Aalto University

CS-C2120 - Ohjelmointistudio 2

Project document

Isak Carrero

100829440

Information Networks
2nd-year student (started 2023)

April 28st 2025

1. General description

I have built an interactive stock data dashboard that has a wide range of different functionality using Scala. In the dashboard, users can create multiple portfolios with distinct names. To each portfolio, the user can add stocks in just a few clicks of a button and by inserting data of the stock, such as the stock's ticker, the purchase price per share, the number of shares purchased, and the date of purchase. The dashboard allows the user to keep track of their equities in a smooth manner, easily.

Using the inserted stock data in the portfolios, the user can visualize and analyze it with the help of the dashboard's methods. These visualisations and methods to analyse the data are displayed on the big tiles (cards), in the form of charts or information cards, when the user inserts them. These include scatter plots, column charts, and pie charts, which help users follow historical stock data, allocations within a specific portfolio, and historical purchase dates and prices of the stocks in one or many portfolios. In addition to these charts, the information cards allow the user to follow their portfolio's growth and value, in the form of weekly/total gain and total value. Data fetching from files/API supports these visualizations and portfolio/stock analysis. In this project, the Alpha Vantage Stock API was used.

The project has been completed on a demanding level and includes all the advanced methods in order to enhance the dashboard's usability. It handles loading and refreshing dynamic data from either the pre-downloaded JSON files or from the API, depending on the user's wants and needs. It contains personalized color selection for charts and gives the user the ability to visualize multiple portfolios at once, using the multiple-series scatter plot. Additionally, the dashboard allows the user to gain a better understanding of displayed datapoints by hovering or clicking on them. Furthermore, the users can choose if they want to see historical data for a stock in the timeframe of one week or even three months, which allows personalized data visualization preferences. Most importantly, the dashboard has a clear UI, with helpful descriptions, that improves the user experience and understandability.

Overall, the design has been focused on creating a clean, efficient, and responsive user interface that allows users to easily track and analyze stock data interactively and engagingly.

2. User Interface

ScalaFX was used to create the UI for the stock data dashboard, as it contained ready methods and components that were useful for the project. The usage of ScalaFX made it easy to create a dashboard that gives the user a clear understanding of how the functionality works, while being neat and customizable.In order to run the dashboard,

the user has to, in IntelliJ, navigate to the Main file. Here, the user can find the *Main* object that extends *JFXApp3*. By pressing the green arrow next to the object, the program is executed, and the Stock Portfolio Dashboard window opens with the full user interface ready for interaction.

The dashboard uses a BorderPane layout, which makes it easy to initialize the location of components. *Image 1* shows the initial state of the dashboard. The top Pane includes the menubar, which includes functionality for saving/loading files, creating portfolios, and providing descriptions for the users. On the left-hand side, there is a sidebar. In this sidebar, the created portfolios are displayed together with their stocks. The central dashboard area is reserved for the four card tiles. In these card tiles, users can insert charts or information cards, based on what they want to visualize or analyse, and resize the tiles based on their preferences.

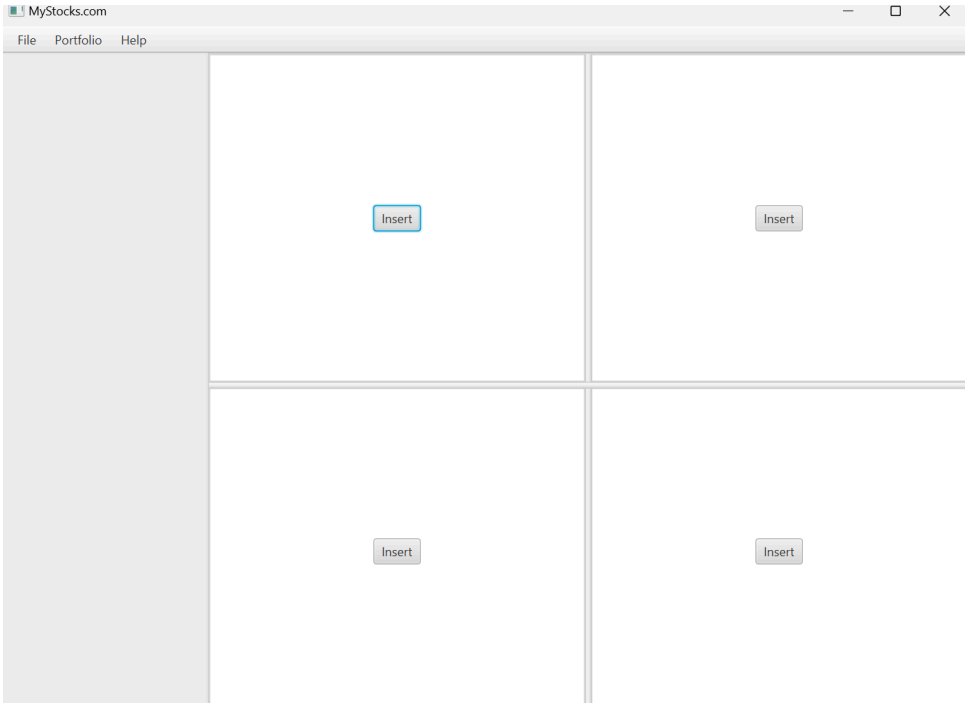


Image 1: Initial state of the dashboard.

