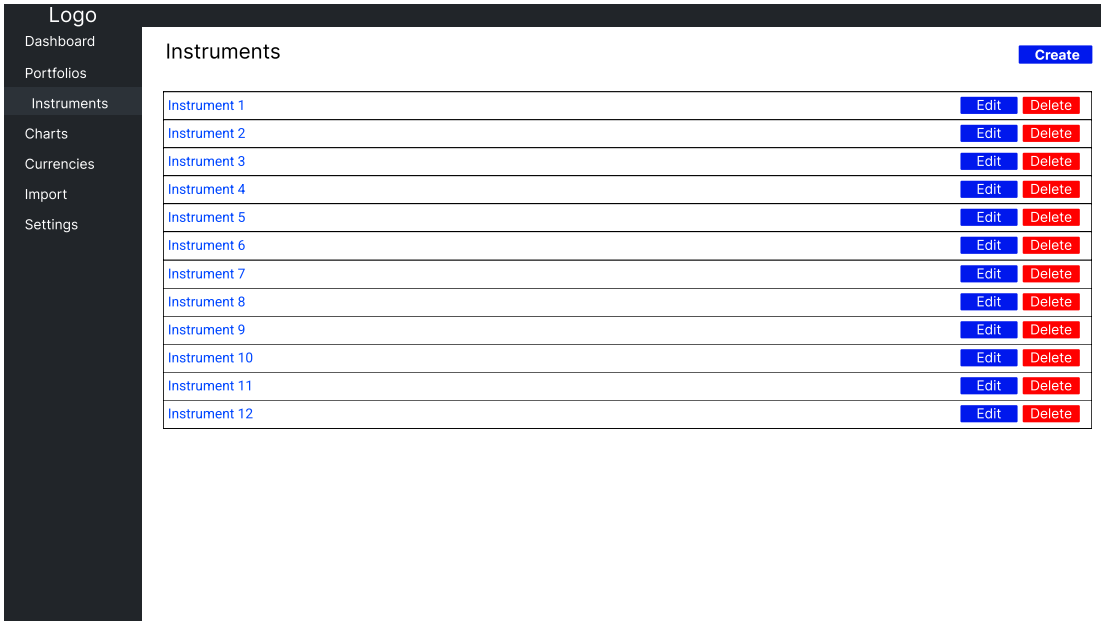

Figure 2.5: UI design - instruments

The instruments page (Figure 2.5) allows the user to see all instruments created in the application and navigate to their pages. Furthermore, this view enables the user to add new instruments to the application, or manage existing ones.

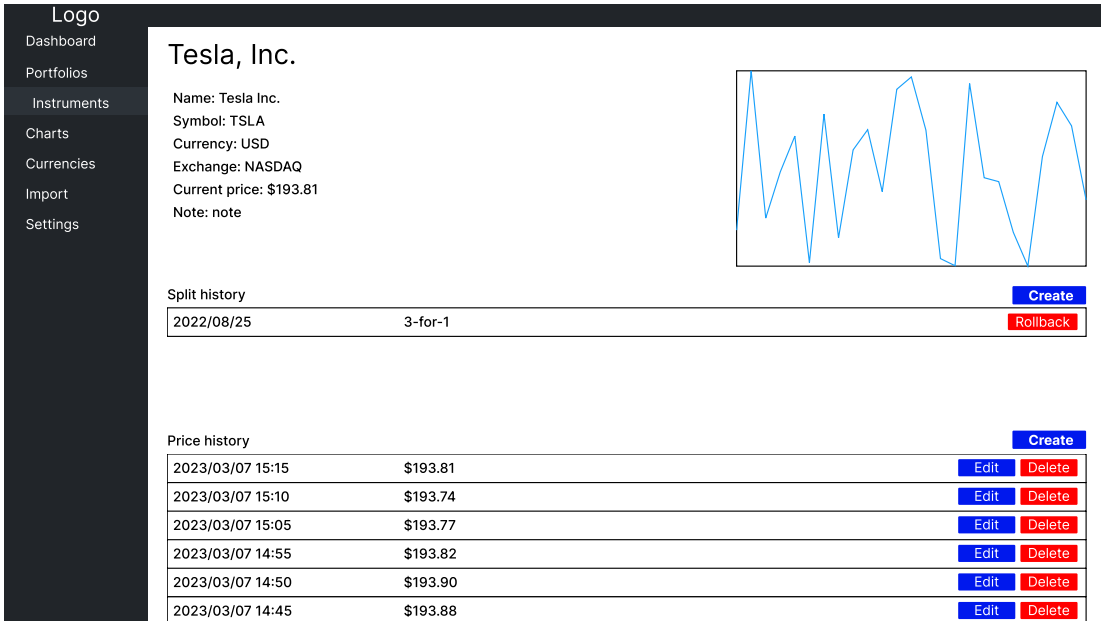

Figure 2.6: UI design - single instrument

The instrument page (Figure 2.6) contains key information about the instrument, its price chart, and allows the user to manage the instrument's splits or prices.

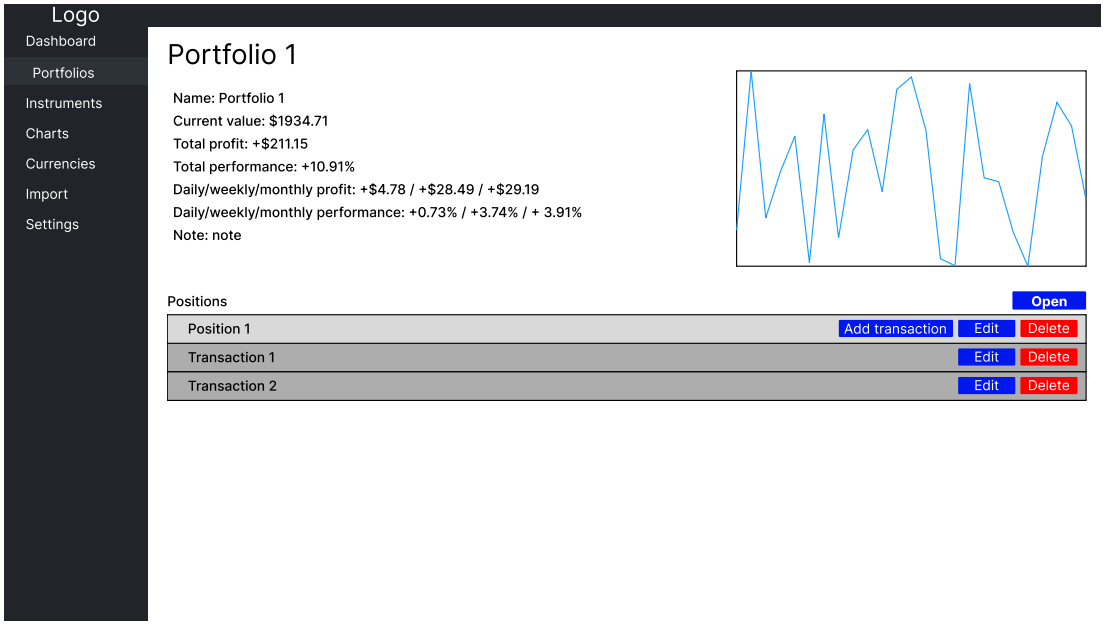

Figure 2.7: UI design - single portfolio

The portfolio page (Figure 2.7) looks similar to the instrument page, displaying information about the portfolio, its value chart, and an expandatable position table.

Figure 2.8: UI design - chart editing

The chart editing page (Figure 2.8) contains the chart, its configuration options, and a list of instruments and portfolios which can be added to the chart. To add positions, the user can press the *Positions* button on the parent portfolio, which displays a similar list of available positions.

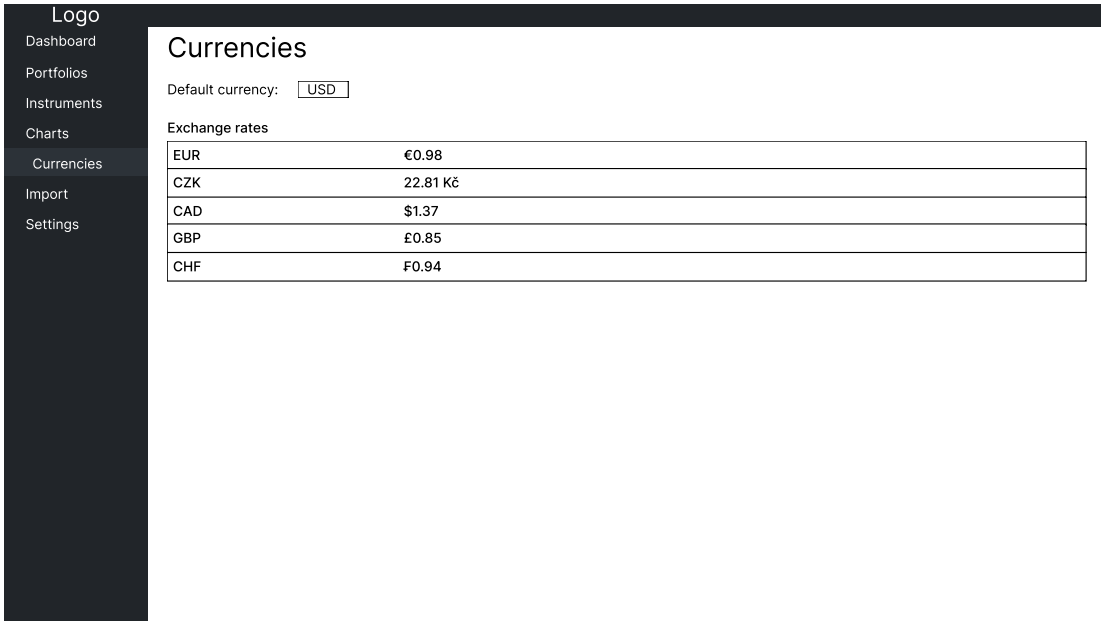

Figure 2.9: UI design - currencies

The currencies page (Figure 2.9) enables the user to change the default currency, while additionally displaying a list of current exchange rates from the selected default currency.

2.4.2 Chart rendering

One of the frontend's key responsibilities is the ability to render multi-line charts. When working with web graphics, the first decision to make is whether to use vector or raster graphics. Vector graphics are defined by paths, which allows for creation of resolution-independent graphics, while raster graphics use bitmaps, where a value is stored for each pixel of the image. For images such as line charts, vector graphics seem like a natural choice, as the final image will consist only of lines and text.

While there are numerous libraries available for SVG line chart rendering, none of them completely fulfilled the requirements of this application, performance being the common bottleneck. For this reason, a custom rendering logic was built with d3.js, which has proven itself during the prototyping phase.

2.4.3 State management

The frontend of the application needs to manage large amounts of application state, typically containing data retrieved from the backend. While it would be possible to use React component state to maintain such state, it poses problems when sharing state across multiple components, as it would need to be explicitly passed down to each component using the data. Similarly, any changes to the state would need to be passed back up using explicitly defined callbacks.

In the React ecosystem, there are two common methods of resolving these problems: the Context API[22] and Redux[23].

The Context API provides a way to share data in a tree of React components using statically defined context objects. While it seems to resolve the state management problems, there are several caveats to consider. First, any change to the context triggers a re-render of all the components subscribed to that context, even if the components do not use the affected data. This means that frequent updates to the state will significantly degrade performance of the application. Second, because the contexts are statically defined, they are not extensible, and any changes to the contexts may require changes in the code which uses them.

Redux, on the other hand, is a library inspired by the *Flux* architecture[24]. It revolves around the concept of *actions* which can be plugged into *reducers* to modify application state. Subscribers to the changed piece of state can then be re-rendered without affecting other components.

It can be expected that data managed by the backend will be frequently updated in the frontend. For this type of data, we use Redux to manage state. Nevertheless, for data specific to the frontend, the application instead utilizes React features mentioned above.

Backend integration

The frontend needs to communicate with the backend to retrieve necessary data and perform mutations on such data. While it would be possible to simply fetch the data from the backend whenever it is needed, this would result in a large number of requests for unchanged data, incurring unnecessary load on the backend.

A more effective approach would be to cache the retrieved data and reuse it until expiration. Cached data should be part of the global application state handled by Redux. To achieve this, we can use the Redux Toolkit (RTK)[25] library, which simplifies working with Redux state and offers a module for streamlined API integration with built-in caching support.

[22] `https://react.dev/learn/passing-data-deeply-with-context`
[23] `https://redux.js.org/`
[24] `https://facebook.github.io/flux/docs/in-depth-overview/`
[25] `https://redux-toolkit.js.org/`

3. Implementation

This chapter describes the key decisions made during the implementation of the application. First, we describe the development environment, listing the tools and dependencies used during implementation. Then, we describe implementations of the backend and the frontend respectively.

The source code of the application is available in a public GitHub repository[1]. Instructions on how to use the software can be found in attachment A.3.

3.1 Development environment

The backend of the application was developed using Visual Studio 2022 as an ASP.NET Core 7.0 Web API. Additionally, we used the ReSharper[2] and dotCover[3] extensions for code and test coverage analysis. To compile the application, .NET 7.0 SDK (software development kit) must be installed on the machine. In order to run the application, both .NET 7.0 and ASP.NET Core 7.0 runtimes must also be installed. Both these runtimes are provided by Microsoft as a single hosting bundle[4].

Alternatively, developers may opt to use Linux-based Docker[5] images, which are defined in Dockerfiles located in the root directory of the solution. These Dockerfiles build the application using images for the aforementioned SDK and run it in development or production environments with the appropriate runtime image.

On the other hand, the frontend was developed using Visual Studio Code as a React 17 application. To streamline bundling and the initial configuration, we used the Create React App (CRA)[6] toolchain. To build the frontend, Node.js must be installed on the machine. Similar to the backend, compilation and running of the frontend can also be accomplished using Dockerfiles located in the root directory of the application.

Further information on deployment of the application can be found in attachment A.2.

3.2 Backend

As discussed in *Design*, the backend of the application was implemented in several layers as per the Onion Architecture, with dependencies of these layers flowing inward. This was accomplished by heavy use of the dependency injection and inversion principles, taking advantage of the ASP.NET built-in Inversion of Control (IoC) container.

This section often references *injectable* functionality, which means that it was implemented as an interface/class pair, registered in the IoC container at applica-

[1] `https://github.com/simonodm/porteval`
[2] `https://www.jetbrains.com/resharper/`
[3] `https://www.jetbrains.com/dotcover/`
[4] `https://dotnet.microsoft.com/en-us/download/dotnet/7.0`
[5] `https://www.docker.com/`
[6] `https://create-react-app.dev/`

tion startup. The classes which use such functionality then receive the interfaces as constructor parameters, while the IoC container is responsible for providing the correct implementation.

3.2.1 API

The backend provides 82 RESTful endpoints, implemented in 13 controllers. Each controller covers a specific type of data located in a specific route. These routes are the following:

- `/instruments/*`
- `/instruments/{id}/prices/*`
- `/instruments/{id}/splits/*`
- `/portfolios/*`
- `/positions/*`
- `/transactions/*`
- `/currencies/*`
- `/currencies/{code}/exchange_rates/*`
- `/charts/*`
- `/dashboard/*`
- `/exchanges/*`
- `/imports/*`
- `/exports/*`

These controllers only handle the basic request/response flow, utilizing injected services for actual data processing or retrieval.

The API is documented using Swagger/OpenAPI[7]. This documentation is available at the root URL of the API.

Request validation

Before a request is processed, its body is validated according to a pre-defined set of rules. For example, the `name` field of a portfolio is configured as required, and its length is limited to 64 characters. This validation was implemented with the *FluentValidation*[8] library, which allows us to implement strongly-typed validators using a fluent API. These validators can then be executed manually, or they can be plugged into the ASP.NET Core validation pipeline, which executes the validation before the request body is available in the controller. In this work, we have opted for the second option for simplicity.

If a validation fails, a response of the following format is returned to the client:

[7] `https://swagger.io/`

[8] `https://fluentvalidation.net/`

```
{
    "errors": {
        "name": [
            "The length of 'Name' must be 64 characters or
               fewer. You entered 177 characters."
        ]
    },
    "type": "https://tools.ietf.org/html/rfc7231#section
       -6.5.1",
    "title": "One or more validation errors occurred.",
    "status": 400,
    "traceId": "00-2a3f43485b41224eac26989d7a7a7fd3-665
       ee622d068a140-00"
}
```

Error handling

There is always a possibility that an exception will occur during processing of the request. In this case, the clients will expect the API to return a valid response describing the error.

To achieve this, an exception-catching middleware was implemented. This middleware wraps the processing logic in a try-catch sequence, distinguishing between exceptions caused by invalid user input, and application exceptions. In the first case, the API returns a `400 Bad Request` response containing the exception message, while in the second case, it returns a `500 Internal Server Error` response with a generic *"An error has occurred."* message.

An example error response looks as follows:

```
{
    "statusCode": 400,
    "errorMessage": "Currency USDZ does not exist."
}
```

Notifications

To assist clients in synchronizing the displayed data with the backend data, a real-time notification mechanism was implemented. This mechanism utilizes the *SignalR*[9] framework, which allows the backend to push content to connected clients using the WebSocket protocol.

Notifications are emitted whenever backend downloads data from external sources, or whenever it finishes some background processing, such as a CSV import or a post-split price adjustment. This is achieved with a single SignalR hub, which is additionally wrapped in a notifications application service, enabling easy replacement of the underlying real-time communication mechanism.

[9] `https://dotnet.microsoft.com/en-us/apps/aspnet/signalr`

3.2.2 Services

As defined in *Design*, application services are stateless classes which implement application use cases. They are designed to be injected into controllers or other services. Examples of implemented services include `PortfolioService`, which implements read and write operations on portfolios, or `CsvExportService`, which implements conversion of data into CSV format. In total, 13 such services were implemented.

Services typically depend upon repositories or queries, which facilitate communication with the application database. These repositories and queries are implemented in the infrastructure layer. Furthermore, services also utilize injected calculators and chart data generators to perform additional processing where needed, i. e. to generate chart line data.

Each public service method returns an `OperationResponse` wrapper, which provides information on whether the operation was successful, and the operation result where applicable.

3.2.3 Calculators

The application logic layer additionally implements several *calculators*, which are injectable classes responsible for calculating financial metrics of portfolios, positions, and instruments. These calculators typically accept a list of prices and/or transactions and return a single numeric value representing the metric.

A relevant example of such a calculator is `PositionPerformanceCalculator`, which evaluates the performance of a set of positions using the calculation discussed in *Analysis*. It does so by plugging the positions' transactions into formula 1.2 as cash flows, where purchases represent negative cash flows, and sales represent positive cash flows. Additionally, to account for the current size of the positions and the underlying instruments' current price, the calculation adds additional positive cash flows simulating the complete sale of the positions at their instruments' current price. As an example, assuming a position with the following transactions:

- January 1, 2023 - 10 units purchased at $100
- February 1, 2023 - 5 units sold at $110
- March 1, 2023 - 3 units sold at $115

And assuming that the instrument's current price is $120, the calculation performed on April 1, 2023 would build the following equation:

$$-10 \times 100r^{90} + 5 \times 110r^{59} + 3 \times 115r^{31} + 2 \times 120r^{0} = 0$$

This equation is encapsulated in a `PolynomialEquation` instance, which implements the operation of solving the equation using Newton's method. To do so, it requires an initial guess to be provided. The initial guess for the algorithm is calculated as follows:

$$\sqrt[N]{\frac{S+E}{P}}$$

where:

- S - total sell value (the sum of all sale transactions in the period)
- E - end market value (position size at the end of the period multiplied by the instrument's current price)
- P - total purchase value (the sum of all purchase transactions in the period)
- N - total number of subperiods in IRR calculation

This formula represents a simple estimate of the position's performance in one subperiod. It was arrived at somewhat experimentally, resulting in the least number of iterations of Newton's method compared to several attempted alternatives.

Chart data generators

One of the more challenging responsibilities for the application's backend is chart data generation. According to non-functional requirement **NFR8**, the application must be able to render 10 chart lines in under two seconds, irrespective of the type of the chart or the entities depicted by its lines. Although the implementation of the chart rendering mechanism in the frontend also affects this performance, the aggregation and processing of the data on the backend can be expected to play a much more significant role. This is because the backend is responsible for several performance-intensive tasks, such as retrieving necessary data, aggregating it according to the chart's frequency, calculating metrics, and in some cases, converting currency.

To fulfill these requirements while maximizing the performance of the data generation, our goal was to minimize the number of database queries performed, while also minimizing the time needed to process retrieved data. The implemented algorithm for a single chart line can be described as follows:

1. Retrieve all the necessary data sorted by time and store it as in-memory collections. This data might include instrument prices, position transactions, and currency exchange rates, depending on the configuration of the chart and the financial entity represented by the line.
2. Split the chart time range into sub-intervals `(T1, T2)` based on the frequency of the chart. Each such sub-interval will ultimately represent a single point of the chart line.
3. Define an iterator for each collection from step 1.
4. For each sub-interval `(T1, T2)` from step 2, do the following:
   (a) Advance each iterator from step 3 until it points to the last element with time earlier than `T2`.
   (b) Calculate the value of the chart point at `T2` based on visited elements using an appropriate injected calculator.
5. Return the chart line as a collection of `(time, value)` pairs.

This algorithm allows for a fixed number of database queries per line, and it works in linear time with regards to the sum of sizes of all retrieved collections.

3.2.4 CSV processing

CSV import and export can generally be described as two separate concerns: (de-)serialization of CSV files and bulk data processing. To facilitate conversions from and to CSV format, we used the *CsvHelper*[10] library, which enabled us to easily convert between application DTOs and their CSV representation.

CSV export logic then simply retrieves the necessary data using the appropriate service, and converts that data to CSV using `CsvExportService`, which encapsulates conversion to CSV implemented using CsvHelper.

CSV imports are slightly more complex, as they require validation and involve data mutations. However, the core idea remains the same: the user's CSV file is first deserialized into application DTOs, which are then validated and imported using appropriate application services.

3.2.5 Background jobs

As was established in the previous chapter, the application uses asynchronous background jobs to handle various long-running data processing needs. Specifically, based on the requirements from *Analysis*, nine such jobs have been implemented, further described in Table 3.1.

Recurring jobs are scheduled at application startup using cron-like syntax, while one-off jobs are typically scheduled by an appropriate domain event handler. For example, the initial price download job is triggered during processing of the *Instrument created* event emitted by the domain layer. The jobs are then executed by Hangfire at an appropriate time.

Several jobs are responsible for bulk insertion of data into the database. This has proven to be fairly slow with out of the box EF functionality, even with several optimizations in place (such as disabling automatic change tracking). Ultimately, we opted for the *EF Core BulkExtensions*[11] library, which provides a set of extension methods for optimized bulk operations.

3.2.6 Data fetching

As was discussed in *Design*, the data fetching mechanism was implemented in two parts. The first part is a library enabling retrieval of homogenous data from different data sources. The second part is the implementations of individual data sources. These implementations are contained in the infrastructure layer and are designed to work with the aforementioned library.

The library was designed to not make any assumptions about the type of data it is going to retrieve, delegating this responsibility to data source implementations. Instead, it only implements two core operations: registration of data sources and request processing.

To enable arbitrary data sources to be registered in the library, we had to implement certain constraints. In particular, the data sources need to be implemented as classes inheriting from the `DataSource` abstract class provided by the library. This abstract class provides its implementations with a configuration

[10] `https://joshclose.github.io/CsvHelper/`

[11] `https://github.com/borisdj/EFCore.BulkExtensions`

Job	Description	Trigger
Data import	Processes a user-initiated CSV data import	A CSV file is uploaded by the user.
Import cleanup	Deletes import entries and files older than 24 hours.	Every 24 hours or on application restart.
Price cleanup	Deletes overabundant prices to maintain appropriate intervals.	Every 24 hours or on application restart.
Initial price download	Downloads the complete price history for an instrument.	An instrument is created by the user.
Missing price download	Downloads missing instrument prices.	Every 24 hours or on application restart.
Latest price download	Downloads the current prices of all created instruments.	Every 5 minutes.
Missing exchange rates download	Downloads missing exchange rates from the default currency.	Every 24 hours or on application restart.
Split download	Downloads unaccounted-for share splits for all instruments.	Every 24 hours or on application restart.
Split adjustment	Retrospectively adjusts instrument prices and transactions for splits.	An instrument split is created or rolled back.

Table 3.1: Implemented background jobs

object and an HTTP client. The ability to process requests for specific types of data can then implemented as methods with a `RequestProcessor` attribute.

When the library receives a request for a certain type of data, it first utilizes reflection to discover all registered data sources containing a method with a matching `RequestProcessor` attribute. It then attempts to use the first registered data source to fulfill the request. If this succeeds, the response is returned to the client of the library and request processing is terminated. Otherwise, the library starts distributing the request among all supported data sources. In case of network failure, each such data source, including the originally attempted one, is retried according to the retry policy specified during the initialization of the library.

This request handling process is completely parallelized, resulting in each data source being attempted independently of others. If any of the attempted data sources succeed, processing of all other sources is immediately terminated.

The public interface of the library is defined in a single *Facade*, which is configured and injected into the application during startup as a *Singleton* [Erich et al., 1994].

3.3 Frontend

The frontend was implemented in TypeScript as a React SPA, utilizing modern React development practices and tools. Among other things, this means that the application is composed of various *functional components*, which internally use *hooks* for state and lifecycle management. Functional components are simply JavaScript functions which return markup, while hooks are functions prefixed with `use` which can only be called at the top level of a component or another hook.

3.3.1 Routing

As the frontend is an SPA, routing was implemented on JavaScript level, utilizing the commonly used *react-router*[12] package. As was hinted in the *User interface* section, the following routes were implemented:

- `/[dashboard]` displays the user-configured dashboard, possibly containing charts created by the user.
- `/portfolios` displays a list of user portfolios, positions, and transactions.
- `/portfolios/{id}` displays portfolio details.
- `/instruments` displays a list of user instruments.
- `/instruments/{id}` displays instrument details.
- `/charts` displays a list of charts.
- `/charts/view/[{id}]` displays a full-page view of a chart.
- `/currencies` displays currency exchange rates and allows the user to change the default currency.
- `/import` allows the user to export or import data in CSV format.
- `/settings` allows the user to change their formatting settings.

3.3.2 Styling

While we could style the application using custom CSS (Cascading Style Sheets) only, it is easier to use a CSS framework. For this reason, we used *Bootstrap*[13], specifically the *react-bootstrap*[14] package, which provides a set of React components corresponding to components defined by Bootstrap.

This approach is enough to cover most of the application's styling needs. Nonetheless, for rare scenarios requiring a more customized approach, we utilized vanilla CSS stylesheets. Although usage of CSS preprocessors like Sass or LESS was also considered, they were ultimately unnecessary.

[12] `https://reactrouter.com/`
[13] `https://getbootstrap.com/`
[14] `https://react-bootstrap.github.io/`

3.3.3 Dashboard

According to requirements **FR8** and **NFR9**, the application should allow the user to drag-and-drop charts into a custom grid, where the charts can be moved and resized. As was further discussed in *Use cases* and *User interface* sections, this grid should be contained on a separate dashboard page.

To achieve this, we have utilized the *react-grid-layout*[15] package, which provides a responsive grid system with dragging functionality. The implemented grid is 6 columns wide on desktop screens, compacting itself down to 1 column on smaller screens.

3.3.4 Charts

As was discussed in *Design*, chart rendering was implemented using custom logic built with d3.js. This logic is encapsulated in the `SVGLineChart` JavaScript class, which provides a fluent API to configure the chart and append it to a DOM container.