In the menu bar, there are three visible options: “File”, “Portfolio,” and “Help”. By pressing File, the user can either save the current state of the dashboard to a CSV file or upload a CSV file, given that the contents are in the right format. The Portfolio option allows the user to add a portfolio to the dashboard with a unique name, which gets inserted into a text dialog. The last option, Help, includes guidance text for helping the user understand how to add or remove components in the dashboard.

On the left, the sidebar enables portfolio control. The portfolios that get created by the user with the help of the functionality in the menu bar get added to the sidebar. The user can add stocks to a specific portfolio by pressing “+” next to the corresponding portfolio (see *Image 2*). When the button is pressed, a dialog pops up on the screen. Here, the

user inserts the stock ticker, price of purchase per share, number of shares that have been purchased, and the date of purchase. The Stocks then appear underneath the portfolio to which the stock has been added. The stock within a portfolio can be hidden or viewed by pressing the "▼" in the portfolio container. By pressing the "Del" button with text, the user can delete a whole portfolio. If a large number of portfolios or stocks are added to the sidebar, a scrollbar feature appears to handle the overflow.

The center of the application consists of a dashboard made of resizable tiles arranged in a grid layout. Charts or information cards can be inserted dynamically by clicking "Insert" buttons on empty tiles, then choosing from a dialog what they want to visualize, and finally choosing the specific data that should be used. There is a total of four tiles/cards, and each can contain one type of visualization at once (see *Image 2*).

For creating an information card, scatter plot, or pie chart, the user only has to choose from a drop-down list which portfolio should be used in the chart. But when it comes to the column chart, the user has to write in the ticker of the stock that they want to visualize, and choose from the color picker what color they want the chart to be. If they do not specify a color, the chart becomes automatically white.

With the column chart, the user can visualize a specific stock's historical data. The column chart has a time series so that the user can see the historical data for the time period of one week, two weeks, one month, or three months. When hovering over the column, additional information gets displayed. The information card displays information about a specific portfolio. On the card, the user can see the weekly/total growth of the portfolio, the number of stocks that it contains, the total value of the portfolio, and the largest allocation. The multiple-series scatter plot shows the historical purchases of one or many portfolios. On the x-axis, it displays the dates, and on the y-axis, it displays the price per share that the shares were bought for. When hovering on a data point, additional information is displayed, such as the stock ticker, which portfolio the data point is in, the purchase price, etc. When clicking on the data point, the user can see the latest closing price of the stock. The scatter plot has legends so that the user can see which portfolio each data point belongs to. Finally, the pie chart visualizes the allocation within a portfolio. It shows how much of the total spending each stock has accounted for.

When a chart or information card is inserted into a tile, it gets put into a close wrapper. The close wrapper gives the charts/cards an "x" in the top right corner so that they can be removed and new data can be inserted. This allows for flexible usage of cards.

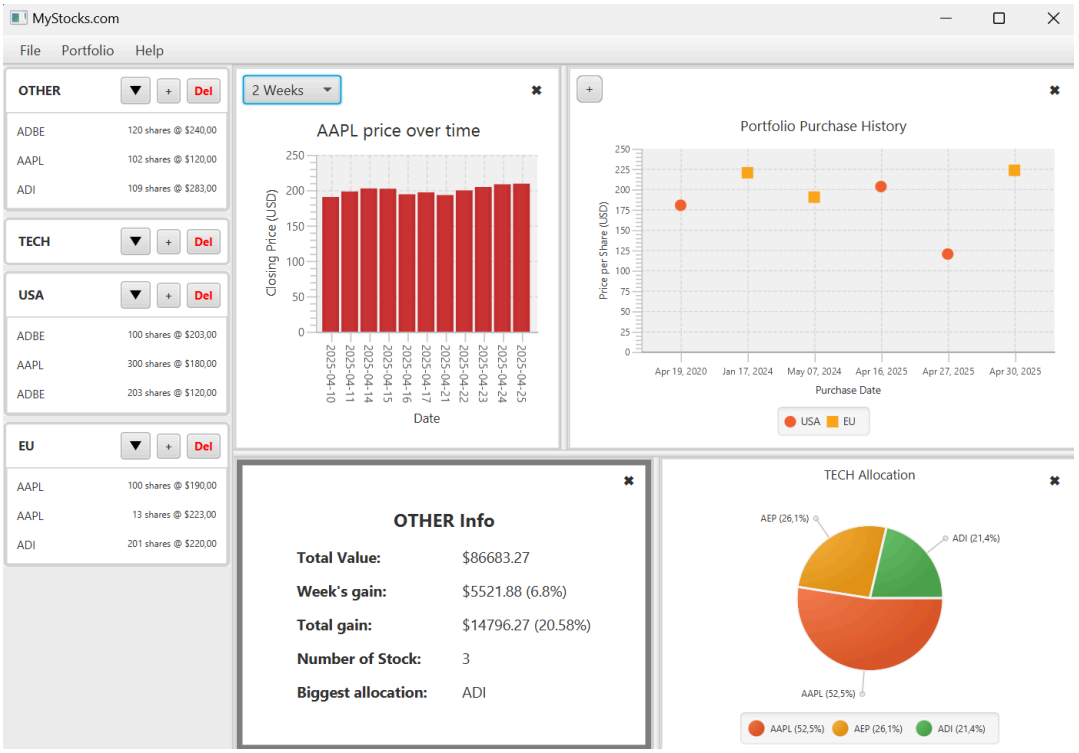


Image 2: Dashboard with inserted data.