These charts were implemented to be as configurable as possible, with particular focus on line styling, label formatting, and tooltip contents. Furthermore, according to requirement **FR7**, the chart lines render indicators for executed transactions. This functionality was achieved by enabling the client to set up a custom rendering callback which is called for each data point of the chart. The callback provided in the application renders a small `+` or `-` SVG icon on the line point if any transactions were executed since the time represented by the previous data point.

A similar approach was used for rendering tooltips. By default, tooltips display the time of the data point and the values of all the chart lines at that point. This can be further configured by a separate callback, which allows the client to append additional information to the tooltip. This is used in the application to render details about transactions executed at that data point. The `SVGLineChart` class additionally memoizes internal tooltip rendering and the configured callback to prevent recalculation on each mouse movement.

When manipulating the DOM directly, one needs to be careful to avoid conflicts with the virtual DOM managed by React. Specifically, issues may arise if custom DOM logic will be applied to elements which are tracked by React. To circumvent this, the `SVGLineChart` class does not do any mutations on existing DOM elements, simply appending the chart to a pre-defined container. For this approach to work, the container's size must be known in advance, with possible resizing of the container triggering a re-render of the chart. This is encapsulated in the React `LineChart` component, which is responsible for creating the container, initializing the `SVGLineChart` instance, appending it to the container, and registering event handlers.

3.3.5 Tables

The application utilizes several tables to display various data sets to the user, such as portfolios, instrument prices, currency exchange rates, and so forth. Many of

[15] `https://github.com/react-grid-layout/react-grid-layout`

these tables were designed with additional requirements in mind, such as the ability to expand individual rows, sort the table by a specific column, group columns, or display a more compact version of the table on smaller screen sizes.

To achieve this, a generic strongly-typed `DataTable` component was implemented, supporting all of the functionalities above. One of the key features of this component is the ability to declaratively define the columns of the table, mapping them to the fields of the underlying collection of objects. This collection is then provided to the component as props.

Internally, `DataTable` is backed by the *react-table*[16] package, which provides hook-based utilities to build data tables. *react-table* is a headless library, which means that it does not handle any visual aspects of these tables, such as specific markup or styles. Instead, it is responsible for building the contents of the table, while additionally providing sorting and expansion functionality. The ability to expand and collapse all rows as discussed in use case analysis is then implemented on top of this library using custom JavaScript events.

3.3.6 Backend communication

As was established in *Design*, communication with the backend was implemented with RTK, specifically its RTK Query submodule. This submodule allows the user to define various *endpoints*, which implement queries or mutations performed against a web API. These queries and mutations are then available to the application as automatically generated React hooks.

RTK Query is also responsible for caching and invalidation of query responses. Each query response is automatically cached in the underlying Redux store. Invalidation is then achieved primarily using *tags*, which are string identifiers that can be provided by queries. The application can then invalidate these tags either manually or after a mutation, forcing RTK to re-fetch affected queries when they are needed.

RTK endpoints were defined for most API endpoints provided by the backend. Typically these endpoints are fairly simple, representing a single HTTP request to the backend. However, some of them required additional custom logic, such as pagination or aggregation of data from multiple API endpoints.

Furthermore, the frontend connects to the SignalR notification endpoint provided by the backend. Whenever a notification representing new data availability is emitted by the backend, RTK tags are invalidated, forcing the frontend to refresh its data.

[16] `https://react-table-v7.tanstack.com/`

4. Implementation Details

The following chapter contains the technical description of the implementation, detailing the structure of the application's source code and the key components of individual layers. Individual classes and other low-level details are then documented in the `docs/` folder of the electronic attachment (Attachment A.1).

4.1 Backend

The backend of the application is implemented in a single Visual Studio solution, with different concerns further separated into different projects of that solution.

4.1.1 Project structure

The solution consists of the following projects:

- `PortEval.Application` is the entry point of the backend. This project implements the API, controllers, middleware, and configures the application's IoC container.
- `PortEval.Application.Model` implements the application model, which includes request and response models, DTOs, and validators.
- `PortEval.Application.Core` implements the application logic, which includes services, calculators, background jobs, and domain event handlers.
- `PortEval.DataFetcher` implements the data fetcher library.
- `PortEval.Domain` implements the domain layer of the application.
- `PortEval.Infrastructure` implements the infrastructure layer, which includes repositories, queries, Code First configurations, database migrations, and integrations with external data sources.
- `PortEval.Tests.Unit` implements unit tests.
- `PortEval.Tests.Integration` implements integration tests.

The following sections describe the implementation details of each of these projects.

4.1.2 PortEval.Application

`PortEval.Application` is an ASP.NET Core Web API project which is responsible for configuring the application and starting the Kestrel web server. The entry point of this project is the `Main` method of the `Program` class. This method initializes the application host and configures it using the `Configure` and `ConfigureServices` methods of the `Startup` class. These methods are responsible for setting up the API and registering dependencies in the application's IoC container.

- `Extensions/` folder implements various helper extension methods, which are typically extensions related to application configuration and registration of dependencies.
- `Controllers/` folder implements the API controller classes.
- `Static/` directory contains data which is seeded into the database at first startup, specifically currency and stock exchange information.
- `Middleware/` directory contains custom ASP.NET middleware, specifically the exception-catching middleware first described in *Implementation*.

Controllers

Specific API endpoints are implemented as methods of controller classes from the `Controllers/` directory, each with an attribute defining the route and HTTP method of the endpoint. These controllers typically depend upon service interfaces from the `PortEval.Application.Core` project.

Furthermore, all controllers inherit from a `PortEvalControllerBase` abstract class, which provides methods allowing conversion of `OperationResponse` returned by services to `ActionResult` representing the HTTP response returned to the client. This allows the endpoint implementations to be fairly compact, as in the following example of a method implementing the retrieval of a single instrument:

```
private readonly IInstrumentService _instrumentService;

// GET instruments/1
[HttpGet("{id}")]
public async Task<ActionResult<InstrumentDto>>
    GetInstrument(int id)
{
    OperationResponse<InstrumentDto> response = await
        _instrumentService.GetInstrumentAsync(id);
    return GenerateActionResult(response);
}
```

4.1.3 PortEval.Application.Model

`PortEval.Application.Model` is a class library implementing the application logic models and several related concerns, such as validation and serialization.

- `DTOs/` directory implements the application's DTOs.

  These DTOs are then used to generate responses to clients, or to transfer data between different layers of the application. They are implemented as simple objects containing no behavior, instead only providing a set of public properties.

  The `Converters/` subdirectory additionally contains several converters, which handle custom serialization of application models into JSON or text representations. This is needed for types such as `Color`, which is serialized

into an RGB string for JSON responses. This also includes CsvHelper class maps, which handle conversion between CSV and DTOs and are located in the `ClassMaps/` subdirectory.

- `Validators/` folder implements application model validators using the FluentValidation library.
- `FinancialDataFetcher/` directory implements models which are used as response types in the data source implementations, such as `PricePoint` representing a single externally retrieved price, or `ExchangeRates` representing a collection of exchange rates from a single currency at a certain time.
- `QueryParams/` folder contains models for query parameter deserialization, such as `PaginationParams`, which encapsulates `page` and `limit` parameters used for paginated endpoints.

4.1.4 PortEval.Application.Core

`PortEval.Application.Core` is a class library implementing the application logic. This project contains the majority of application's code, as it is responsible for implementation of services, background jobs, calculators, domain event handlers, and other important concerns.

- `BackgroundJobs/` directory contains implementations of background jobs.
- `Common/` directory contains various functionality which does not belong to a single service or background job.

  This functionality includes concerns such as calculators, chart data generators, and CSV import processing.
- `DomainEventHandlers/` directory contains domain event handlers.
- `Extensions/` directory contains extension methods used in the application logic layer.
- `Hubs/` directory contains the definition of the notifications SignalR hub.
- `Interfaces/` directory contains various interfaces used for dependency injection.

  Most of these interfaces are implemented by classes in this project, however, implementations of several of them are a responsibility of the infrastructure layer. These implementations will be discussed further in the `PortEval.Infrastructure` project.
- `Services/` directory contains implementations of application services.

Background jobs

Background job implementations are located in the `BackgroundJobs/` directory. Most of these implementations are fairly simple and self-sustaining, exposing only a single `Run` method, which acts as the entry point of the job.

Jobs which are responsible for downloading data from external sources typically utilize the injected `IFinancialFetcher` instance, which is implemented in the infrastructure layer. Instrument price retrieval jobs `LatestPricesFetchJob`, `MissingPricesFetchJob`, and `InitialPriceFetchJob` additionally inherit from the `InstrumentPriceFetchJobBase` abstract class. This class provides its inheriting classes with several wrapper methods over the injected `IFinancialFetcher` instance.

Calculators

Financial metric calculators are implemented in the `Common/Calculators/` directory. These calculators are simple stateless classes, typically providing only a single method to calculate the metric, as in the following implementation of a position break-even point calculator:

```
public class PositionBreakEvenPointCalculator :
    IPositionBreakEvenPointCalculator
{
    public decimal CalculatePositionBreakEvenPoint(
        IEnumerable<TransactionDto> transactions
    ) {
        decimal realizedProfit = 0m;
        decimal positionAmount = 0m;

        foreach (var transaction in transactions)
        {
            realizedProfit += transaction.Amount *
                transaction.Price;
            positionAmount += transaction.Amount;
        }

        var bep = positionAmount != 0
            ? realizedProfit / positionAmount
            : 0;
        return bep;
    }
}
```

Chart data generators

Chart data generators are responsible for building a chart line based on the chart's configuration and the underlying financial entity's data. These generators are located in the `Common/ChartDataGenerators/` directory.

These generators provide methods to generate price, profit, performance, aggregated profit, and aggregated profit lines. Internally, they use injected calculators to determine the value of each chart point.

As discussed in *Implementation*, these generators accept multiple collections of financial data. These collections are then plugged into a *range data generator*, which is responsible for aggregating necessary data for each chart point based on data from the provided collections. Range data generators act as iterators themselves, providing a `GetNextRangeData` method to retrieve data for the next chart point, and an `IsFinished` method to determine whether there is any more data to be generated.

For example, assuming an input of the following instrument prices:

1. `2023-01-01`: $100.00
2. `2023-01-02`: $110.00
3. `2023-01-03`: $120.00
4. `2023-01-04`: $130.00
5. `2023-01-05`: $140.00

And the following chart point intervals:

1. `2023-01-01` - `2023-01-03`
2. `2023-01-03` - `2023-01-05`

`InstrumentRangeDataGenerator` would behave as follows:

1. First call to `GetNextRangeData`: range from `2023-01-01` to `2023-01-03`, price at range start is $100.00, price at range end is $120.00.
2. Second call to `GetNextRangeData`: range from `2023-01-03` to `2023-01-05`, price at range start is $120.00, price at range end is $140.00.
3. Third and subsequent calls to `GetNextRangeData`: return nothing

The caller can use this data to calculate needed values for each chart point.

Domain event handlers

Side effects of domain events are implemented in domain event handlers, which can be found in the `DomainEventHandlers/` directory. These handlers are typically responsible for starting a background job using the `IBackgroundJobClient` interface provided by Hangfire.

The process of handling domain events is facilitated by the *MediatR*[1] library, which exposes two interfaces: `INotification`, which represents a message to be processed, and `INotificationHandler<TNotification>`, which defines a `Handle(INotification)` method. This poses a problem, as domain events are defined by the domain layer, and having them implement the `INotification` interface would result in a violation of separation of concerns.

To circumvent this, the `DomainEventNotificationAdapter<TEvent>` wrapper class was created, which contains a reference to the domain event while

[1] `https://github.com/jbogard/MediatR`

implementing the `INotification` interface. Specific handler implementations then accept a parameter of this type, which fulfills the contract defined by `INotificationHandler<TNotification>`, and allows these handlers to be registered in MediatR.

The adapter class is implemented as follows:

```
public class DomainEventNotificationAdapter<T> :
    INotification where T : IDomainEvent
{
    public T DomainEvent { get; private set; }

    public DomainEventNotificationAdapter(T domainEvent)
    {
        DomainEvent = domainEvent;
    }
}
```

Bulk import

Functionality related to processing of bulk data imports can be found in the `Common/BulkImport/` directory. The `ImportProcessor` generic abstract class provides an `ImportRecords` method, which imports the provided records into the system, while creating a log entry for each one.

Classes derived from `ImportProcessor` are then expected to implement the `ProcessItem` method, which is responsible for importing a single record.

This functionality is used by the CSV processing mechanism implemented in the `DataImportJob` background job.

Services

Services are classes located in the `Services/` directory and they implement various application use cases.

These services typically utilize injected repository and query interfaces to read and write data to the application's database.

4.1.5 PortEval.DataFetcher

`PortEval.DataFetcher` implements the data fetcher library as discussed in *Implementation*. This library's functionality is exposed in the `DataFetcher` class, which implements behavior related to registration of data sources and request processing. The exact mechanisms will be described in the following subsections.

Implementation of data sources

The library does not implement any data sources itself, instead delegating it to the infrastructure layer. Implemented data sources must be derived from the `DataSource` abstract class, which provides the following protected properties:

- `Configuration` - contains credentials of the data source
- `HttpClient` - the HTTP client to be used by the data source

These properties are set using reflection when the data source is first registered with the library.

Each implemented data source is expected to provide methods enabling processing of specific request types returning specific result types. The exact way these methods are implemented is left to the client, however, they should fulfill the following constraints to be successfully detected and called:

- have the `[RequestProcessor(typeof(TRequest), typeof(TResult))` attribute.
- accept a single argument of type `TRequest`, where `TRequest` implements the `IRequest` interface provided by the library
- return a `Response<TResult>` or `Task<Response<TResult>>` instance, where `TResult` is the type of the fetch operation result

The data source class itself has only one constraint: it must have a parameterless constructor.