The dashboard ensures modularity and flexibility. Users can mix different chart types, resize cards independently, and maintain an organized view even with many visualizations at once. All cards can contain any type of chart, which means it is easy to duplicate charts, as they only need a given portfolio from the user to be visualized.

Error handling is built into the user interface to prevent confusion. Informational popups appear when invalid information has been inserted/selected and when delicate operations are about to be executed (deleting portfolios).

3. Program structure

The program was split into three sub-parts: The Main, a folder for Visuals, and a folder for Data. The final realized class structure is modular; each class or object focuses on a clear responsibility within the application.

3.1 Main

The main object is where everything is connected and comes together. It is a subclass of JFXApp3. The Borderpane layout and everything that is included in it is in the main object. This includes the menu bar, sidebar, and the center field where all the charts are displayed. The main handles the interaction with the user and sees to it that information is translated in the right way.

3.2 Data

The data folder intuitively includes all the classes and objects that handle data. The Datafetching class uses Scala.io, Java.io, and Java.net to make API calls and store the data into JSON files in a predetermined folder. A helper function has been defined that allows the user to specify which stocks they want to retrieve data.

The PortfolioManager object handles the storage of stock and portfolio data. It uses two case classes, one for separately storing inserted stock data and one for storing the stock data within a specified portfolio. The Portfolio case class is then stored in a linked hash map with all the other portfolios, which allows easy tracking of portfolios.

The StockDataParser object parsed data from JSON files or API calls into usable formats.

3.3 Visuals

In the visuals folder, there is the Card object that is the cornerstone of the dashboard chart visualization. It is in the four cards that are defined in the Card object that the charts get displayed. The Card object constantly keeps track of the states of the distinct cards. It is built up using a SplitPanes that allows the user to resize the cards into the desired sizes.

Piechart, Columnchart, Scatterplot, and Portfolioinfo are the four other classes in the Visuals folder, and the classes that can be used to display information in the cards.

The Piechart class is responsible for visualizing the allocation of the invested money. In it, weighted costs for each stock are calculated, using the portfolioio information in PortfolioManager, and then used to display a pie chart, featuring legends.

The Columnchart class is used to visualize a specific stock's historical closing prices. It uses the StockDataParse object to retrieve the historical price, which it displays according to the color given by the user.

The Scatterplot Visualizes the historical purchases of one or many portfolios. It gets the information from the PortfolioManager object and uses it to display the points on the charts. It also uses the StockDataParses to get the latest closing prices of the stocks, which can be seen when clicking on a data point.

The PortfolioInfo class displays how a specific portfolio is performing. It shows the Weekly/total gain, total value, number of socks in the portfolio, and the largest allocation by cost. It uses the PortfolioManager and StockDataParser to display information.

During the project, other structures were considered. One idea was to let each visualization class also handle its own data fetching. That would have meant mixing data handling and visualization in the same classes. In the end I decided to keep them separate, with all data handling in the Data folder and all visuals in the Visuals folder. This made the code clearer, easier to test, and easier to expand later.

4. Algorithms

The dashboard uses several algorithms to handle stock data, portfolio management, and data visualization. Below are the main algorithms used in the project.

4.1 Calculations

The dashboard contains a few calculations that are used to display the needed data on the charts or charts. In the information card, algorithms that calculate the total value, total gain, and weekly gain are used. For the pie chart, the weighted cost for each stock is calculated and then used when displaying a portfolio's allocation of costs. They function as follows:

For the information card:

$$Total\ Value\ =\ \sum(Latest\ price\ \times\ Number\ of\ shares)$$

$$Total\ Cost\ =\ \sum(Buy\ \ price\ \times\ Number\ of\ shares)$$

$$Total\ Gain\ =\ Total\ Value\ -\ Total\ Gain$$

$$Total\ Gain(\%)\ =\ (1\ -\ (Total\ Value\ /\ Total\ Gain))\ \times\ 100$$

$$Weekly\ Gain\ =\ \sum((Latest\ \ price\ \ -\ \ Price\ a\ week\ ago)\ \times\ \ Number\ of\ shares)$$

For the pie chart:

$$Weighted\ percentage\ of\ stock\ cost\ (\%)\ =\ ((Buy\ price\ \times\ \ Number\ of\ shares)\ /\ Total\ Cost)\ \times\ \ 100)$$

4.2 Data parsing

To display the stock data, the dashboard has to fetch and parse data, either from a pre-downloaded JSON file from the API or by calling the API directly. The user chooses this by setting the value useAPI to true or false. The algorithm fetches data for one specific stock at a time and extracts the closing prices. It returns the N latest closing

prices, depending on what the data is needed for. For instance, if it is for the column chart with a period of one week, then it N=6 (there is no data for weekends, as the market is then closed). For the purpose of this simple dashboard, this way of fetching the data is smooth and efficient, but for larger uses, this would not be optimal, as it would not be that fast. An option could be to store it in a database, but that would be a more complicated solution.

If the dashboard is used with an API that has unlimited API calls, then there would be no need to use files. But as the API only has 25 calls per day for the free version, I created a way of storing the data in JSON files. The getStockData method takes a ticker as a parameter and then fetches the data. It then creates a file in the format of [stock ticker].json and stores it in the folder StockAPIDataDaily. If the [stock ticker].json file already exists, then it updates the file with the latest data. These files can then be used instead of API calls to run the program and visualize charts.

4.2 Saving and loading dashboard state

In the Main, there are algorithms for saving and loading a dashboard state. When saving the dashboard state, the algorithm gets all the portfolios that are displayed on the dashboard from the PortfolioManager object, and all the displayed charts and information from the Card object. The data is formatted and then written into CSV files with portfolios and stock information stored in rows like this: portfolio,ticker,date,price,amount. Charts are stored in rows like this: chartType,portOrStock,color.

When a file is then uploaded to the dashboard, the loadData method first splits the data in the files so that the information about portfolios and charts is separated. The algorithm first clears all current information from the dashboard so that new data can be inserted. Then it adds all the portfolios (and all the stocks in those portfolios) into the dashboard, followed by all the charts and other information.

5. Data Structures

In this project, data is primarily fetched from an external API and then processed and stored in files or used directly after parsing. Fetched data consists of stock price information in a JSON format as seen in *Image 3.* When the data gets parsed for a specific stock, it gets stored in a list, List[(String, Double)], with the date and closing price as parameters. This was the easiest way to get the data, given the JSON structure. I did consider using a Map to map the tickers with the closing price and date, but in this project, I did not need that type of functionality

The main data structure used to hold stock information is the StockData case class. Each StockData object contains fields such as ticker (the stock's ticker), amount (the number of shares), price (the price per share), and purchaseDate (the date of the transaction).

The StockData is stored within the corresponding portfolios case class Portfolio(String, mutable.Buffer[StockData]), which takes as a parameter the portfolio's name and the list of stocks in the portfolio. The Buffer is mutable, so that stocks can be added to it. This information is then stored within val portfolios: mutable.Map[String, Portfolio]. This data structure was chosen as it allows for easy adding and removing of portfolios, and retrieving stock data in a specific portfolio.

ObservableBuffers is used, for instance, to give the pie chart the data that is needed to be displayed, and to control what portfolios are visible in the scatterplot, and which should be displayed in the choice box. It was used because it is good at handling changing data.

I used a case class CardState in order to easily follow what information is displayed in which chart. CardState takes as parameters what kind of information is displayed in the card, the name of the stock or the portfolio that is displayed, and if it has a specified color. This information then gets stored in the cardStates array. The cardStates array has four predefined slots, one for each card. When something happens in a specific card, the information gets updated within the cardStates. This makes it easy to load the saved data back onto the dashboard, as every chart has a known position.

6. Files and Internet access

If the user uses an API that does not have unlimited or a large enough number of calls per day, then they can store the information in files. In the project, I stored the API data into .json files to that I could easily use it for testing. The Datafetching class in the Data folder was solely created in order to save data into .json files. The formatting of the data can be seen below in *Image 3.*

Image 3: JSON format of stock data. NOTE: This is not the full data; the data continues so that each date has its own JSON object, as seen in the image

If the user has set the useAPI to false, and displays, for instance, a column chart, then the getClosingPrice method in StockDataParses first checks if there exists a JSON file for the specific stock ticker. If there is a file, then data is retrieved from there, but if there is no file, then the method calls Datacfetching in order to save the data into a file and then uses it.

The program accesses the internet using the URL class from Java to make an HTTP request. The URL used in the getStockData method is a URL string starting with https://, which indicates that the program is making a secure HTTP request using the HTTPS protocol.

When saving the dashboard, the saveData method first writes the portfolio and stock information. Each stock entry includes the portfolio name, stock ticker, purchase date, purchase price, and the number of stocks purchased. The data is saved under the header "portfolio,ticker,date,price,amount". After all stocks have been entered, the saveData method saves the chart configurations for the dashboard. This section starts with the header "chartType,portOrStock,color" and records the type of chart, the portfolios or stocks displayed, and the selected chart color (for column charts only). If a scatter plot displays multiple portfolios, they are separated by the "|" character.

Here is an example of how the data is stored:

portfolio,ticker,date,price,amount
USA,ADBE,2024-04-19,240.0,120
USA,ADI,2025-04-13,283.0,109
EU,AAPL,2025-04-02,201.0,90
EU,AAPL,2024-04-02,189.0,232
chartType,portOrStock,color
ColumnChart,AAPL,#CC3333
„
„
scatterPlot,EU|USA,

The data gets stored in CSV files. The charts are saved in the order that they are displayed on the dashboard, the first one is in card 1 and the last one in card 4. The double comma „," means that there is no chart or information in that card.

7. Testing

The program was tested continuously during my development process using a combination of system testing, manual testing, print-based testing, and functional testing on a small scale (one function at a time). The goal was always to verify that

individual components behaved as planned and to ensure that the full dashboard worked in a reliable way.

7.1 Testing Methods

7.1.1 System and Manual Testing

System testing was performed continuously by running the full application after implementing major features or changes. This included testing the addition and removal of stocks in portfolios, updating charts with new data, and saving and loading portfolio information from files. The goal was to ensure that the system behaved as expected in realistic user scenarios.

Manual testing through the graphical user interface (GUI) played a key role. Different features, such as opening multiple portfolios, inserting charts, resizing and closing components, and adjusting colors, were carefully tested. Special attention was paid to verifying that the visual elements remained functional and responsive throughout development.