See the following example of a valid data source implementation:

```
public class WeatherRequest : IRequest {
    public string City { get; set; }
    public string CountryCode { get; set; }
    public DateTime Time { get; set; }
}

public class WeatherResponse {
    public decimal CelsiusTemperature { get; set; }
    public decimal ChanceOfRain { get; set; }
    ...
}

public class WeatherDataSource : DataSource {
    [RequestProcessor(typeof(WeatherRequest),
       typeof(WeatherResponse))]
    public async Task<Response<WeatherResponse>>
       GetWeather(WeatherRequest request) {
        WeatherResponse data = /* retrieve from somewhere,
           e. g. an API */;

        return new Response<WeatherResponse> {
            StatusCode = StatusCode.Ok,
            Result = data
        };
    }
}
```

Registration of data sources

The data sources are registered in the `DataFetcher` instance. For this, it provides a `RegisterDataSource<TDataSource>()` generic method, where `TDataSource` is the type of the data source. This method accepts one optional parameter, which is an instance of `DataSourceConfiguration`. Following up on the example from the previous subsection, the `WeatherDataSource` can be registered as follows:

```
IDataFetcher dataFetcher = new DataFetcher();
dataFetcher.RegisterDataSource<WeatherDataSource>(new
    DataSourceConfiguration {
     Credentials = new DataSourceCredentials {
          Token = "api-refresh-token",
          Username = "client-id",
          Password = "client-secret"
     }
});
```

Request handling

Requests are then processed using the `ProcessRequest<TRequest, TResult>()` method on the `DataFetcher` instance. This method will attempt to find all registered data sources which contain a processing method for `TRequest` and `TResult` request/result combination, after which it will delegate the processing of the request to the `RequestHandler<TRequest, TResult>` generic class. Internally, this class encapsulates communication with each provided data source in a `RetryableAsyncJob` instance, which implements retry functionality on an asynchronous operation according to the provided `RetryPolicy`.

Following up on our running example, the `ProcessRequest` method can be used as follows:

```
var request = new WeatherRequest {
  City = "Prague",
  CountryCode = "CZ",
  Time = DateTime.UtcNow
};
var response = await dataFetcher.ProcessRequest<
  WeatherRequest,
  WeatherResponse
>(request);
if(response.StatusCode == StatusCode.Ok) {
  var data = response.Result;
  Console.WriteLine(data.CelsiusTemperature);
}
```

4.1.6 PortEval.Domain

`PortEval.Domain` implements the domain layer of the application.

- `Events/` directory contains domain events. These events are simple data objects, only containing enough information to handle the event. All events

implement the `IDomainEvent` interface.

- `Exceptions/` directory contains domain exceptions. All domain exceptions inherit from the `PortEvalException` abstract class.

- `Models/` directory contains the domain model. This model consists of entities, value objects, and enums, which can be found in the `Entities/`, `ValueObjects/`, and `Enums/` subdirectories respectively.

  All entities inherit from the `Entity` base class, which is further extended by the `VersionedEntity` class, which provides version tracking. Aggregate roots additionally implement the `IAggregateRoot` interface located in the root of the `Models/` directory.

- `Services/` directory contains implemented domain services, specifically the `CurrencyDomainService`, which implements the operation of changing the application's default currency.

4.1.7 PortEval.Infrastructure

`PortEval.Infrastructure` implements the infrastructure layer of the application, which includes communication with the database and external data sources.

Communication with the application's database is implemented in two classes: `PortEvalDbContext`, which implements and configures an EF database context, and `PortEvalDbConnectionCreator`, which acts as a factory for SQL connections to the application's database..

- `Configurations/` folder contains Code First configurations implemented using EF Core, which define the database schema using domain entities from `PortEval.Domain`.

- `FinancialDataFetcher/` directory contains implementations of individual data sources, which can then be integrated with the data fetcher library.

  These data sources are implemented as simple API clients, utilizing a custom `HttpClient.GetJson()` extension method for data retrieval, which returns a response of type `Response` used by the data fetcher library. Additionally, the infrastructure layer implements the `IFinancialDataFetcher` interface defined in `PortEval.Application.Core`, which acts as a facade over the data fetcher library.

- `Queries/` directory contains implementations of read queries according to interfaces defined by the `PortEval.Application.Core` project. They are implemented using Dapper extension methods, and they typically return DTOs to the caller.

- `Migrations/` folder contains auto-generated database migrations based on configurations from the `Configurations/` directory. These migrations were generated using the `Add-Migration`[2] command provided by EF Core, and are executed on application startup.

[2]Further information can be found here: `https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli`

- `Repositories/` directory contains implementations of EF repositories according to interfaces defined by the `PortEval.Application.Core` project. These repositories return domain entities to the caller.

The infrastructure layer is also responsible for dispatching domain events so they can be processed by appropriate domain event handlers from the application logic layer. Such events are dispatched in the `Commit()` and `CommitAsync()` methods of `PortEvalDbContext` class, and are dispatched after the underlying transaction is committed.

4.1.8 PortEval.Tests.Unit

`PortEval.Tests.Unit` implements the unit tests for the application.

- `BackgroundJobTests/` directory contains unit tests validating behavior of individual background jobs.
- `ControllerTests/` directory contains unit tests validating behavior of controllers.
- `CoreTests/` directory contains unit tests of services, domain event handlers, calculators, chart data generators, and bulk import and export functionality.
- `DataFetcherTests/` directory contains unit tests of the data fetcher library. This library was tested with a fake API implementation contained in the `HelperTypes/` subdirectory.
- `DomainTests/` directory contains unit tests of the domain layer.
- `ModelTests/` directory contains unit tests of application model validators.
- `Helper/` directory contains utility functionality, such as extension methods for creating mocks, or test data for testing CSV processing.

4.1.9 PortEval.Tests.Integration

`PortEval.Tests.Integration` implements integration tests for the application. These tests focus on testing behavior which needs to communicate with the database or external services. This includes queries, repositories, and financial data sources.

- `QueryTests/` directory contains integration tests for application queries.
- `RepositoryTests/` directory contains integration tests for repositories.
- `FinancialDataSourceTests/` directory contains integration tests for individual data sources implemented by the infrastructure layer.

4.2 Frontend

The frontend is implemented as a Node.js TypeScript React application. Its implementation can be found in the `app/web/` directory.

4.2.1 Project structure

The root directory of the project contains the configuration files of the project. These files include:

- `package.json`: A manifest file describing the package and its dependencies.
- `package-lock.json`: A lockfile holding information about the dependency versions used.
- `.eslintrc.json`: A configuration file defining *ESLint*[3] rules.
- `jsdoc.json`: A configuration file for *JSDoc*[4] documentation generator.
- `tsconfig.json`: A configuration file for TypeScript compilation.

The package itself is structured as follows:

- `nginx/` directory contains *nginx*[5] configuration, which is used in production Docker environment.
- `public/` directory contains static files, such as `index.html`, the favicon, and `robots.txt`.
- `src/` directory contains the source code of the application, and is further structured as follows:
  - `__tests__/` directory contains automated tests and related data.
  - `components/` directory contains React components.
  - `context/` directory contains definitions of React Context objects.
  - `hooks/` directory contains custom React hooks.
  - `redux/` directory contains the definition of the Redux store and the implementations of RTK Query endpoints.
  - `utils/` directory contains various non-React utilities.
  - `constants.ts` file implements various application constants.
  - `index.tsx` file implements the entry point of the application.
  - `setupProxy.js` file implements a reverse proxy to the application's backend for the development Node.js environment.
  - `setupTests.ts` file configures Jest tests.
  - `types.ts` file defines TypeScript types which are shared across multiple concerns.

[3] `https://eslint.org/`
[4] `https://jsdoc.app/`
[5] `https://www.nginx.com/`

4.2.2 Shared components

The frontend implements a set of cross-cutting components which do not belong to any particular view. These components are:

- `App`: Root component of the application, which also sets up the SignalR connection to the backend's notification system.
- `Layout`: Implements the general layout of the application and handles routing.
- `Header`: Renders the header of the application.
- `Sidebar`: Renders the sidebar of the application.
- `OffcanvasSidebar`: Wraps the sidebar in a responsive drawer, which is hidden by default on mobile screens.
- `ModalWrapper`: Renders a modal window with the specified children using the *react-modal*[6] package.
- `LoadingWrapper`: Provides a wrapper for components which depend on asynchronously fetched data, displaying a loading animation until the fetch finishes.
- `LoadingSpinner`: Renders a loading spinner animation.
- `LoadingBubbles`: Renders a loading bubbles animation.
- `PageHeading`: Renders a page heading.
- `PageSelector`: Renders pagination controls.

4.2.3 Views

Different pages of the application are implemented as *views*, and can be found in the `components/views/` directory. The following views have been implemented:

- `Dashboard`: Renders the dashboard of the application.
- `PortfolioListView`: Renders an expandable list of all user's portfolios .
- `PortfolioView`: Renders an overview of a single portfolio.
- `InstrumentListView`: Renders a paginated list of all created instruments .
- `InstrumentView`: Renders an overview of a single instrument.
- `ChartListView`: Renders a list of charts created by the user.
- `ChartView`: Renders an interface to configure a line chart .

[6] https://github.com/reactjs/react-modal

- `CurrenciesView`: Renders a form to change the application's default currency and a list of current exchange rates .
- `ImportExportView`: Renders an interface to export or import CSV data .
- `SettingsView`: Renders a form to change user settings.

These views are typically comprised of multiple components, the most important of which will be described in the following sections.

4.2.4 Tables

The application implements the following tables:

- `ChartsTable`: Displays all created charts.
- `ExchangeRatesTable`: Displays current exchange rate from the specified currency code.
- `ImportsTable`: Displays all initiated CSV imports.
- `InstrumentPricesTable`: Displays paginated price history of a specific instrument.
- `InstrumentSplitsTable`: Displays split history of a specific instrument.
- `InstrumentsTable`: Displays paginated instruments.
- `PortfoliosTable`: Displays all created portfolios.

  This table is expandable, displaying `PositionsTable` on expansion.
- `PositionsTable`: Displays positions of a specific portfolio.

  This table is expandable, displaying `TransactionsTable` on expansion.
- `TransactionsTable`: Displays transactions of a specific position.

These tables are backed by a generic `DataTable` component, which renders the table based on the provided data and column definitions. `DataTable` additionally uses the `DataTableExpandableComponent` wrapper for expandable content. Furthermore, it allows different column definitions based on screen size, which is represented by Bootstrap's breakpoints. This functionality is utilized for wide tables, such as `PortfoliosTable`, where a more compact version of the table is displayed on mobile screens.

4.2.5 Forms

The frontend utilizes multiple forms for data editing. They can be found in the `src/components/forms/` directory. These forms are:

- `CreateInstrumentForm`: Renders an instrument creation form.
- `CreateInstrumentPriceForm`: Renders an instrument price creation form.

- `CreateInstrumentSplitForm`: Renders an instrument split creation form.
- `OpenPositionForm`: Renders a form to open a new position in a portfolio. This form additionally enables the user to create a corresponding instrument.
- `CreatePortfolioForm`: Renders a portfolio creation form.
- `CreateTransactionForm`: Renders a transaction creation form.
- `EditChartMetaForm`: Renders a form for editing a chart's name.
- `EditInstrumentForm`: Renders an instrument edit form.
- `EditPortfolioForm`: Renders a portfolio edit form.
- `EditPositionForm`: Renders a position edit form.
- `EditTransactionForm`: Renders a transaction edit form.
- `ExportDataForm`: Renders a form for exporting data in CSV format.
- `ImportDataForm`: Renders a form for importing data in CSV format.
- `ChangeDefaultCurrencyForm`: Renders a form to change the default currency of the application.
- `SettingsForm`: Renders a form to change the user settings.
- `ChartLineConfiguratorForm`: Renders a form to change the style of a chart line.

Forms are composed of custom form fields located in the `fields/` subdirectory. They are also responsible for calling the appropriate mutation hooks generated by RTK on form submit. However, they additionally enable custom callbacks on success, which are typically used to display a toast notification or to close the parent modal.

4.2.6 Charts

The main facade for chart rendering is the `PortEvalChart` component, which accepts the chart definition as a prop and retrieves the appropriate chart data from the backend using the `useGetChartDataQuery` hook generated by RTK.

The actual rendering of the chart is then delegated to the `LineChart` component, which is a React wrapper for custom d3.js code implemented in the `src/utils/lineChart.ts` file as an `SVGLineChart` class.

The application additionally implements several components for chart configurability. These components are:

- `ChartConfigurator`: Enables configuration of chart settings, such as type or date range.
- `ChartPreview`: Renders a chart preview with a link to the full-sized chart.

- `InstrumentPicker`: Renders a list of instruments which can be added to the chart.

  Individual instruments are rendered using the `InstrumentPickerItem` component.

- `PortfolioPicker`: Renders a list of portfolios which can be added to the chart.

  Individual portfolios are rendered using the `PortfolioPickerItem` component.

- `PositionPicker`: Renders a list of portfolio positions which can be added to the chart.

  Individual positions are rendered using the `PositionPickerItem` component.

- `LinePreview`: Renders a small preview of the chart line based on its configuration.

Some of these components depend on the `ChartLineConfigurationContext` React Context, which is used to read and update the configuration of the parent chart.

4.2.7 Hooks

Several custom hooks were also implemented. These hooks encapsulate functionality which requires usage of different, often built-in React hooks. They can be found in the `src/hooks/` directory.

- `useGetQueryParam`: Retrieves the value of the specified query parameter.
- `useGetRouteState`: Retrieves the value of the provided route state.
- `useInstrumentPriceAutoFetchingState`: A custom state wrapper which enables automatic fetching of instrument price at the specified time, while auto-refreshing whenever this time changes.
- `useLocalStorage`: Provides a state-like interface enabling retrieval and storage of data in the browser's local storage.
- `usePageTitle`: Sets the title of the page.
- `useUserSettings`: Provides a state-like interface enabling retrieval of user settings saved in the browser's local storage.

4.2.8 Redux

As discussed in *Implementation*, state management was implemented using Redux and RTK. Additionally, communication with the RESTful API provided by the backend was implemented using the RTK Query submodule.