While the dashboard automatically marked faulty code with visual red highlights when errors occurred (such as exceptions or runtime problems), this feature supported but did not replace structured system testing.

7.1.2 Exploratory Testing and Print-Based Debugging

To better understand the internal behavior of the application, extensive use of print statements was made, particularly in the Data package classes. Methods such as getAllPortfolios, getStockData(ticker: String), and getClosingPrices(ticker: Int, days: Int) were frequently tested to verify that portfolios, stock data, and parsed information were handled correctly and stored in the intended formats.

This exploratory style of testing helped detect mistakes early and allowed for debugging without relying only on GUI behavior.

7.1.3 Graphical Testing

Graphical testing focused on the visual correctness of the dashboard under different user actions. This included observing how charts updated after data changes, how components responded to resizing or closure, and whether user actions such as inserting multiple charts behaved as intended. Visual errors like missing updates, incorrect chart sizing, or broken interactions were detected and fixed iteratively.

7.1.4 Comparison to the Initial Testing Plan

The original testing plan stated that testing would be conducted through a combination of system testing and unit testing. The plan detailed that unit testing would cover:

- Calculating portfolio values and returns
- Handling edge cases like empty portfolios and drastic price changes
- Sorting historical stock prices
- Processing data from the API

In reality, testing evolved a bit and relied more heavily on manual system testing, print-debugging, and interactive graphical testing rather than formal unit tests. Instead of traditional isolated unit tests, issues were caught early by continuously running the program, observing outputs, and doing tests on individual components when I saw fit.

The program passes all functional and system tests that I have conducted. There are no significant untested parts, although some methods tied to the GUI were not separately unit tested, as their behavior could be directly observed through the graphical interface. These include, for instance, the charts and the information cards.

8. Known bugs and missing features

The program is not completely perfect, and there are some features and bugs. I did not have time to implement the rectangle selection tool as I did not have time due to my laptop breaking down, and not having one for one and a half weeks. Additionally, the duplication method has not been implemented as I did not find a suitable way to use it. The dashboard is created so that it is easy and quick to use, so for instance, creating two of the same charts would take a few seconds.

There are a few bugs in the scatterplot. If the first two portfolios that get added to the scatter plot have long names, then the legends of the chart format weirdly. Additionally, for some unknown reason, when many portfolios are added to the scatter chart, it becomes blank.

There are some control functions for checking the format of different information, like when uploading files or displaying information that could have been implemented. Currently the some charts don’t display anything if wrong information is inserted,

Additionally, adding really large numbers when inserting stock information results in error, even if it is a valid number.

9. 3 best sides and 3 weaknesses

9.1 3 best sides

9.1.1 Modular design

The program's design is well-structured, with distinct classes and objects in different files. By having the functionality split into different locations based on their functionality, the understandability of the full program increases. This not only makes the files easier to read, but also easier to edit, as errors in one file will not necessarily influence the functions in other files (if they are not connected). For instance, making changes in PortfolioManager will not affect Datafetching.

9.1.2 Saving and loading the dashboard

The ability to save and load the dashboard makes it increasingly easy for users to use the dashboard. Thanks to this functionality, the users don't have to insert the same data every time they run the program. One aspect of the saving and loading functionality that is exceptionally good is the ability to place charts into the same card that they were in when the dashboard was saved. Additionally, all the displayed data remains the same throughout the process.

9.1.3 Portfolio management and display

The program offers users the option to create and manage multiple portfolios, which is a key feature for users who want to organize and track various portfolios separately. By allowing the user to add and remove portfolios as they see fit, the flexibility of the program increases. Additionally, the portfolio information is displayed clearly in the sidebar, making it easy for users to see what data each portfolio contains.

9.2 3 weaknesses

9.2.1 Adding stock

When adding a stock to a portfolio, the program does not check if it is a real stock. This is bad as inserting an incorrect stock ticker leads to methods like creating the information card not executing, due to it not being a real ticker. In order to minimize these errors, a function that checks that the inserted string is 1-5 uppercase letters in the range of A-Z. This is how stocks within the NASDAQ are defined. If an API with limited calls were used, then a more advanced checker could be implemented. This could be done by calling the getStockData method in the Datafetching class, and controlling if the correct data gets stored into the file.

9.2.2 Duplication method

The dashboard does not have functionality for duplicating, for instance, pie charts. This can be seen as a weakness, as it was one of the criteria. However, I have focused on creating the offers' quick and easy visualization of inserted data. This means that if the

user wants to duplicate a pie chart to any of the other cards, then it only takes five clicks and five seconds to do so. This means that duplication would not have improved the usability in any drastic way, and might have even led to a more confusing interface with excessive buttons.

9.1.3 Scatter plot

The scatter plot is a bit of a weakness. It has some bugs with the legends and with displaying information. It sometimes becomes blank if too many portfolios are displayed and the legends become misaligned if the portfolio names are too long

10. Deviations from the plan, realized process, and schedule

During the first sprint, I set up the project as described in A+. I familiarized myself with some libraries and tested out some functionality, like loading and running the dashboard. Nothing more was done under the first sprint.

During sprints 2–4 I focused heavily on getting the layout of the dashboard. I implemented the sidebar and the menu bar, and I watched a lot of videos on how to code a dashboard with ScalaFX. I additionally searched for APIs that would be suitable for my project. Here, I still followed my time schedule as planned. I had a lot of other ongoing courses, so not that much time was dedicated to the project at this point.

In the following sprint, I had more time to dedicate to the project. I started implementing the functionality for adding portfolios and stocks. I also worked a lot with data handling. I did a lot of testing, as I wanted to ensure that the program worked properly at an early stage. I started working on the cards and tested different ways of designing them.

The sprints that followed had a pretty evenly distributed workload. At this point, I did not really follow the plan that much anymore. I did what was relevant to get the dashboard progressing. Here, I started to implement charts, which was a challenge. A lot of research and watching videos were done at this point. A lot of trial and error as well.

I missed one whole sprint as my computer broke down, which was not optimal, as I lost a lot of valuable time. Due to this, the last two sprints were very hectic. A significantly larger amount of time was dedicated to these sprints than expected.

If i would start again, I would start to do heavy coding earlier. I had a soft start, which led to a stressful ending.

11. Final evaluation

The stock data dashboard project has resulted in a functional and interactive tool that allows users to track and analyze their stock portfolios easily. The implementation meets most of the advanced requirements, though there are areas that could be improved to enhance UX.

The dashboard's strengths are its modular design and intuitive interface, while some weaknesses exist in data input. One of the project's strongest aspects is its well organized structure. By dividing the code into distinct packages for data handling, visualization, and the main application logic, the program remains clear and maintainable.

The user interface is straightforward, with helpful tooltips and helper descriptions. The ability to dynamically insert and resize charts adds to the dashboard's flexibility makes the it customizable. The rectangulation was left out, which was not optimal.

Looking ahead, several improvements could elevate the dashboard further. Implementing proper input validation, such as checking stock tickers against a known list or ensuring numerical inputs are within reasonable bounds, would prevent many common user errors. Fixing the scatter plot's rendering issues, particularly with legends and large datasets, would enhance reliability.
This project was really fun and i wish I would have dhad more time to work on it!

12. References and code written by someone else.

When doing my project, I constantly had to search for solutions and inspiration on how to implement my desired methods. If I had to guess, I would say that over half of the time spent on this course was dedicated to doing research. I never came across an implementation that exactly corresponded to the code that I needed. There is very limited information about ScalaFX and Scala on the web. Because of this, I got a lot of my inspiration from JavaFX and Java coding posts on the web. I found that Stack Overflow and GitHub were the most helpful websites for finding help.

Alerts: In my code, I use a lot of Alerts to enhance the user's understanding when interacting with the dashboard. I found on the website GeeksforGeeks a very comprehensive overview of the different alert types, which helped me when completing my project

- Link to the post: https://www.geeksforgeeks.org/javafx-alert-with-examples/

Choosing and understanding Panes: In the beginning, it was fairly hard to understand what type of Pane I should use in my project. Fortunately, I found a helpful video by JavaHandsOnTeaching that explained how they worked.

- Link to the video: https://www.youtube.com/watch?v=GH-3YRAuHs0

Changing color to hex code: In the Card object, it can be seen that i have implemented a val hexColor, which takes in the value retrieved from the ColorPicker and translates it into hex code, so that the column chart can use it to display the right colored charts (see line 191 in Card). This solution was highly inspired by Moe, who commented on a post on Stack Overflow.

- Link to the post: https://stackoverflow.com/questions/17925318/how-to-get-hex-web-string-from-javafx-colorpicker-color

12.2 External libraries used

ScalaFX: As mentioned, ScalaFX was widely used in the project. I gained an understanding of the library by reading the ScalaFX documentation and by watching videos of a YouTuber named Mark Lewis. I additionally used the JavaFX documentation, as I felt the tit was better then the ScalaFX documentaton that i found.

- Link to ScalaFX documentation:
  - https://scalafx.org/docs/dialogs_and_alerts/
  - https://scalafx.org/api/8.0/index/index-p.html
- Link to Mark Lewis' YouTube channel: https://www.youtube.com/@MarkLewis
- Link to JavaFX documentation: https://docs.oracle.com/javase/8/javafx/api/

Alpha Vantage: I used Alpha Vantage's stock API to fetch data that I could use in my dashboard. On the website, there is clear documentation of how the API works. Additionally, they have linked a GitHub where thousands of users have shared how they use the API. From this GitHub, I got inspiration on how to fetch and parse the data.

- Alpha Vantage website: https://www.alphavantage.co/?gad_source=1&gbraid=0AAAAA-6A8dPIAg6uH3xAj2hy1Fi8NuJHt&gclid=CjwKCAjwq7fABhB2EiwAwk-YbIlN9FKZ8qUUgvXnJYW8mWWtzqvIxR2a_a9fgEgjr766opRidWXTshoCsgQQAvD_BwE
- GitHub community: https://github.com/search?q=alpha+vantage&type=repositories

Java: In some methods, I used Java packages as it was earlier and quicker for me to implement the methods using them. I used, for instance, the Java.io for writing files, Java.net when handling the API calls, and Java.time for handling time for the charts. All these packages I found in the Java documentation on Oracle's website.

- Java documentation: https://docs.oracle.com/javase/8/docs/api/overview-summary.html

Json4s: For parsing the data, I used the Json4s parsing library. This made it very easy to parse and store the data from the API. I found a very comprehensive documentation on GitHub, which I used for inspiration.