These implementations are contained in the `src/redux/` directory. It contains a `store.ts` file, which exports a factory function for initializing the Redux store. The `api/` subdirectory then implements the individual endpoints of the RESTful API. They are implemented across multiple files, typically separated by a specific type of data.

5. Testing

The following chapter describes the methodology used for testing the final application.

5.1 Automated testing

To test correctness of the application's behavior, numerous automated tests were developed for both backend and frontend. This section discusses these tests and describes how to run them.

5.1.1 Backend

As the backend contains the majority of the application's source code, extensive emphasis was placed on testing its functionality. This was achieved by a set of unit and integration tests, where unit tests validate the behavior of individual classes and methods, while integration tests validate behavior spanning multiple layers, typically including communication with the database or external data sources.

These tests can be executed by running the `dotnet test` command in the solution's root directory.

Unit tests

Backend's unit tests were implemented using the *xUnit*[1] testing framework. As much of the application's behavior utilizes some injected dependencies, it was necessary to isolate tested functionality from these dependencies by providing mocks in their stead. To simplify creation of such mocks, *Moq*[2] mocking library was used.

Furthermore, the *AutoFixture*[3] library was used to facilitate creation of test data, and to act as an IoC container allowing injection of mock dependencies. Together with Moq, this allows unit test implementations to focus on behavior, instead of configuration concerns.

In total, 652 unit tests were implemented, covering the following concerns:

- Background jobs
- Controllers
- Domain services
- Domain event handlers
- Services
- Calculators

[1] `https://xunit.net/`
[2] `https://github.com/moq/moq4`
[3] `https://github.com/AutoFixture/AutoFixture`

- CSV processing
- Data fetcher library
- Application model validators

Integration tests

Similar to unit tests, integration tests were also built using the xUnit framework. These tests focus primarily on behavior which requires communication with an external system, such as:

- Queries

  Query tests are executed against a fully running backend using a test SQL Server database with fake seeded data. This test database is initialized in a Docker container using the *Testcontainers*[4] library. This means that the Docker daemon must be running for these tests to be executed successfully.

- Repositories

  These tests are executed against an in-memory database, which was deemed sufficient for the level of complexity of implemented repositories, as they do not utilize engine-specific functionality.

- Financial data sources

  These tests validate the behavior of implemented financial data sources and their integration with the data fetcher library. They are executed with a mock of an HTTP client returning stub responses on calls to expected data source endpoints.

In total, 140 such tests were implemented.

5.1.2 Frontend

The frontend implemented a separate set of integration tests using *Jest*[5]. Their goal is to test specific use cases and user interactions within the application, mirroring the way the user is expected to use it. This means that these tests are generally executed against the top-level views of the application, and do not test individual components. This approach is facilitated by *React Testing Library*[6].

As frontend depends on the backend's API for a significant part of its functionality, this API needs to be mocked. For this, the *Mock Service Worker*[7] library was utilized, which intercepts network requests and allows the user to handle the request completely in-memory, while providing a stub response to the client.

These tests can be executed by running the `npm run test` command in the package's root directory.

[4] `https://dotnet.testcontainers.org/`
[5] `https://jestjs.io/`
[6] `https://testing-library.com/docs/react-testing-library/intro`
[7] `https://mswjs.io/`

5.2 Performance evaluation

According to non-functional requirement **NFR8**, the application must be able to render any chart containing 10 lines in under two seconds. Fulfilling this requirement is not easy, as chart rendering needs to process and transfer large amounts of data, which requires several layers of the application to function optimally:

- Database and queries
- Financial metric calculators
- Chart data generation algorithm
- Currency conversion
- Communication between frontend and backend
- SVG chart rendering mechanism

To evaluate the fulfillment of this requirement, multiple end-to-end benchmarks were performed on several test charts. These benchmarks are described in the following subsections. It should be noted that the goal of these benchmarks is not to provide an extensive analysis of the chart rendering mechanism, but rather to present us with a general estimate of its performance, which can enable discussion about the fulfillment of the aforementioned requirement.

5.2.1 Test data

Before executing the benchmarks, the application had to be populated with enough data to enable creation of reasonably complex multi-line charts. For this reason, 15 instruments included in Dow Jones Industrial Average (DJIA) were imported into the application, which consequently downloaded complete price histories for each of them.

After this, 15 portfolios were created, each with 1 position and 1 transaction executed in November 2022. Each such position constituted a different instrument, so each imported instrument was represented at least once in created portfolios.

Based on this data, the following types of charts were created:

- USD price chart containing instrument lines
- USD price chart containing portfolio lines
- EUR price chart containing instrument lines
- EUR price chart containing portfolio lines
- Performance chart containing instrument lines
- Performance chart containing portfolio lines

For each of these types, 4 charts were created, containing 1, 5, 10, and 15 different lines respectively.

Chart type	Line entity	Currency	Line count	Time
Price	Instrument	USD	1	190.0 ms
Price	Instrument	USD	5	558.7 ms
Price	Instrument	USD	10	**953.8 ms**
Price	Instrument	USD	15	1430.5 ms
Price	Instrument	EUR	1	347.2 ms
Price	Instrument	EUR	5	1068.1 ms
Price	Instrument	EUR	10	**1820.0 ms**
Price	Instrument	EUR	15	2764.7 ms
Price	Portfolio	USD	1	176.3 ms
Price	Portfolio	USD	5	497.7 ms
Price	Portfolio	USD	10	**929.0 ms**
Price	Portfolio	USD	15	1355.1 ms
Price	Portfolio	EUR	1	338.5 ms
Price	Portfolio	EUR	5	1002.9 ms
Price	Portfolio	EUR	10	**1750.3 ms**
Price	Portfolio	EUR	15	2742.8 ms
Performance	Instrument		1	163.1 ms
Performance	Instrument		5	451.9 ms
Performance	Instrument		10	**757.7 ms**
Performance	Instrument		15	1171.6 ms
Performance	Portfolio		1	151.6 ms
Performance	Portfolio		5	401.2 ms
Performance	Portfolio		10	**717.4 ms**
Performance	Portfolio		15	1082.3 ms

Table 5.1: Chart rendering mechanism benchmark results

5.2.2 Environment

The benchmarks were executed on a Windows machine with a Ryzen 5 3600 CPU and 16GB of RAM, which was running both the backend and the frontend of the application in Internet Information Services (IIS).

To measure the performance in a way that would most align with the users' experience, charts were placed on the dashboard page and scaled to fill the width and the height of the browser window. This was done independently for each chart to prevent interference caused by rendering multiple charts.

The benchmarks were then executed in Google Chrome version 111 using Chrome DevTools. Specifically, we measured the time difference between the first API request for data and the moment the chart was visible and interactive, which was defined as the moment when all rendering, layout shifting, and callbacks were finished.

5.2.3 Results

The benchmarks were performed 10 times for each chart. The averaged results can be seen in Table 5.1, with 10 line measurements highlighted in bold.

Several observations can be made from these results. First of all, charts requiring currency conversions from USD to EUR performed the worst in these benchmarks, in some cases being twice as slow as their USD counterparts. While they fulfilled the requirement, this indicates a good starting point for further optimizations.

On the other hand, portfolio charts have shown very good results, slightly outperforming instrument charts with the same line count. As portfolio chart lines generally require more data than instrument lines, the most likely reasons are sub-optimal database access for instrument data, or an inefficient algorithm for instrument chart generation.

Overall, requirement **NFR8** can be described as fulfilled. Nevertheless, some improvements can be considered:

- **Database indexing**

  Inefficient or fragmented database indexes are a possible reason for sub-optimal currency conversion performance, as the current version of the application does not rebuild or reorganize these indexes, and the currency exchange rates table will generally contain up to a million records.

- **Line aggregation**

  Currently, each line's data is retrieved by executing a separate API request. This was done to allow the frontend to cache individual lines instead of whole charts. Alternatively, the backend can be modified to provide the complete chart data in a single request, allowing reuse of database connections or retrieved data.

- **Backend caching**

  While the frontend can cache individual lines, implementing caching in the backend could prove to be a significant performance improvement, as intermediary data (such as prices, transactions, or currency exchange rates) could be cached as well.

Conclusion

The goal of this work, as stated in *Introduction*, was to design and implement an application enabling tracking and evaluation of investment portfolios, with support for user-defined charts and data retrieval from various data sources, which must be implemented in a modular fashion.

By implementing PortEval, this goal has been fully achieved. The project is functional in accordance with requirements from the *Requirements* section, and can be used to track portfolios consisting of different instruments, with financial market data being regularly retrieved from different data sources. Additionally, the application allows creation and saving of custom charts, which can then be laid out in a user-defined grid on a single page.

Future work

In this section, we will describe possible future extensions of the application, each with a short description of how it could be achieved in the current design.

Alerts

As the application already supports real-time notifications displayed in the front end, the system could be further improved to display alerts based on certain triggers, such as rapid portfolio value decrease. Detection of such triggers could be implemented as side-effects of instrument price creation, or as separate asynchronous jobs.

For these notifications to be useful, they should be displayed to the user even if they do not have the application open at the time. To achieve that, a notification queue would need to be implemented as well, either as a part of the existing database, or as a separately deployed service.

Multi-user support

In its current version, the application does not support multiple users, each tracking their own set of portfolios. To achieve this, an authorization mechanism would need to be implemented in the application. Additionally, user-specific entities (such as portfolios or charts) would need to be extended with a relationship to a specific user. On the other hand, instruments tracked by the application should remain global.

Dividend analysis

The application could additionally support tracking and analysis of dividends. They could be imported using the existing data fetcher library, possibly by extending existing data sources. These dividends could then be incorporated into profit and performance evaluation as additional cash flows. Furthermore, these dividends could be displayed in separate charts, or they could be added to the existing charts' tooltips.

Broker integration

The application could be further improved by importing transactions automatically from various brokers, such as InteractiveBrokers[8] or eToro[9]. To achieve this, an integration with each broker's API would need to be implemented, with additional jobs handling regular data retrieval. Communication with the APIs could be implemented as part of the existing data fetcher library, or using a different solution altogether.

[8] `https://www.interactivebrokers.com/en/home.php`
[9] `https://www.etoro.com/`

Bibliography

Kendall E. Atkinson. *An Introduction to Numerical Analysis*. Second Edition. Wiley, 1989. ISBN 0-471-62489-6.

G. Erich, H. Richard, V. John, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st Edition. 1994. ISBN 978-0-201-63361-0.

E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1st Edition. Addison-Wesley Professional, 2003. ISBN 0-321-12521-5.

Bruce J. Feibel. *Investment Performance Measurement*. Wiley, 2003. ISBN 978-0-471-44563-0.

M. Fowler. *Patterns of Enterprise Application Architecture*. 1st Edition. 2005. ISBN 0-321-12742-0.

S. Kellison. *The Theory of Interest*. 3rd Edition. McGraw-Hill, 2008. ISBN 978-0-073-38244-9.

Palermo, J. The Onion Architecture: part 1, 2008. URL `https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/`. [Accessed: 4 March 2023].

Stack Overflow. Stack Overflow Developer Survey 2022, 2022. URL `https://survey.stackoverflow.co/2022/`. [Accessed: 25 February 2023].

TechEmpower. Web Framework Benchmarks, 2022. URL `https://www.techempower.com/benchmarks/#section=data-r21&test=composite`. [Accessed: 25 February 2023].

List of Figures

List of Abbreviations

CSV comma-separated values

UML Unified Modeling Language

SaaS Software as a service

SVG Scalable Vector Graphics

TTI Time to Interactive

ACID atomicity, consistency, isolation, durability

DBMS database management system

MWR Money-Weighted Return

IRR internal rate of return

TWR Time-Weighted Return

API Application Programming Interface

REST Representational state transfer

JSON JavaScript Object Notation

TSV tab-separated values

XML Extensive Markup Language

SVG Scalable Vector Graphics

EOD end of day

ETF exchange-traded fund

ORM object-relational mapping

EF Entity Framework

DTO Data Transfer Object

UI user interface

GUI graphical user interface

SPA single-page application

SOAP Simple Objects Access Protocol

DDD domain-driven design

SDK software development kit

CRA Create React App

CSS Cascading Style Sheets

DOM Document Object Model

RTK Redux Toolkit

IoC Inversion of Control

IIS Internet Information Services

DJIA Dow Jones Industrial Average

TLS Transport Layer Security

A. Attachments

A.1 Electronic attachment

This work comes with an electronic attachment `PortEval.zip`, which has the following structure:

- `app/` directory contains the source code of the application.
  - `server/` subdirectory contains the Visual Studio solution of the application backend.
  - `web/` subdirectory contains the Node package of the application frontend.
  - `docker-compose.yml` file is a Docker Compose configuration file for the production environment.
  - `docker-compose.development.yml` file is a Docker Compose configuration file for the development environment.
  - `.env` file is a configuration file for production Docker Compose environment variables.
  - `.env.dev` file is a configuration file for development Docker Compose environment variables.
- `docs/` directory contains documentation generated from the application's source code.
  - `backend/` subdirectory contains documentation generated from the backend's XML comments using Doxygen.
  - `frontend/` subdirectory contains documentation generated from the frontend's comments using JSDoc.
- `setup/` directory contains the experimental Windows setup bundle for the whole application.
- `test_data/` directory contains examples of CSV import files.

A.2 Administrator documentation

This documentation describes how to configure and run *PortEval*. The application consists of three layers, each of which needs to be running for the application to function as expected. These layers are a React frontend, an ASP.NET Core backend, and an SQL Server database.

The application is located in the `app/` directory of the electronic attachment. The commands described below need to be executed from that directory.

A.2.1 Data sources

Several data sources require a valid API key to function correctly. These keys can be retrieved at the following URL addresses:

- Mboum: `https://rapidapi.com/sparior/api/mboum-finance/`
- Tiingo: `https://api.tiingo.com/account/api/token`
- Alpha Vantage: `https://www.alphavantage.co/support/#api-key`
- Open Exchange Rates: `https://openexchangerates.org/signup`

However, the application supports Yahoo Finance and ExchangeRate.host out of the box, so while usage of the data sources above is recommended for reliability reasons, it is ultimately optional.

A.2.2 Docker Compose

The easiest and recommended way to run the whole application is using the Docker Compose configuration located in the application's root directory. The application can be built and started using the following commands:

```
$> docker compose build
$> docker compose up
```

The application will then be available at `http://localhost:3080/`.

If the administrator wants to utilize additional data sources in addition to default ones, they need to provide valid API keys for these sources. These keys should be provided to the application as environment variables[1]. The easiest way to do so is using the `.env` file, which should be located in the same directory as the `docker-compose.yml` configuration file.

```
PORTEVAL_Tiingo_Key=[key]
PORTEVAL_RapidAPI_Mboum_Key=[key]
PORTEVAL_AlphaVantage_Key=[key]
PORTEVAL_OpenExchangeRates_Key=[key]
```

Listing A.1: Example `.env` file

[1]More information on supplying environment variables to a Docker Compose environment can be found here: `https://docs.docker.com/compose/environment-variables/set-environment-variables/`

A.2.3 Custom deployment

It is possible to deploy each part of the application manually. This subsection provides information about deployment of each of these parts.

SQL Server

PortEval was developed and tested using SQL Server 2019[2]. The application only needs one SQL Server instance, and it is the responsibility of the administrator to configure it on the platform of their choice.

Backend

To compile and run the backend of the application, the target machine must have the following installed:

- .NET 7.0 SDK
- .NET 7.0 runtime
- ASP.NET Core 7.0 runtime

The Visual Studio solution can be found in the `app/server/` directory. It can then be compiled using the `dotnet publish` command, for example:

```
$> dotnet publish -c Release -f net7.0 -o /var/www/api
   /p:EnvironmentName=Production PortEval.Application/
```

The example above compiles the backend in a production environment, storing the compiled application into the `/var/www/api` directory.

Before running the application, the administrator will need to configure the port, the database connection, the file storage directory, and optionally the data source API keys. These settings can be configured using the `appsettings.json` file in the root directory of the deployed application, for example as follows:

```
{
  "ConnectionStrings": {
    "PortEvalDb": "connection string"
  },
  "PORTEVAL_Tiingo_Key": "key",
  "PORTEVAL_RapidAPI_Mboum_Key": "key",
  "PORTEVAL_AlphaVantage_Key": "key",
  "PORTEVAL_OpenExchangeRates_Key": "key",
  "PORTEVAL_File_Storage": "path to storage directory",
  "Urls": "http://0.0.0.0:8080"
}
```

Listing A.2: Example `appsettings.json` file

[2]The free Express version can be downloaded at the following address for Windows machines: `https://www.microsoft.com/en-us/download/details.aspx?id=101064`. For Linux distributions, refer to the following guide: `https://learn.microsoft.com/en-us/sql/linux/sql-server-linux-setup`

Alternatively, these values can be provided as environment variables. The database connection string should be provided in a `CUSTOMCONNSTR_PortEvalDb` variable, while all other configuration values should be provided using names corresponding to keys from the configuration file above.

The backend can then be started by running the following command in the directory of the compiled application:

```
$> dotnet PortEval.Application.dll
```

Frontend

The frontend is located in the `app/web/` directory of the electronic attachment. To build it, Node.js version 17+ must be installed on the target machine. The application can then be built using the following commands:

```
$> npm install
$> npm run build
```

The compiled frontend will then be located in the `build/` subdirectory.

The frontend expects the backend's API to be available at `/api` relative URL. It is the responsibility of the administrator to configure correct reverse proxy settings on the web server of their choosing to allow the frontend to access the API. Additionally, to support WebSocket notifications, the web server (and the reverse proxy) should support HTTP connection upgrade[3]. An example nginx configuration file supporting all of the above can be found at `app/web/nginx/nginx.conf`.

A.2.4 Setup bundle

For Windows 64bit machines, the administrator can use the setup bundle which can be found in the `setup/` directory of the electronic attachment. This bundle installs the full application to the machine's IIS server at port 5432, including dependencies such as SQL Server and URL Rewrite. It should be noted that this bundle is experimental, and it cannot automatically resolve possible issues with already installed components. It should also be noted that installation of this bundle will typically require a machine restart.

The bundle will first install the required dependencies, after which it will open a PortEval installer. Installation instructions are as follows:

1. Click *Next*.
2. A dialog window will appear allowing the user to optionally configure API access keys for instrument and currency data. To configure access for a specific source, select the source from the list box and enter the required credentials in text fields below the list box.
3. Click *Next*
4. Wait for the installation to finish.

The application will then be available at `http://localhost:5432/`.

[3] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade`

A.3 User documentation

This documentation describes how to use the application, outlining the basic features and the general workflow.

A.3.1 Workflow

The workflow of the application revolves around three key financial entities:

- **Instrument**: A priced entity the price of which evolves over time, such as stocks, ETFs, and cryptocurrencies.
- **Position**: An instrument traded in a portfolio.
- **Portfolio**: A collection of positions.

As is evident from these definitions, everything in the application ultimately revolves around instruments, which can then be composed into positions and portfolios. This is also reflected in the structure of this documentation, which first describes how to manage these instruments, and only then getting into advanced topics such as portfolio or chart management.

A.3.2 Getting started

After the user opens the application, they will be greeted by the dashboard. On first start, this dashboard will be empty. However, eventually it will contain charts created by the user, laid out in a custom flexible grid.

To the left of each page the user will see a navigation sidebar. Each link in this sidebar will lead to a different page, which will be further described in the following subsections.

In the top-right corner there is a refresh button, forcing the application to update displayed data. However, most of the times it is going to be unnecessary, as the application will typically refresh its data automatically, maintaining up-to-date financial information.

A.3.3 Instrument management

The user may navigate to the instruments page (Figure A.1) by pressing the `Instruments` link in the navigation bar. This page contains a paginated list of all instruments created in PortEval, together with their current price.

An instrument can be added to the application by pressing the `Create new instrument` button in the top-right corner of the page. Pressing this button opens a modal window containing the instrument creation form (Figure A.2).

This form contains the following fields:

- Name: Full name of the instrument, such as *Alphabet Inc.*
- Symbol: Instrument ticker, such as *GOOGL*.
- Exchange: The stock exchange at which the instrument is traded, such as *NASDAQ*. This field is optional.

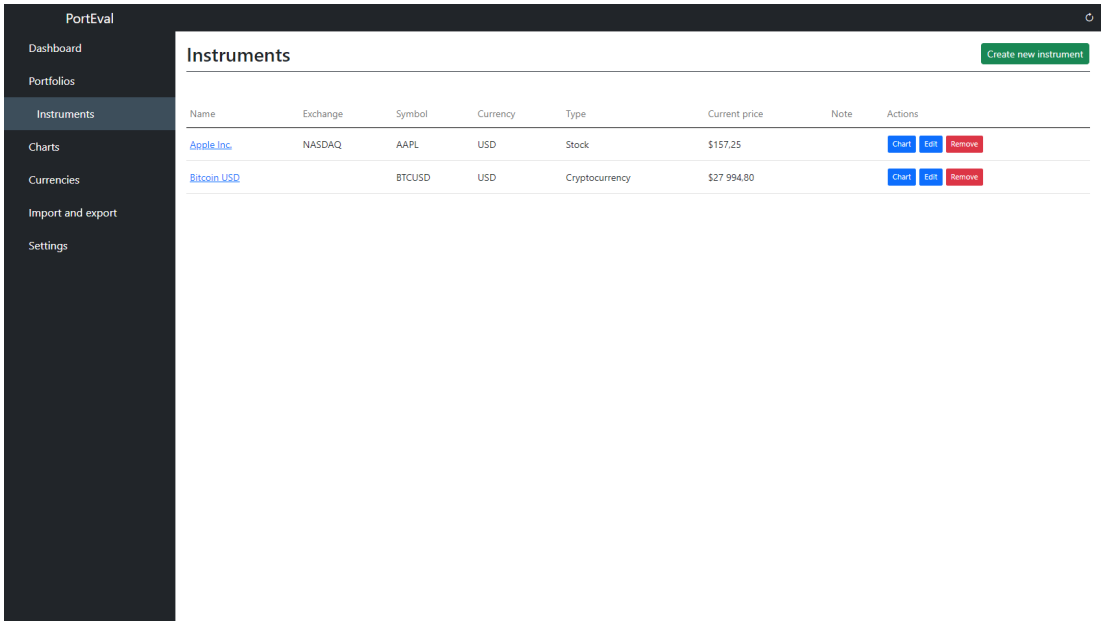

Figure A.1: Instruments page

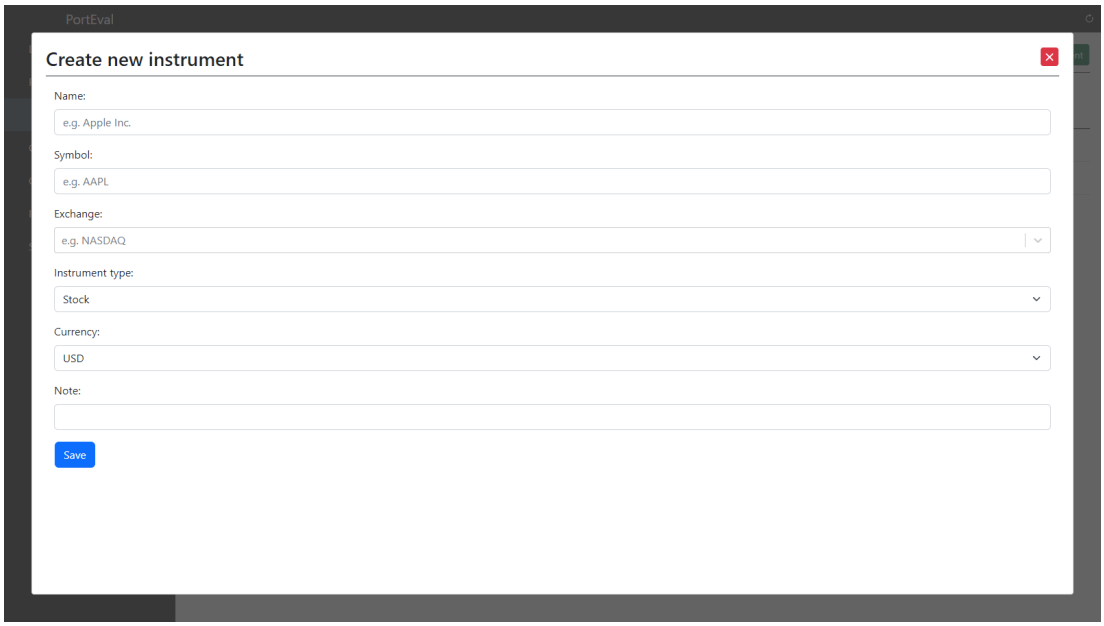

Figure A.2: Instrument creation form

- Type: The type of the instrument, such as *Stock*.
- Note: A custom note for the instrument. This field is optional.

After an instrument is created, PortEval will use the provided instrument information to try and retrieve the instrument's price history from integrated external sources. When this process finishes, a notification will be displayed in the web application, indicating whether any prices were downloaded.

Viewing an instrument

Clicking on instrument's name in the table will navigate the user to the instrument's page (Figure A.3).

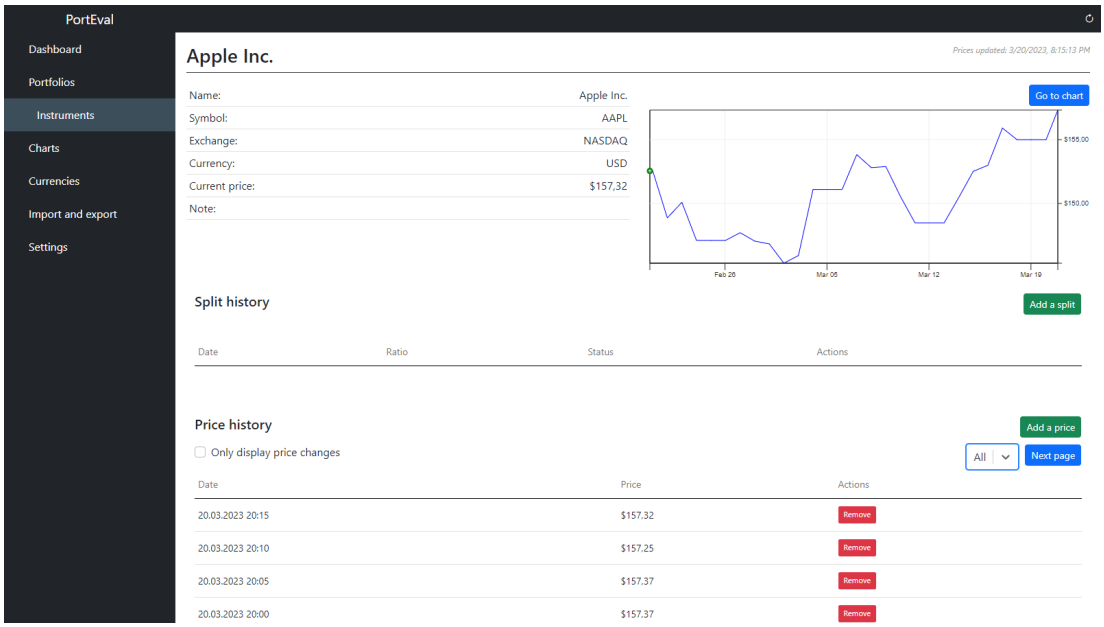

Figure A.3: Instrument page

This page contains key information about the instrument including several metrics and a price chart. Additionally, it contains histories of the instrument's splits and prices. Both these histories should be maintained automatically by the application, however, in some cases the user may want to make their own changes.

Existing prices can be removed by pressing the `Remove` button in the *Actions* column of the price row, while a new price can be created by pressing the `Add a price` buttons above the table.

Splits are more complex than prices, as creating a split will retrospectively adjust the prices of the instrument and the transactions of the related positions according to the specified split factor. Pressing the `Add a split` button will open the split creation form (Figure A.4).

In this form, the date field should indicate the date and time when the instrument starts being traded on a split-adjusted basis, which is typically the start of trading hours on the following day after the split. The numerator and denominator fields then specify the split factor, where *numerator*/*denominator* represents the multiplier of the total number of shares in circulation. For example, a three-for-one split's numerator would be 3, while its denominator would be 1.

After a split is created, the application will automatically adjust the prices and transactions affected by this split, dividing the instrument and transaction prices by the split factor, while multiplying the transaction amounts by the same factor.

An invalid split can be rolled back by pressing the `Rollback` button next to its entry in the split history table. This will perform reverse operations on the

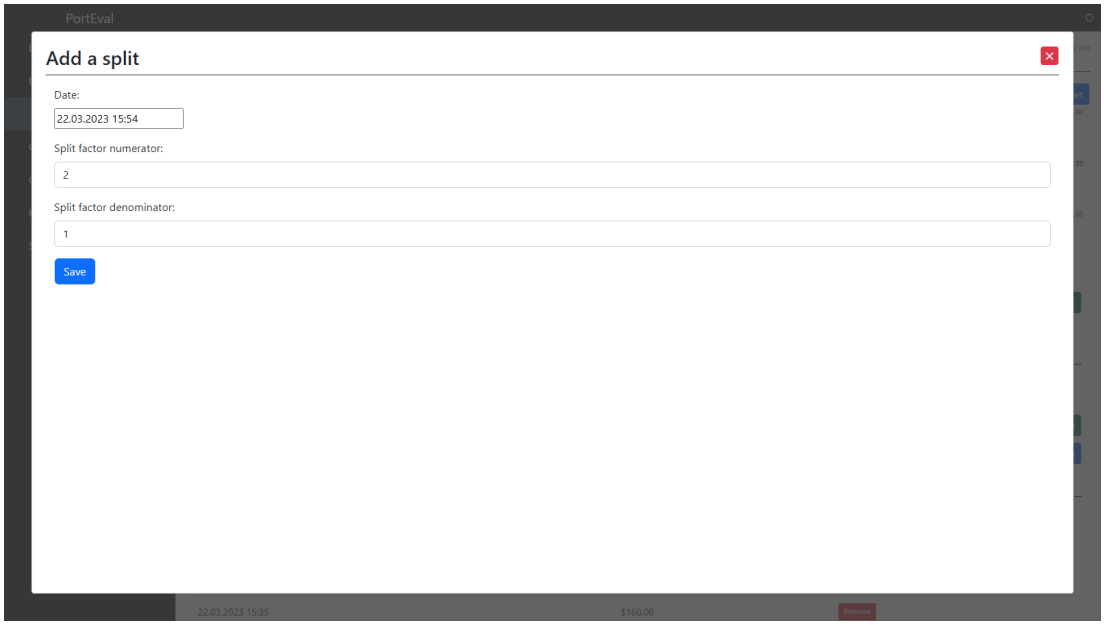

Figure A.4: Split creation form

same prices and transactions, reverting them to the values before the split was created.

A.3.4 Portfolio management

The `Portfolios` link in the navigation bar will lead to the portfolios management page (Figure A.5). This page contains an overview of all portfolios created by the user.

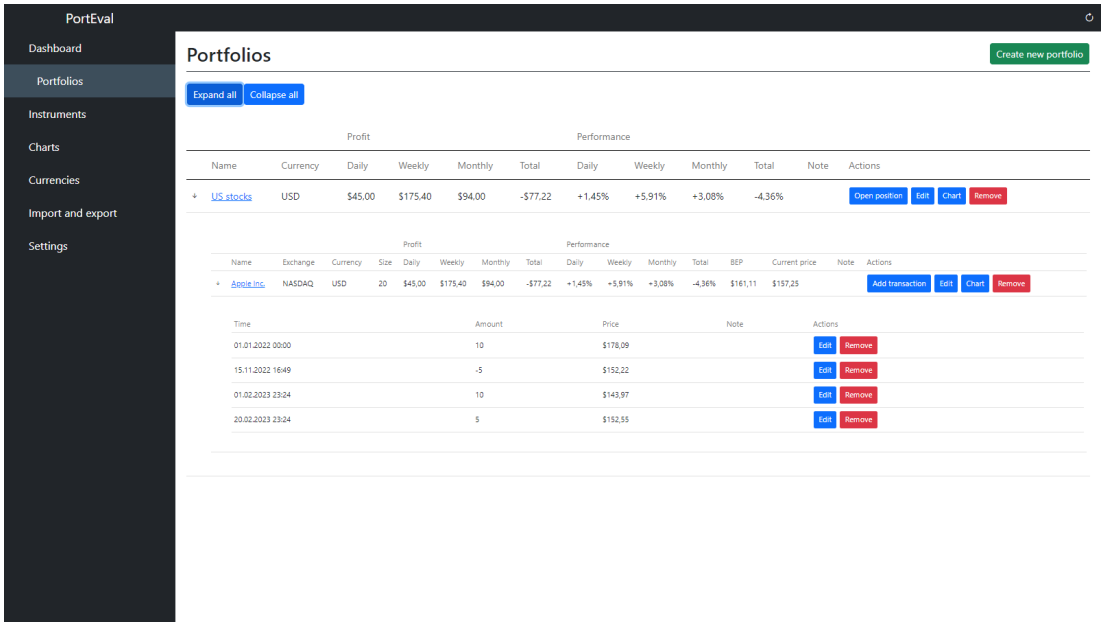

Figure A.5: Portfolios page

Each portfolio can be expanded and collapsed by pressing the arrow button next to the name of the portfolio, which will display the portfolio's positions

under the portfolio row. Similarly, individual positions can be expanded as well, displaying the expanded position's transactions.

A portfolio can be created by pressing the `Create new portfolio` button in the top-right corner of the page. Pressing this button opens a modal window containing the portfolio creation form, where the portfolio's name, currency, and optionally a note can be specified.

After a portfolio is created, the user can start opening positions in it by pressing the `Open position` button in the *Actions* column of the portfolio row.

A position requires an instrument and an initial purchase transaction to be provided. If the instrument does not yet exist, the user may check the `Create new instrument` checkbox below the instrument dropdown to create the instrument in the same form.

Further transactions can then be added to the position by pressing the `Add transaction` button in the *Actions* column of the transaction row. This opens a transaction creation form, where the transaction's date, amount, price per unit, and optionally a note can be specified. A positive amount indicates a purchase, while a negative amount indicates a sale.

Viewing a portfolio

Clicking on the portfolio's name in the table will navigate the user to the portfolio's page (Figure A.6).

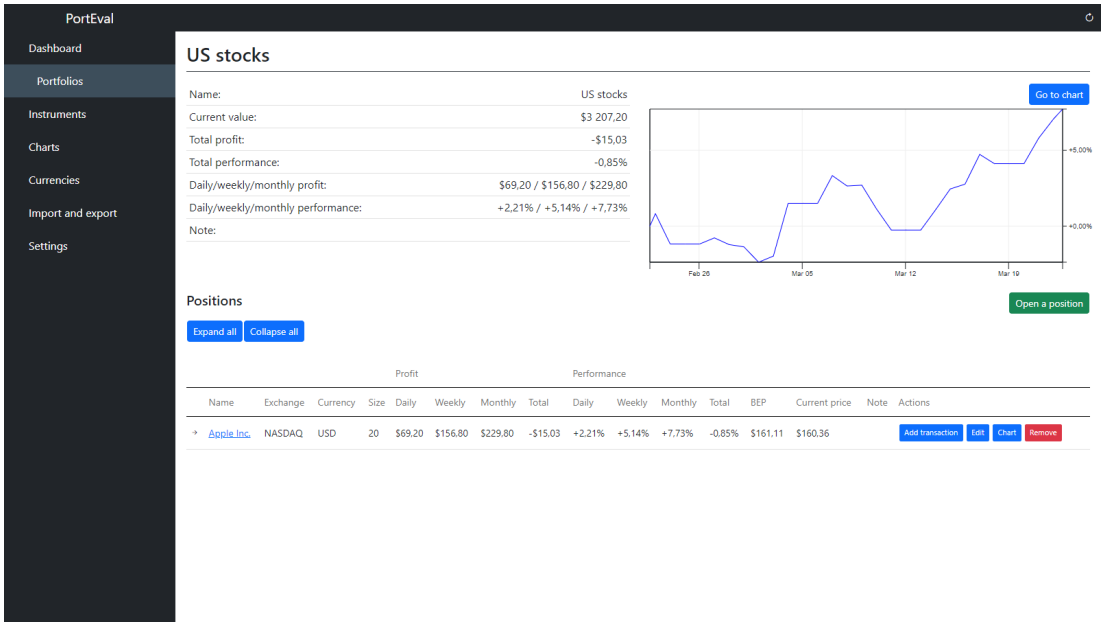

Figure A.6: Portfolio page

This page contains key information about the portfolio including its metrics and a performance chart. Additionally, it contains a table of portfolio's positions, which is equivalent to the expandable list of positions on the portfolios page.

A.3.5 Chart management

Pressing the `Charts` link in the navigation bar will open the charts page (Figure A.7). This page contains all the charts created by the user.

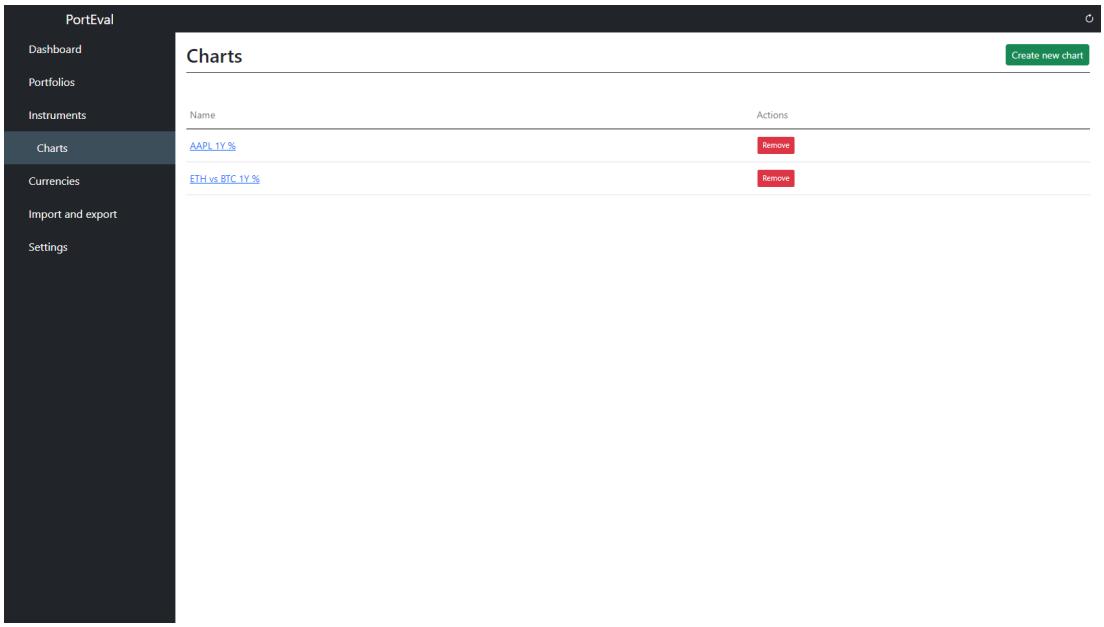

Figure A.7: Charts page

There are multiple ways to start creating custom charts in the application. Portfolio, position, and instrument tables outlined in previous subsections contain a `Chart` button in the *Actions* column, which opens a chart editing page with the selected financial entity already added. Alternatively, the charts page contains the `Create new chart` button in the top-right corner, which opens an empty chart.

Chart configuration

The chart editing page (Figure A.8) allows the user to view and configure their chart.

There are several configuration options available above the chart. The leftmost dropdown menu allows the user to select the type of the chart. The available types are:

- Price: At point X, a chart line will display the value of the financial entity at X.
- Profit: At point X, a chart line will display the profit of the financial entity between the start of the chart and X.
- Performance: At point X, a chart line will display the performance of the financial entity between the start of the chart and X.
- Aggregated profit: At point X, a chart line will display the profit of the financial entity between the previous chart point and X.

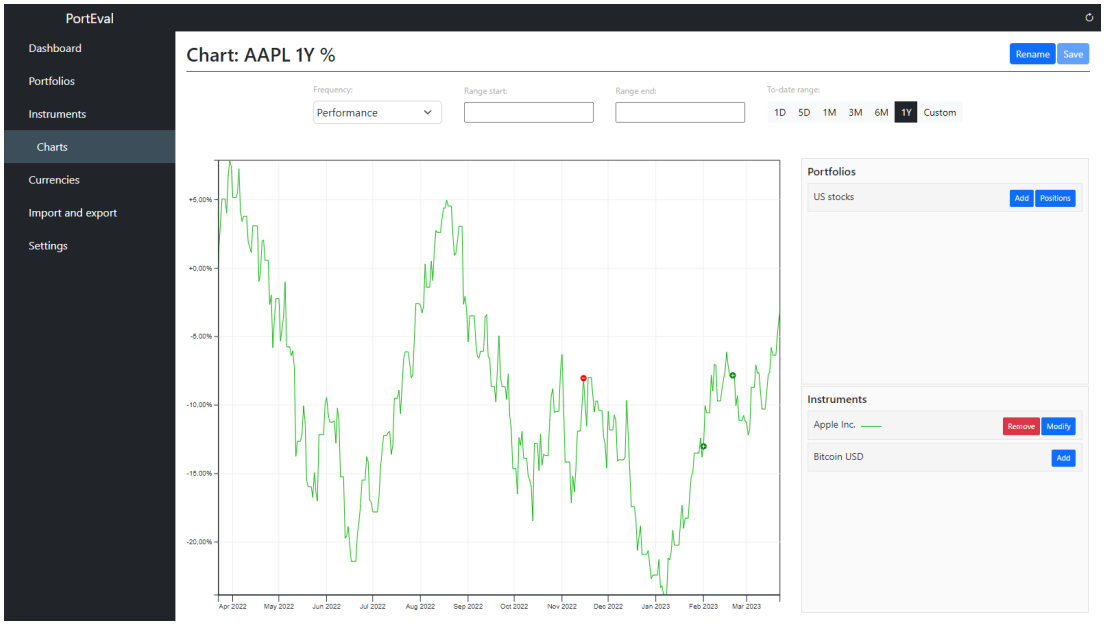

Figure A.8: Chart editing page

- Aggregated performance: At point X, a chart line will display the performance of the financial entity between the previous chart point and X.

Other configuration options include:

- Currency: Currency in which price values should be displayed. This option will only be displayed for price, profit, and aggregated profit charts.
- Date range start: Determines the date at which the chart should start, unless to-date range is selected.
- Date range end: Determines the date at which the chart should end, unless to-date range is selected.
- Frequency: Determines the interval between two chart points. This option will only be displayed for aggregated profit and aggregated performance charts.
- To-date range: Pre-configured date range. Charts with to-date range will always show data in the selected range, ending at current date and time.

The rightmost column contains two panels for adding lines to the chart. The top panel displays available portfolios, while the bottom panel displays available instruments. To add positions to the chart, the user can press the `Positions` button next to the portfolio containing the desired position. This will open a modal window allowing the user to add positions from the selected portfolio.

Each chart is automatically saved in five-second intervals.

A.3.6 Dashboard

The application dashboard is a configurable grid-like view of selected charts. To add a chart to the dashboard, the user can press the `Add charts` button in the top-right corner of the dashboard. After pressing the button, a modal window will appear listing the existing charts which have not yet been added to the dashboard. The user can then add one of these charts by starting to drag the chart name box, after which it can be dropped onto the desired position on the dashboard.

After the chart is dropped onto the dashboard, it switches into *Edit* mode (Figure A.9), where charts can be resized, moved, or removed from the dashboard. A chart can be resized by dragging an arrow in the bottom-right corner of the chart, or moved by holding the left mouse button over the chart and moving it.

When the user is done editing the dashboard, they can press the `Toggle dashboard edit` button in the top-right corner of the page. The same button can then be used to switch back into *Edit* mode.

Figure A.9: Dashboard in *Edit* mode

A.3.7 Data import and export

The `Import and export` navigation link will lead the user to the data import and export page (Figure A.10). On this page, the user can import or export application data in CSV format.

The user can initiate an export by selecting the desired data type in the `Export data type` dropdown and pressing the `Export` button. This will download a CSV file containing the requested data to the user's device.

Data can then be imported into the application using one of the predefined import templates. These templates can be downloaded by selecting the appropriate template in the `Import template` dropdown and pressing the `Download template` button.

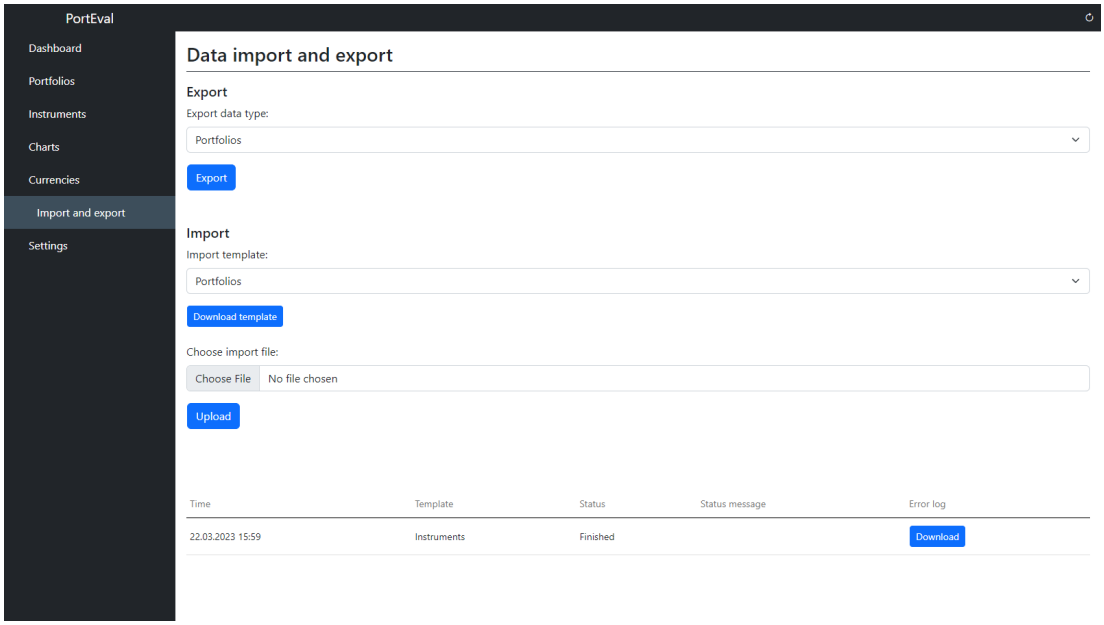

Figure A.10: Data import and export page

After filling in the template, the user can upload the data by selecting the file from their device using the `Choose import file` control and then pressing the `Upload` button. It is important to note that the type of the template being uploaded needs to be selected in the `Import template` dropdown before uploading the file.

After the import file is uploaded, it will be immediately scheduled for processing and its import status will be displayed in the imports table. When the processing is finished, the import status in the table will change to `Finished` or `Error`. In case of an error, a detailed error message will be displayed in the `Status message` column. Otherwise, an error log will be available for download in the `Error log` column.

Import templates

Import templates allows the user to insert or modify a specific type of data. To determine whether an entry should be inserted or modified, matching is performed based on key columns, outlined in bold below. The only exception is the *Instrument prices* template, which only allows insertion, not modification.

- Portfolios
  - **Portfolio ID**: An ID of an existing portfolio, must be left empty to create a new portfolio.
  - Name: Name of the portfolio.
  - Currency: Portfolio currency as a three-letter currency code, such as *USD*.
  - Note: User-defined note.

- Positions
  - **Position ID**: An ID of an existing position, must be left empty to create a new position.
  - Instrument ID: ID of the instrument represented by this position.
  - Portfolio ID: ID of the parent portfolio.
  - Note: User-defined note.
  - Time: UTC date and time of the initial transaction in `MM/DD/YYYY HH:mm` format.
  - Amount: Amount of the initial transaction.
  - Price: Price of the initial transaction per 1 unit.
- Transactions
  - **Transaction ID**: An ID of an existing transaction, must be left empty to create a new transaction.
  - Position ID: ID of the parent position.
  - Price: Price of the transaction per 1 unit.
  - Amount: Amount of the transaction, positive amount indicates a purchase, negative amount indicates a sale.
  - Time: UTC date and time of the transaction in `MM/DD/YYYY HH:mm` format.
  - Note: User-defined note.
- Instruments
  - **Instrument ID**: An ID of an existing instrument, must be left empty to create a new instrument.
  - Symbol: Instrument's ticker.
  - Name: Name of the instrument.
  - Exchange: Exchange at which the instrument is traded.
  - Type: Instrument type, allowed values are `stock`, `bond`, `mutual fund`, `etf`, `commodity`, `cryptocurrency`, `index`, or `other`.
  - Currency: Instrument currency as a three-letter currency code, such as *USD*.
  - Note: User-defined note.
- Instrument prices
  - **Instrument ID**: ID of the parent instrument
  - Price: Instrument price at the provided time.
  - **Time**: UTC date and time of the price in `MM/DD/YYYY HH:mm` format.

Example templates are available in the `test_data/` directory of the electronic attachment.

A.3.8 Currency management

By navigating to the `Currencies` page (Figure A.11), the user can view the current available exchange rates and configure the application-wide default currency.

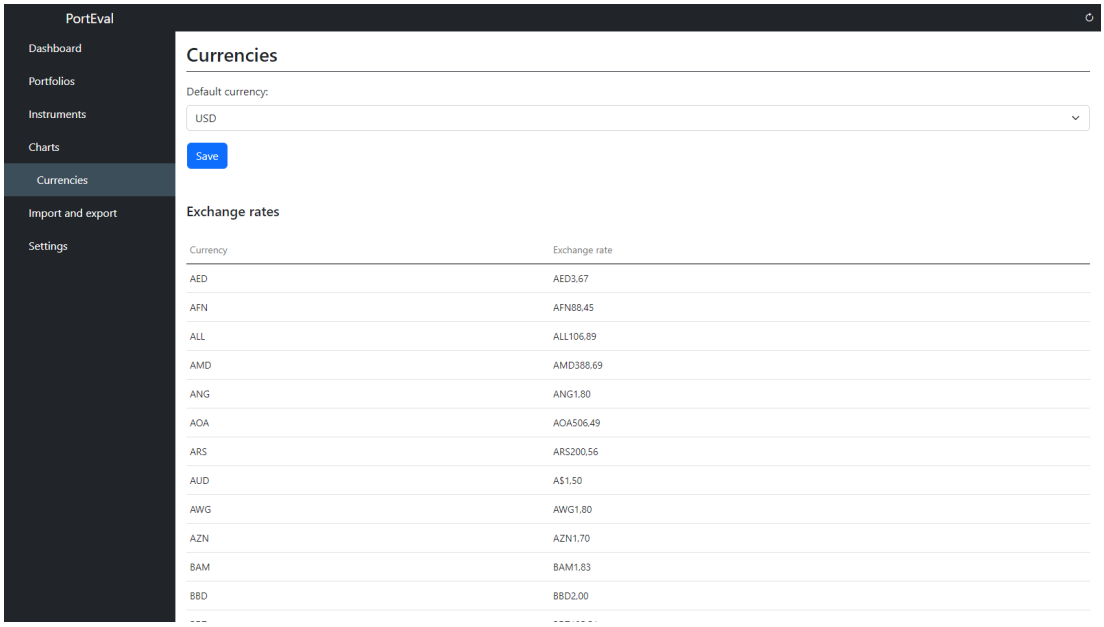

Figure A.11: Currencies page

The application only maintains the exchange rates for the selected default currency. If the default currency is changed, then the application will automatically attempt to download the exchange rates for the newly selected default currency.

A.3.9 User settings

The `Settings` page (Figure A.12) allows the user to change their formatting settings.

The following settings can be configured:

- Date format: format to be used when displaying dates or parsing entered dates in date selectors. The following placeholders are allowed:
  - `d`: one-to-two-digit day of the month, such as 9 or 24
  - `dd`: two-digit day of the month, such as 09
  - `M`: one-to-two-digit month number, such as 7 or 12
  - `MM`: two-digit month number, such as 07
  - `MMM`: short month name, such as `Jan`
  - `MMMM`: full month name, such as `January`
  - `yy`: two-digit year number, such as 23
  - `yyyy`: four-digit year number, such as 2023
- Time format: format to be used when displaying times or parsing entered times in date selectors. The following placeholders are allowed:

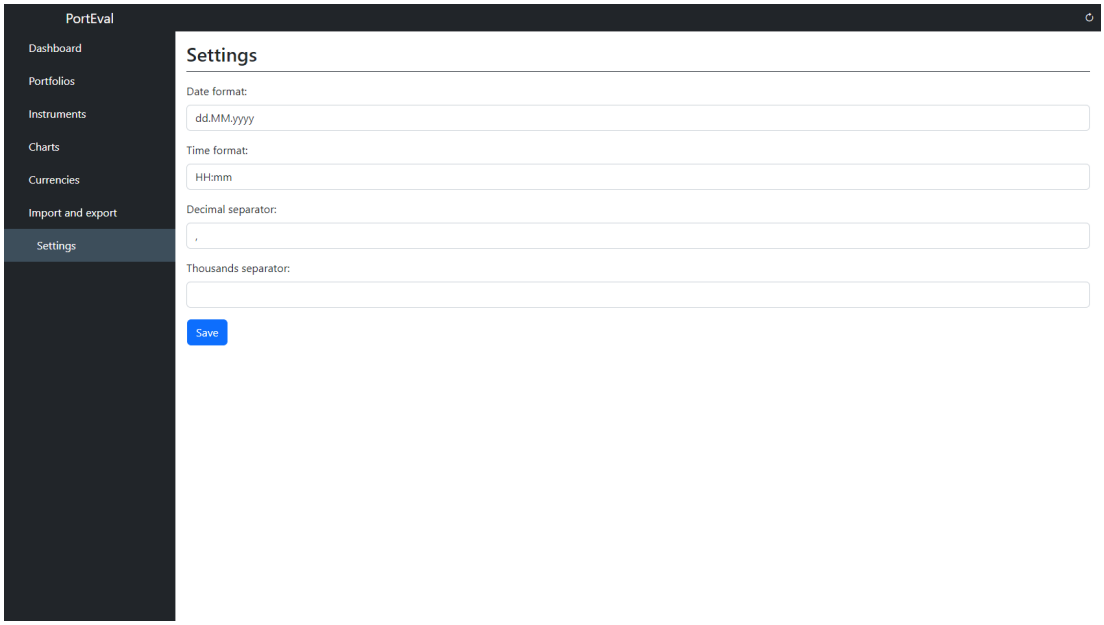

Figure A.12: User settings page

  - `H`: one-to-two-digit hour in 24-hour format, such as 9 or 23
  - `HH`: two-digit hour in 24-hour format, such as 09 or 23
  - `h`: one-to-two-digit hour in 12-hour format, such as 9 or 11
  - `hh`: two-digit hour in 12-hour format, such as 09 or 11
  - `mm`: two-digit minutes, such as 55
  - `aa`: `AM` or `PM`
- Decimal separator: character(s) to be used for decimal points.
- Thousands separator: character(s) to be used to separate thousands in larger numbers.