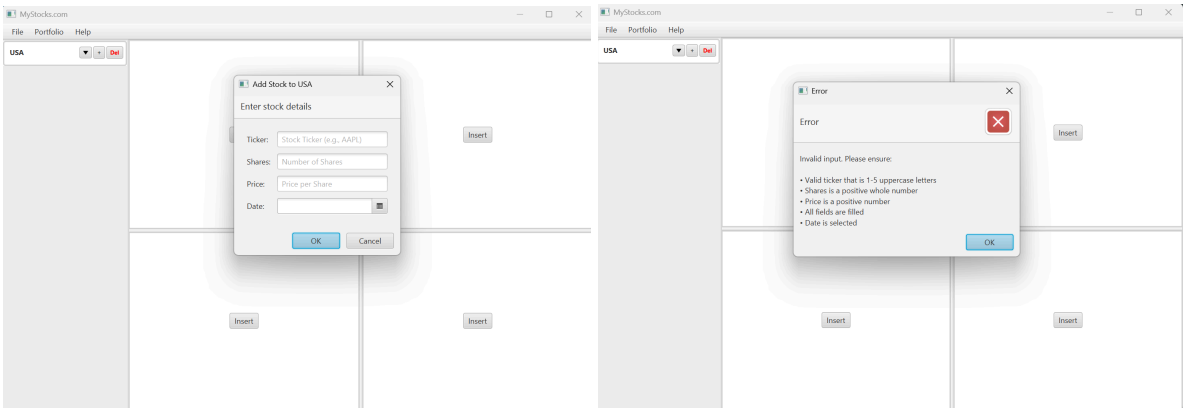
- Json4s documentation: https://github.com/json4s/json4s

12.3 Use of AI

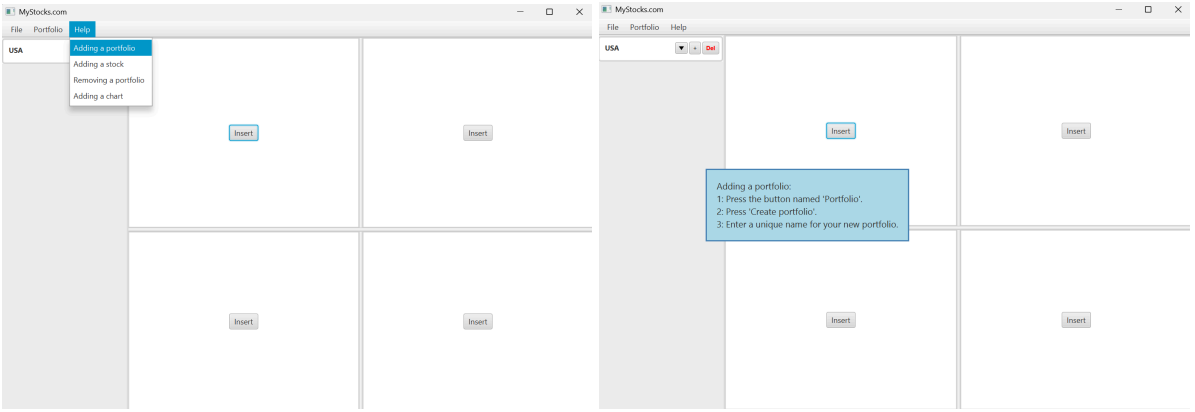
During the project, I used ChatGPT as a tutor. When there was a library that I did not understand how it worked, I asked ChatGPT to explain it to me in a simpler manner. Also, when I came across solutions with coding languages, of which I did not understand the syntax, I asked ChatGPT to write it in pseudocode so that I would understand it. ChatGPT also helped with explaining how styling methods work for the ScalaFX components, so I did not have to return to the documentation every time I wanted to make styling changes or implementations. No code was directly copied from an external source into my project.

13. Appendices

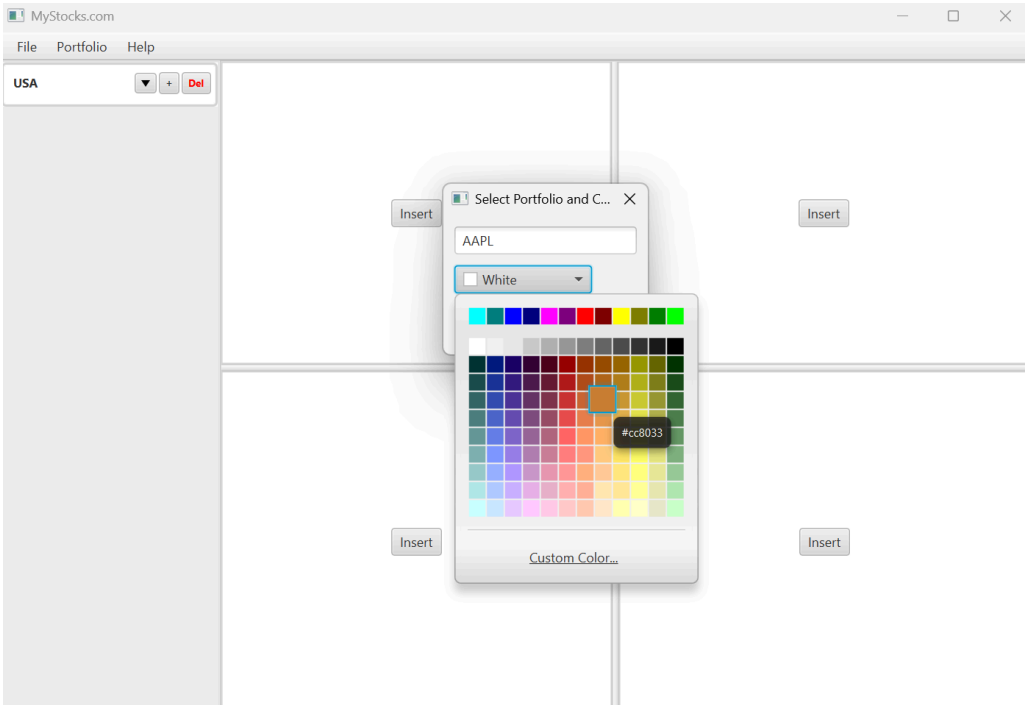
When pressing the "+" button in the portfolio container, the dialog in the left image appears. If the inserted information is not correct, then the descriptive error seen in the right-hand image appears:



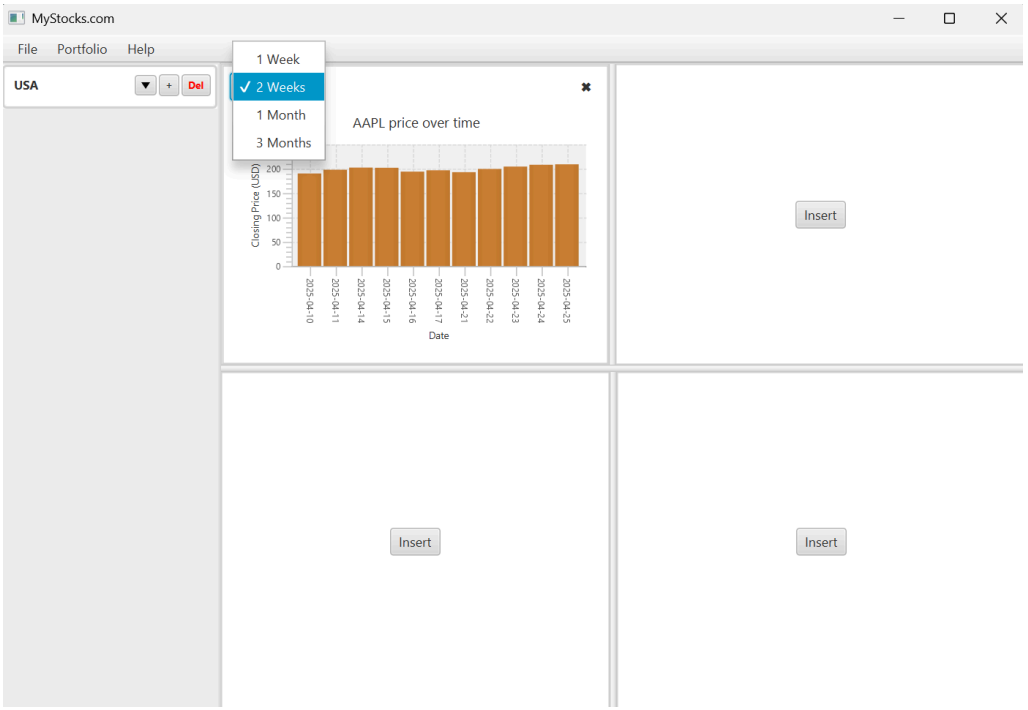
When pressing the help button, a drop-down list with different help options appears:



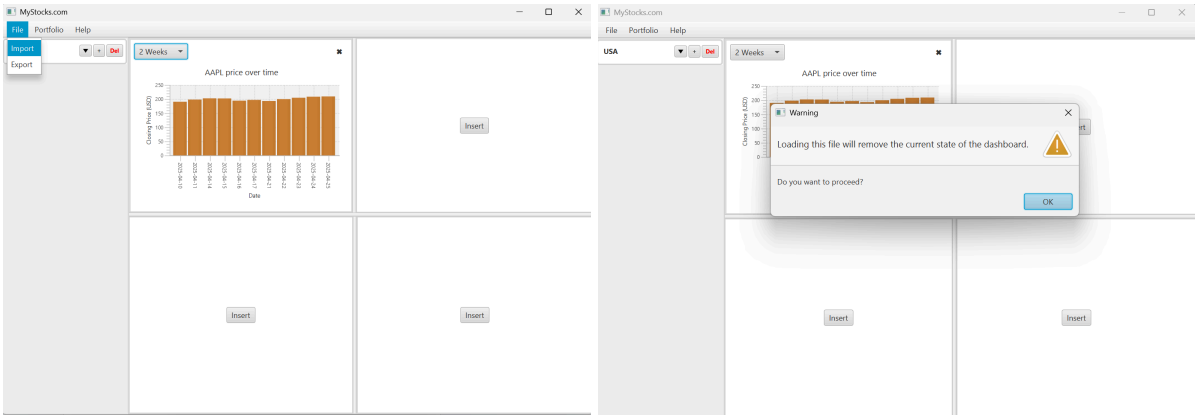
The operation Insert → Column will get you to this state. Here a ticker is inserted and a color chosen:



The chart is then displayed (given that the stock ticker is valid).



The user can import files by pressing Files → Import. When the user has selected a file, the program displays a warning to inform the user that the current state will be discarded:



An imported dashboard can, for instance, look like this: