2021

Stock trading simulation

A-LEVEL COMPUTER SCIENCE COURSEWORK

ARTEM STRELTSOV

Table of Contents

Analysis *3*
Project definition 3
Identifying suitable stakeholders 4
Existing systems 4
Underpinning knowledge and calculations 8
Stakeholder and user needs 13
Identifying and explaining any limitations 14
Computational methods 14
Specifying software and hardware requirements 15
User requirements and measurable success criteria 15
Design *17*
Breaking the problem down systematically 17
Explanation of each module 19
Interfaces 20
Validation of inputs 21
External files 22
Data structures 22
Algorithms 25
Test data for development 36
Test data for post-development 38
Development *40*
Project setup 40
Milestone №1 – creating the main interfaces 40
Milestone №2 – selecting a level 50
Milestone №3 - displaying the wallet and level requirements for each stock 51
Milestone №4 – drawing a candlestick chart 55
Milestone №5 – drawing a moving average chart 58
Meeting with stakeholders 63
Milestone №6 - highlighting and annotating candlestick patterns 63
Milestone №7 – highlighting and annotating intersections of moving averages 68
Milestone №8 – allow the user to trade the chosen stock 73
Milestone №9 - allow the user to hide/show candlestick patterns 76
Milestone №10 - allow the user to stop the simulation and view results 78
Milestone №11 – allow the user to change the type of moving average and its parameters 80
Meeting with stakeholders 83
Evaluation *84*
Post-development testing 84

Evidence for post-development testing **85**
Success criteria evaluation **99**
Usability features **100**
Limitations and improvements **100**
Maintenance **102**
Code listings ***103***
Bibliography ***124***

Analysis

Project definition

Inflation is a rise in the price level in an economy over some time. The standard annual inflation rate is considered to be 2-3% (Amadeo, 2021). It means that every year the purchasing power of the money in your bank account decreases if not invested.

Firstly, due to high inflation rates, many people are interested in investing their savings in the stock market and making a profit. However, they are afraid to lose money due to inexperience in using trading strategies. As a result, people incorrectly make decisions whether to buy or sell a particular stock and lose money.

Secondly, in most countries people under 18 are not allowed to enter the stock market by the law (Konchar, 2018). In countries that allow children to enter the stock market, this process is usually complicated and involves agreement of parents. However, many economics students are interested in the stock market and want to gain some experience. Therefore, they use simulations on the Internet that are too complicated for beginners and end up not understanding how trading works.

Thirdly, the average annual return in the stock market is considered to be 9.2 - 13.6% (Knueven, 2020). This number shows that the stock market expects traders to wait for a long time to make a significant profit. However, this makes learning experience not as efficient as it could be. Waiting for months until realisation that the used strategy is wrong wastes valuable time.

I intend to create a stock market simulation with a selection of stocks. The aim of the simulation will be to make as much profit as possible while learning how to use different trading strategies. The simulation will be speeded up to ensure the most efficient learning experience. Actual historical stocks prices will be stored in appropriate data structures and retrieved when needed. All stocks data will be downloaded from the Yahoo Finance website: https://uk.finance.yahoo.com.

The key features of the simulation will allow the user to:

1. Choose a level from the selection with different difficulties
2. View wallet, that will include available money, owned stocks and profit
3. View level requirements, that will include initial capital and required profit
4. Buy and sell a number of stocks if the budget is sufficient
5. View a candlestick chart plotted with actual historical data of the selected stock
6. View and learn candlestick patterns that will be highlighted with their names annotated on the candlestick chart
7. View a moving average chart and change its type and parameters if needed
8. View intersection points of two moving averages that will be highlighted with an annotation suggesting whether to buy or sell the stock
9. Change the visibility of candlestick patterns
10. Stop the simulation at any time and view results

The first option is to trade the chosen stock based on candlestick patterns. The program will find and highlight them on the candlestick chart with an annotation indicating their names and suggesting whether to buy or to sell the chosen stock. The user will benefit from using this approach because they will be able to learn some basic candlestick patterns that usually correctly predict the future direction of the price and apply this knowledge in the real stock market to potentially gain profit.

The second option is to trade the chosen stock based on intersections of short-term and long-term moving averages. The type of the intersection, which is either Golden Cross or Death Cross, is likely to predict the future price direction of the stock. The user will benefit from using this approach because they will be able to learn types of intersections of moving averages to correctly predict the future direction of the price of a stock in the real stock market and potentially gain profit.

The solution is needed for beginners to easily start with trading in a format similar to a game. They will learn how to use trading strategies in practice, gain experience much faster than in other simulations and reduce

future risks in the real stock market. In addition, the simulation will be helpful for economics teachers as a practical and entertaining way of explaining trading strategies.

Identifying suitable stakeholders

The first and the most important group of stakeholders is economics students. Many of them are interested in trading, which is not explained in detail at school. Due to the interest, students try online trading simulations, but they are too complicated for beginners to understand. As a result, most of the students waste their time and do not understand how to trade. Misunderstandings and the lack of practice often leads to significant loses of money in the future in the real stock market. However, my stock market simulation will be aimed at beginners and will be simple to use and learn trading strategies. In addition, students will practice the knowledge gained in economics lessons, which will only make understanding of the subject better.

The second group of stakeholders that might consider using my simulation is economics teachers. In particular, the head of the economics department at my school, Mr. Moore. 'Even though trading strategies are not included in the school curriculum, it is a popular topic in extension classes and a good simulation is essential to explain the topic' – Mr. Moore says. There is no better way to explain trading than by showing it in practice. However, the most of online trading simulations are too complicated to get started with. My simulation can be used to explain some of the most popular trading strategies in an engaging and practical way. Moreover, due to the fact that my simulation will be speeded up, patterns and trends can be found more quickly, which will make the learning process even more efficient.

The last group of stakeholders is people who want to enter the stock market and invest their savings but are too afraid to lose money due to the lack of practice and experience. My simulation will be helpful because getting some practice with a recently learnt trading strategy is a good idea before applying it in the real stock market. Also, with the use of virtual wallet my simulation will not involve any risk of losing money, making learning not stressful at all. Finally, due to carefully selected features aimed at beginners, the simulation will be much easier to use than the most of modern online trading simulations.

Throughout the project, I will keep contact with my friend Alex, who is an economics student, as well as with Mr Moore, the head of the Economics department at my school. They will be able to point out disadvantages and suggest improvements to my program.

Existing systems

Student Stock Trader

Student Stock Trader is a trading simulation, where the user gets $50,000.00 at the start and can buy and sell many different stocks, which prices and graphs are displayed. Even though the system shows price fluctuations on the graph for each stock, the usability is not great, because the graph is in an image format, so the user has no way of interacting with it. Furthermore, the graph of price fluctuations is shown from the past, so there is no way to know what changes to the price have occurred recently. The price fluctuations graph that Student Stock Trader provides is illustrated in Figure 1.

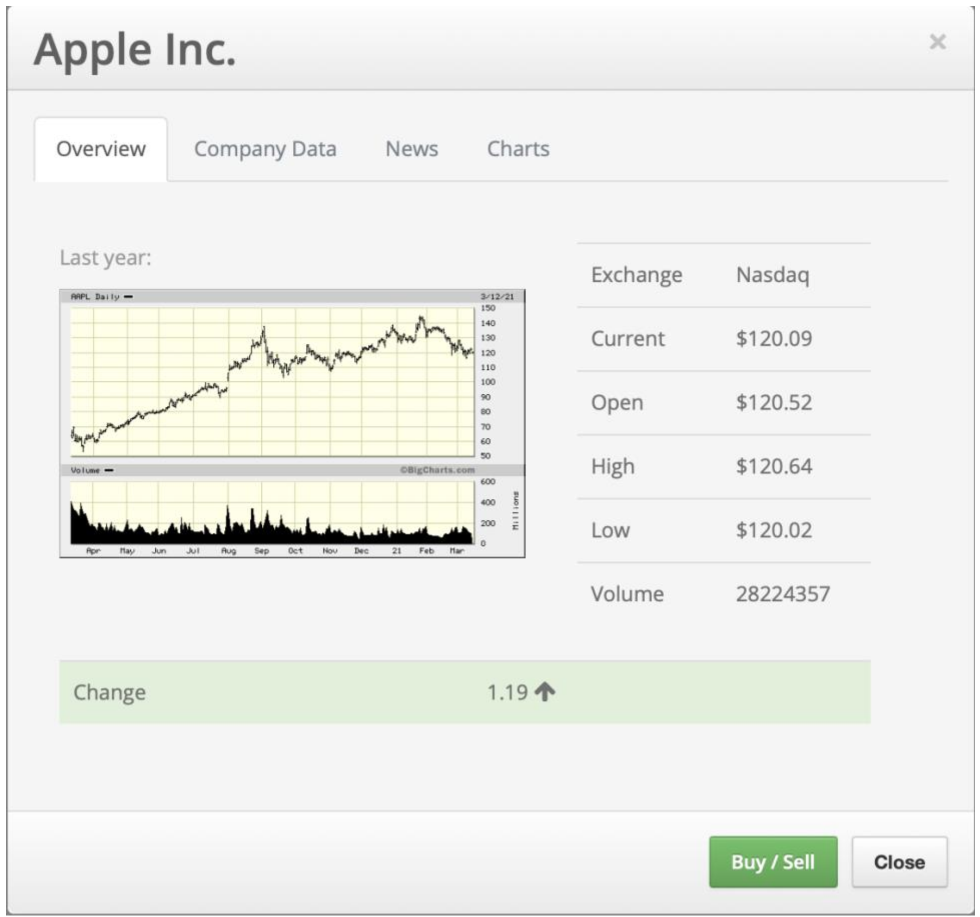


Figure 1

In my simulation, I want to allow the user to interact with the graph. The main interactions I want to implement will include:

- Using different colours for bullish and bearish candlesticks.
- Ability to hover over a candle with a mouse to see full information about it.
- Highlighting a selection of candlestick patterns.

The given simulation has many valuable features, such as buying and selling stocks using virtual money. Therefore, there is no risk of losing real money, and it is an excellent chance to practice trading skills before entering the real stock market. The use of virtual money is one of the features I want to include in my simulation. The user's wallet in my simulation is likely to look similar to the one that Student Stock Trader has created, illustrated in Figure 2.

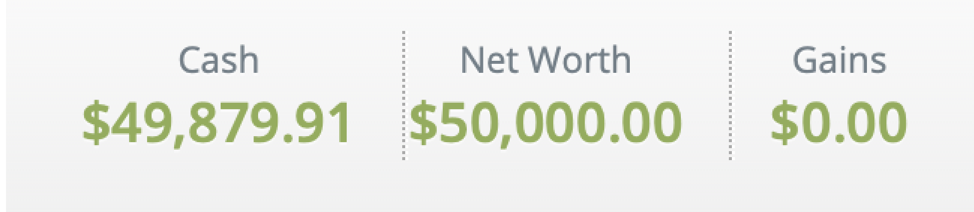


Figure 2

Also, as the stocks' prices are displayed in real-time, the simulation becomes tedious, as the user has to wait for a long time period until some action happens. The average annual return in the stock market is considered to be 9.2 - 13.6% (Knueven, 2020). Waiting time in the simulation could be used to trade real stocks and make a profit, so speeding up the simulation might be a great idea to save time.

Moreover, the learning experience becomes not as efficient as it could be. Waiting for months to realise that the chosen strategy is incorrect wastes valuable time. Therefore, I want to speed up my simulation to make it more dynamic, learning more efficient, and save valuable time. With this feature, I would not be able to use real-time stocks prices, but this is not essential for my simulation. Instead, I will use the actual historical data of stocks prices from around 2017-2020.

Trading View

Trading View is a professional trading analytics platform where the user can get a full analysis of any stock, track candlestick patterns, use their custom Python script to analyse extra data features, etc. The given simulation has a lot of features that I will not be able to implement in the given time, however it gives a lot of ideas what to include in my simulation.

The given simulation has an exceptional candlestick chart-plotting system, which will be essential for my simulation. An example of such a chart is illustrated in Figure 3. The chart has time labels on the horizontal axis and price labels on the vertical axis. A helpful feature is that the graph displays different colours for bearish and bullish candlesticks. Also, it allows the user to change the time frame, the scale of the chart and highlight any candles they find interesting. Besides, the simulation can highlight candlestick patterns, and many more.

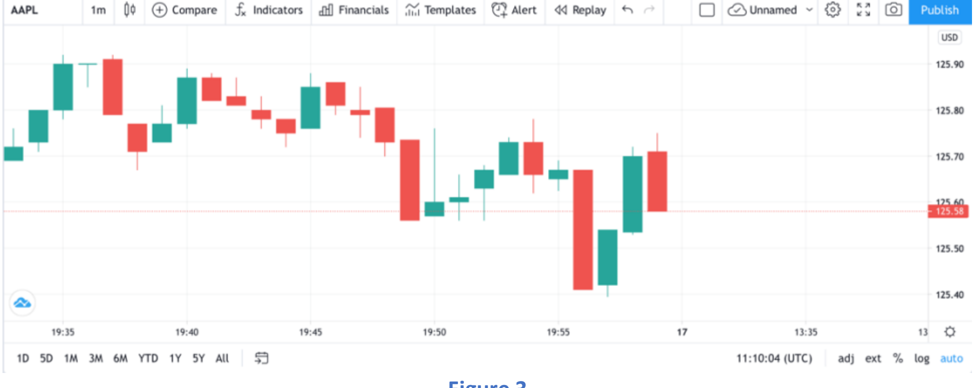


Figure 3

Moreover, the given system has such a complex feature as adding a custom Python script to analyse and predict future trends. I will not include this feature in my simulation. Firstly, beginner traders, the target audience of my simulation, generally do not know how to write custom Python scripts to analyse prices of stocks. Secondly, this is a very complex functionality that I most likely will not be able to implement.

Furthermore, Trading View allows displaying the stock's price in many different chart formats, illustrated in Figure 4. My simulation will include only Candles, as it is the most popular choice traders use and is generally the easiest to start with. Also, other types of charts do not display more information; it is just a different way of visualising the same information. Therefore, it is not worth spending traders' time learning how these charts work and developer's time implementing them.

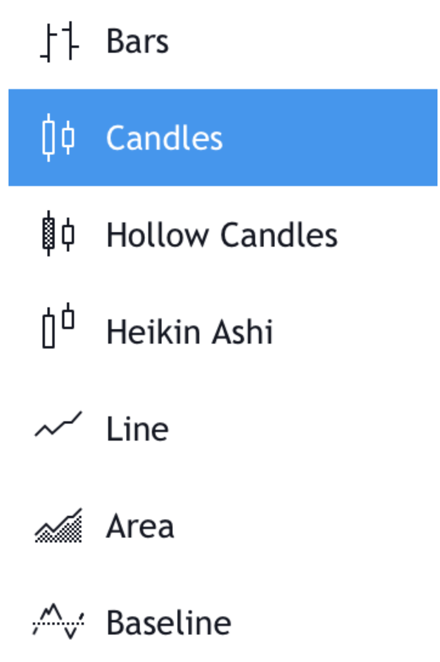


Figure 4

Capital

Capital is a professional trading platform, where the user can use both real and demo accounts to trade stocks. However, as the system is aimed at advanced traders, it is not very accessible for beginners. It has way too many advanced features and information that are not essential for a beginner trader.

A valuable feature in the given simulation is the ability to add £10,000 to the demo wallet from the account menu dropdown, illustrated in Figure 5. It is helpful because the user does not have to start the simulation again if they go bankrupt. However, the simulation should not allow the user to add an infinite sum of money to their account since it will be impossible to lose and learn on mistakes. Therefore, I am considering adding a one-time ability to top up the wallet by £10,000 as a 'second chance'.

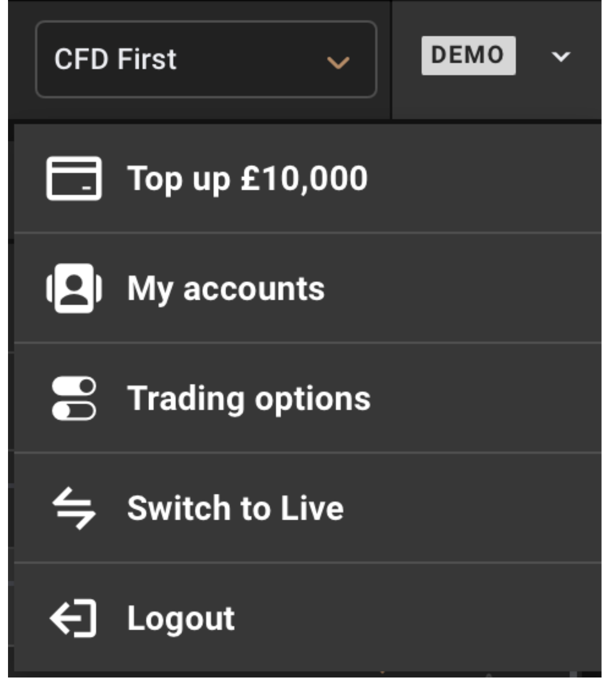


Figure 5

Underpinning knowledge and calculations

Candlestick charts

A candlestick chart is a financial chart style used to describe stock price movements and must be well understood to trade stocks.

A candlestick is a part of a chart that displays the high, low, open, and closing prices of a stock for a specific period, eg one day. Two types of candlesticks are shown in Figure 6 (Forex Trading 200, 2018).

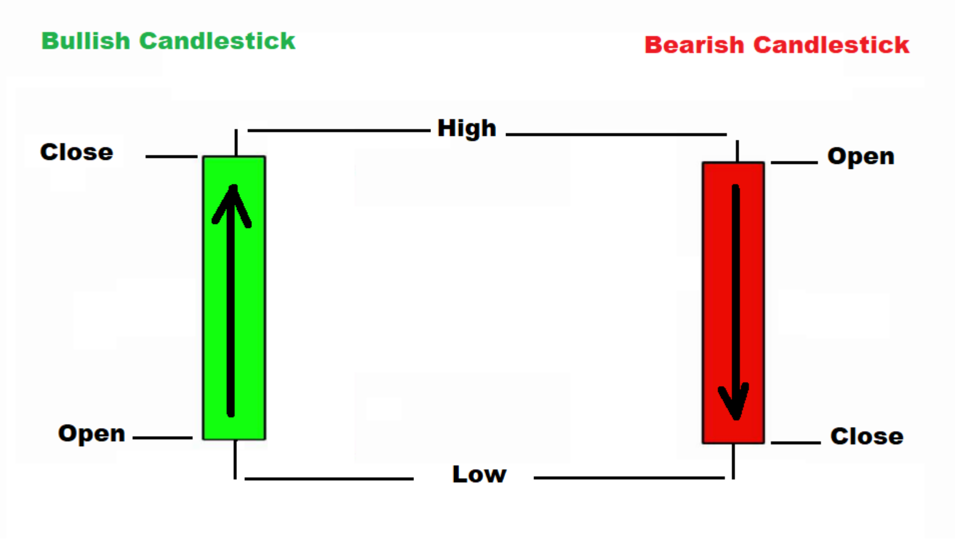


Figure 6

The wide part of the candlestick is called the 'real body' and tells traders whether the closing price was higher or lower than the opening price (red if the stock closed lower, green if the stock closed higher). The candlestick's shadows, which are the thin lines above and below the real body, show the day's high and low and how they compare to the open and close. A candlestick's shape varies based on the relationship between the day's high, low, opening and closing prices (Hayes, 2020).

The horizontal axis of a candlestick chart represents time, and the vertical axis represents a price. A candlestick chart consists of many candlesticks and shows the size of price movements over time. An example of a candlestick chart is shown in Figure 7 (Matange, 2014).

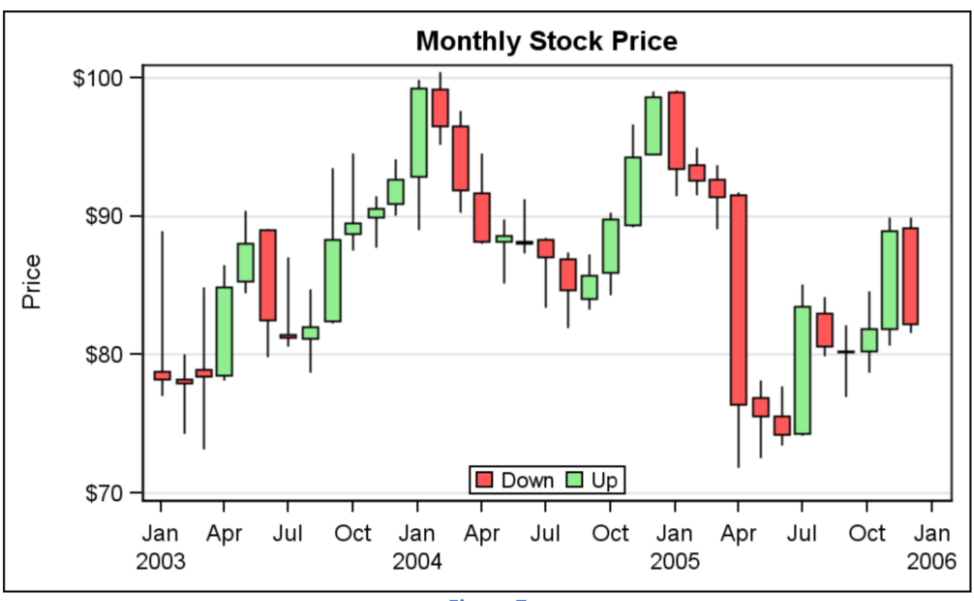


Figure 7

Candlestick patterns

One or more candles can form recognisable patterns that can tell what happened to demand, supply and predict the future direction of the stock's price movement. I will provide several popular candlestick patterns with their descriptions below. I have not decided which candlestick patterns will be included in my simulation yet.

The hammer candlestick, illustrated in Figure 8, consists of a short body with a much longer lower shadow. As a rule, you will find it at the bottom of a downtrend. The pattern indicates that the price was pushed back up. While there may be hammer patterns with both green and red candles, the former pointing to a stronger uptrend than red hammers (Bybit Learn, 2020).

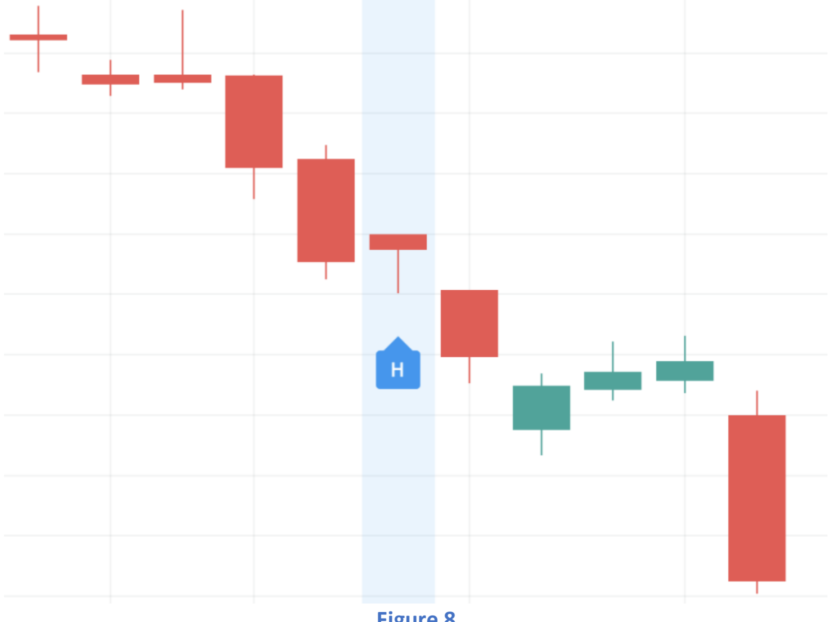


Figure 8

The inverted hammer, illustrated in Figure 9, is quite similar to the previously described pattern. It is different from the standard hammer in that it has a much longer upper shadow while the lower shadow is very short. The pattern suggests the price will be pushed higher (Bybit Learn, 2020).

Figure 9

Unlike the previous two patterns, bullish engulfing, illustrated in Figure 10, comprises two candlesticks. The first candle should be a short red body engulfed by a green candle, which is more extensive. While the second candle opens lower than the previous red one, the buying pressure increases, leading to a reversal of the downtrend (Bybit Learn, 2020).

Figure 10

Another two-candlestick pattern is the piercing line, illustrated in Figure 11, which may show up at the bottom of a downtrend. The pattern consists of a long red candle that is followed by a long green candle. This pattern's critical aspect is that there is a significant gap between the red candle's closing price and the green candle's open price. The fact that the green candle opens much higher points to buying pressure (Bybit Learn, 2020).

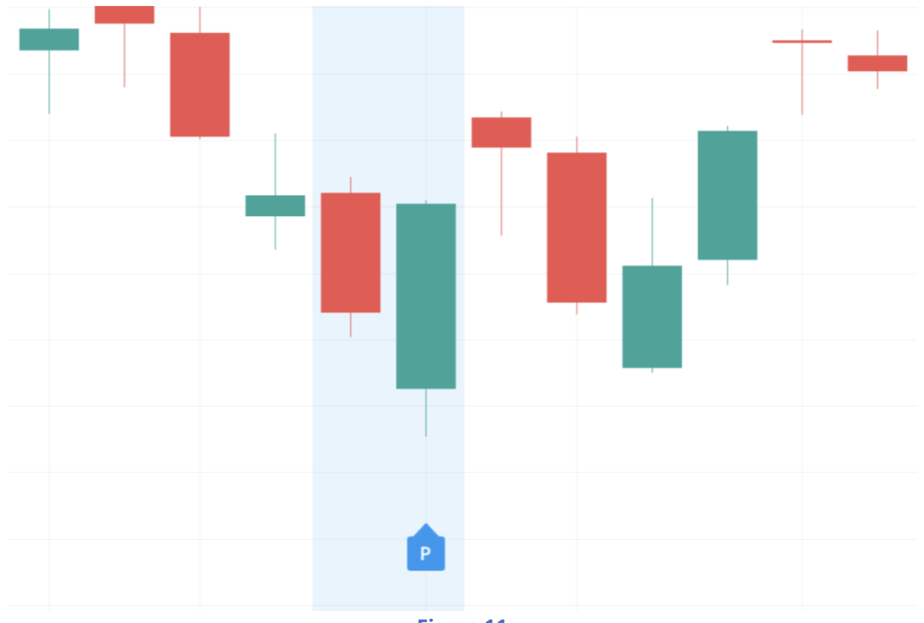


Figure 11

Moving averages

The moving average is a technical indicator used by traders to spot emerging and common trends in markets. It is a mathematical equation for finding averages, trends and smoothes out price action by filtering out 'noise' from random fluctuations (Mahony, 2019). There are two main types of moving averages: weighted moving average and simple moving average.

The weighted moving average is calculated by placing greater weight on the most recent data points. Because of this, weighted moving averages react significantly to the most recent price changes. (Mahony, 2019)

The simple moving average is calculated by taking the average closing price of the candles over any period desired. To find simple moving average for a specific period, the total closing price is divided by the number of periods (Mahony, 2019). Example of a simple moving average is shown in Figure 12 by a thin blue line.

Figure 12

Crossover

The crossover is a point on the trading chart in which two indicators cross. Crossovers are used to estimate the performance of a financial instrument and to predict coming changes in trend (Chen, 2020).

The golden cross is a chart pattern that is a bullish signal in which a relatively short-term moving average crosses above a long-term moving average. An example of such a crossover is illustrated in Figure 13.

The death cross is a chart pattern that is a bearish signal in which a relatively short-term moving average crosses below a long-term moving average. An example of such a crossover is illustrated in Figure 14.

In both figures, the green line is a 15-day, and the blue line is a 50-day simple moving average.

Figure 13

Figure 14

Pattern recognition

Pattern recognition allows traders to find candlestick patterns and predict the likely future trend of the stock. Candlestick patterns can be recognised by humans or computers. As my simulation is designed for beginner traders, they will not be able to recognise any candlestick patterns. Therefore, searching for candlestick patterns with a computer and highlighting patterns for the user will allow them to easily learn the most popular candlestick patterns and apply them later in the real stock market.

Stakeholder and user needs

Having had a discussion with Alex and Mr Moore, I pointed out the key user needs for my simulation, which are listed as a summary below.

What are the main problems my program is trying to solve?

- Allow the users to practice trading strategies without the risk of losing money
- Make the program easy to use for beginner traders
- Make learning process as efficient as possible by speeding up the simulation

What are the stages of the process?

- Select a level
- Choose moving average type and set its parameters
- Look for intersections and identify the type of crossover
- View and learn candlestick patterns
- Make decisions about buying and selling the stock based on crossovers and patterns
- View results and make conclusions

What happens at each stage?

- There will be three levels to choose from with requirements and descriptions
- The user will be able to choose moving average type from a selection and change its parameters at any time throughout the simulation
- Parameters will have a default value
- Candlestick patterns will be highlighted on a candlestick chart and annotated with the name of the pattern and a suggestion whether to buy or sell the stock
- The program will highlight crossovers of moving averages and annotate a suggestion whether to buy or sell the stock
- The user will be able to enter how many stocks they want to buy or sell
- The profit will be calculated constantly and when the required profit is achieved, the level is passed

What input and output screens will be required?

- There will be a couple of input fields for the moving average parameters and the number of stocks to buy or sell
- An output screen to display a candlestick chart, a moving average chart, a virtual wallet and the requirements of the chosen level

Are there any specific requirements for the graphical interface?

- A candlestick chart where the user can interact with it by hovering over with their mouse
- A moving average chart where the user can easily spot intersections of moving averages

How will the user interact with the product?

- A keyboard will be used to enter required parameters, number of stocks, etc.
- A mouse will be used to interact with the simulation and charts
- A monitor will be used to display the charts, virtual wallet and other information

What data needs to be input into the system and how will this happen?

- Parameters such as what type of moving average the user wants to use
- Number of stocks to buy or sell

- Stock data which will be stored in a separate file and retrieved when needed

What data needs to be output from the system and in what format?

- A candlestick chart with highlighted and annotated candlestick patterns
- A moving average chart with highlighted and annotated intersections
- A virtual wallet, level requirements and the results at the end of the simulation

Identifying and explaining any limitations

As discussed in the existing systems, professional trading platforms have a lot of advanced features that are not essential for beginners. One of such features is allowing users to write custom Python script to analyse fluctuations in the price of a stock and predict the future trend. This requires the user to know Python very well and the developer to know how to implement the feature. As the target audience of my simulation is beginners, it is not likely that they know how to analyse stocks with Python. I will not be able to implement such a complex feature, because I do not have the required skills and it is difficult if not impossible to do in the available time. Therefore, allowing the user to write custom Python script is one of the limitations of my simulation.

Secondly, Trading View allows its users to use many different charts to display information about stocks. They include Line, Area, Baseline, Bars, Candles, Hollow candles and Heikin Ashi. Using a different type of chart does not benefit the user, it is just a different way of visualising the same data. Furthermore, beginner traders do not have to know all types of charts to start trading. Moreover, I will not be able to add several types of charts in the available time, so displaying several types of charts for the user is another limitation of my simulation.

Finally, Trading View and Capital can search for hundreds of complex candlestick patterns and allow the user to add hundreds of indicators, similar to the Simple Moving Average. Even though it benefits a professional trader, who makes a living by trading stocks, many beginners will be confused. In addition, I will not be able to implement hundreds of complex algorithms in the time available. Therefore, my simulation will only be able to search for some of the main candlestick patterns and allow the user to add the main and the most important indicators. Number of technical indicators and candlestick patterns is another limitation of my simulation.

Computational methods

Firstly, in my simulation, a lot of algorithms have to be followed and calculations to be done in order to correctly predict the future trend of a stock. For instance, calculating Simple Moving Average, finding crossovers and plotting a chart takes a long time to do for a human being, but a computer can do it almost instantaneously. Computers become especially helpful when the number of calculations is huge, which is exactly what will happen in my simulation. The chance of human making a mistake is large, which might lead to incorrect outcome with a loss of money. Therefore, using computational approach is crucial in trading, as it saves a lot of time and eliminates human calculation errors.

Secondly, my program will use pattern recognition to predict the future trend of stocks. This involves comparing stocks data to the known candlestick patterns and deciding if they are similar. While a human can instantaneously find some obvious patterns, it might be difficult to find patterns that are well hidden in the data. However, a computer is able to make millions of comparisons each second and is not likely to miss a pattern if it exists in the data. Therefore, the use of a computer to find candlestick patterns in my simulation is essential to eliminate even more human mistakes.

Thirdly, my simulation will output a candlestick chart so that the user could visualise information in a graphical way, as it is more intuitive than raw numbers. Plotting a chart for every stock by hand takes a lot of calculations and time and does not prevent any errors. As computers can perform calculations without any mistakes and much quicker than humans, plotting charts is another task that computers do much better than humans. Hence, computational power of a computer should be used in my simulation to plot candlestick and moving average charts quickly and correctly.

Specifying software and hardware requirements

Criteria	Requirement	Justification
Programming language	JavaScript	I want my simulation to run on as many devices as possible, so it will be browser-based. My simulation will be programmed in JavaScript, because every browser can execute it.
Integrated Development Environment	JavaScript, HTML, CSS support	As the logic of the simulation will be written in JavaScript, the content in HTML and the styles in CSS, the IDE must support all of them. Visual Studio Code is the IDE that I will use.
Libraries	Plotly	Plotly is a chart-plotting JavaScript library that will help me to display candlestick charts as part of my simulation. This library is needed because I will not be able to implement a chart-plotting system myself, but the feature is necessary for the simulation.
Operating system	Browser support	As my simulation is browser-based, an operating system must have a browser support. Windows, MacOS and Linux can be used to run the simulation, but I will be using MacOS.
Browser	Send HTTP requests and execute JavaScript	As my program is written in JavaScript, a browser must be able to execute it. All data will be stored on a server, so a browser must be able to send HTTP requests to retrieve necessary pieces of information. I will be using Chrome, because it has a modern JavaScript engine.
Monitor	At least 720px wide	Way of viewing the simulation. There will be charts, so a wider monitor will benefit the user.
Keyboard	Any	Way of entering required parameters.
Mouse	Any	Way of interacting with the simulation.

User requirements and measurable success criteria

№	Success criteria	Justification
1	The user can see the starting screen when they enter the address of the website into a browser	The user needs to know what website they just entered. The starting screen should contain the name of the simulation and a button to proceed to selecting a level.
2	The user can see the screen with a selection of levels after pressing a button on the starting screen	The user should be able to choose a level to complete. In the selection there has to be some basic information about each level.
3	The stock data is successfully sent from a server and received by the user	Stock data will be stored on a server, so it has to be retrieved after the user chooses a level.
4	The user should be able to see the virtual wallet and requirements of the level they chose	Vurtual wallet is needed to keep track of the budget, take action in case budget is running extremely low and see if the chosen trading strategy works. Level requirement will act as an aim. It will show what profit a professional trader is likely to make.
5	The user can see a candlestick chart for the chosen stock	A candlestick chart is required to spot candlestick patterns, as well as fluctuations in the price of the stock.

6	The user can hover over a candle to see information about a specific candle at a particular time	Professional traders always go back and view historical fluctuations of stocks. It might tell the user if the stock will go up or down in price. Hovering over with a mouse allows to see full information about a specific candle at a particular time.
7	The user can see two line graphs: short-term and long-term	Moving averages is one of the indicators to predict the future trend of stocks. Therefore, the user should have an ability to view line graphs of the short and long-term moving averages.
8	The user can see spots where moving averages intersect with a suggestion whether to buy or sell the stock	Crossovers are needed to show the user the best spots to buy or sell stocks. These spots should be highlighted and annotated for the best usability.
9	The program highlights and annotates candlestick patterns in the data of a particular stock	Pattern recognition is another way of predicting the future trend of stocks, so teaching the users to find them is one of the problems my simulation is trying to solve. Highlighting and annotating patterns will ensure the best user experience.
10	The user can change parameters and moving average type at any time	The user will not necessarily choose the right strategy from the first time, so they should be able to change it any time they want.
11	The user can buy and sell the chosen stocks if they have enough funds or stocks	The user should be able to trade the chosen stock and actually make profit or lose money. This is the best way to learn trading strategies, which is the aim of my simulation.
12	The user can stop the simulation at any time and view results	The user will not necessarily wait until the level is finished and might decide to leave the simulation sooner. Therefore, there should be a button to stop the simulation at any time.
13	The user can see how much they earned or lost during the simulation	Statistics is needed to show how well the user performed in the chosen level and to make conclusions about the trading strategy used.

Design

Breaking the problem down systematically

The use of top-down design helps me identify the structure of my program and break it down systematically into smaller parts. By doing this, I will be able to structure my code more efficiently and develop the solution quicker. Also, I will identify which inputs and outputs each part of my program needs, and therefore, suitable validation checks. Moreover, it will be easier for me to identify data structures and algorithms needed in each part of my simulation.

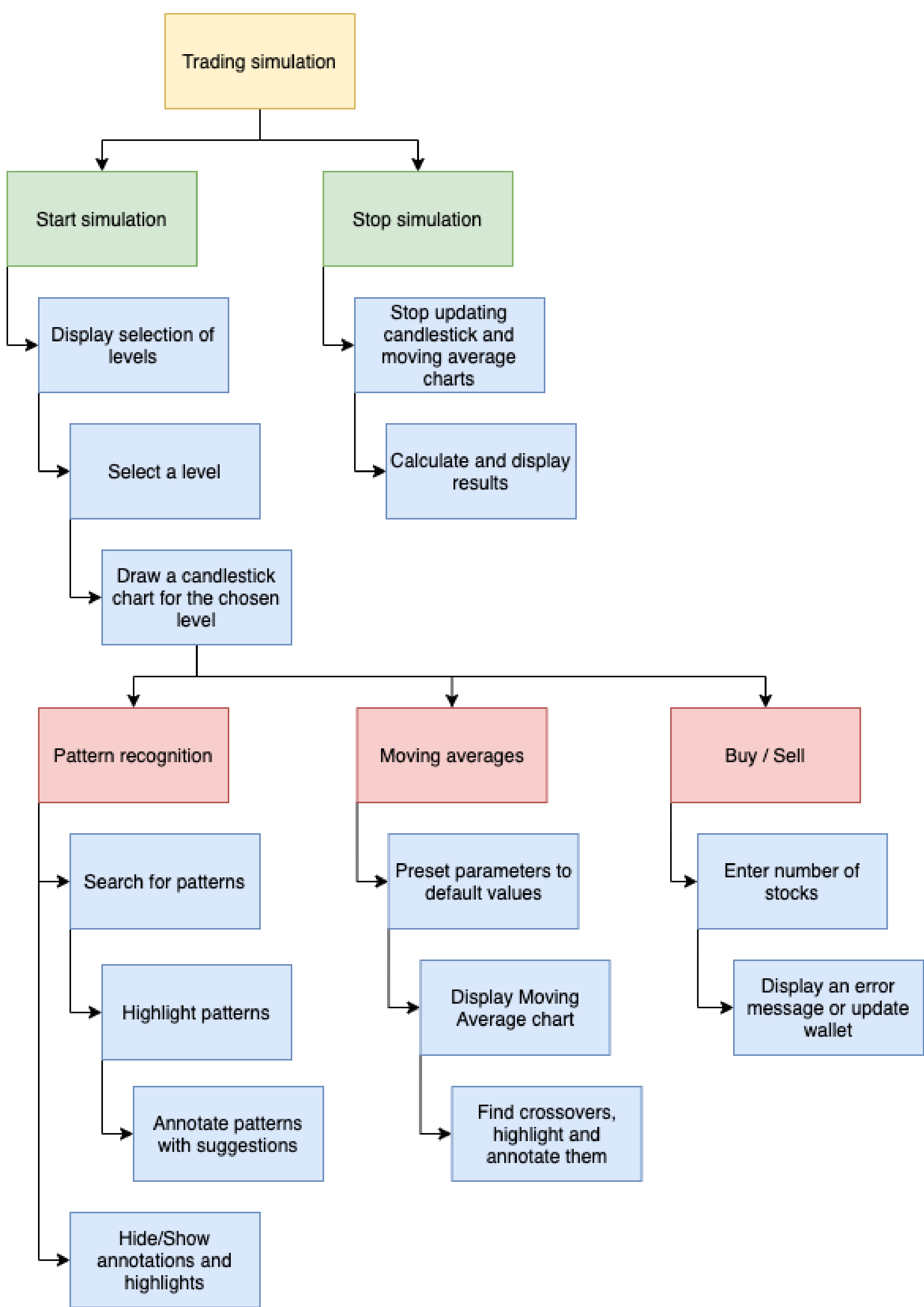


Figure 15

Explanation of each module

Start simulation

- **Display selection of levels.** This function is needed to allow the user see which levels are available in the simulation. Levels will show requirements, initial capital and will have different colour depending on the difficulty of the level. It will display information about each level in a separate box.
- **Select a level.** This function is needed to allow the user select a level from the selection and start practicing trading algorithms and pattern recognition. It will allow the user to click a level they want and they will be redirected to the main screen of the simulation, where they would be able to trade the chosen stock.
- **Draw Candlestick Chart for the chosen level.** This function will be responsible for drawing a candlestick chart for the chosen stock and updating it regularly. In order to update it, I will create an interval and set the time period to be around 1-2 seconds.

Pattern recognition

- **Search for patterns.** The program will analyse the stock data and check if there are any patterns. It will take two candles at one time and check if the two candles form a pattern.
- **Highlight patterns.** If the program has found any candlestick patterns in the data, they will be highlighted on the candlestick chart. This function will highlight the pattern depending on the dates of the candles which form the pattern.
- **Annotate patterns with suggestions.** The program will display the name of the found patterns and suggestion whether to buy or sell the stock depending on the pattern.
- **Hide/Show annotations and highlights.** The user will have a choice whether to hide or show highlights and annotations of candlestick patterns.

Moving averages

- **Preset parameters to default values.** There will be default values in parameters of moving averages at the beginning of the simulation. The user will have a choice to change or keep them.
- **Display Moving Average chart.** After the simulation received all required inputs from the user or the default values, a chart can be drawn and displayed using a chart-plotting library. The chart will be constantly updated every 1-2 seconds to add new data points to it.
- **Find crossovers, highlight and annotate them.** After the chart has been displayed, the program will find where the two moving averages cross and identify whether it is a golden or death cross to predict the future price direction of the stock. Also, the program will highlight the intersection points and suggest whether to buy or sell the stock by using annotations on the moving average chart.

Buy / Sell

- **Enter number of stocks.** The user will be able to buy and sell stocks regardless of what the simulation suggested about the future trend of the price. However, in order to buy and sell stocks, the user will need to enter the number of stocks that they want to buy.
- **Display an error message or update wallet.** If the number of stocks entered does not pass the validation check, an error message will be displayed. Otherwise, the transaction will happen and the wallet will be updated.

Stop simulation

- **Stop updating candlestick and moving average charts.** This function will stop the interval created to update both charts so that they stop updating.
- **Calculate and display results.** This function will be responsible for calculating and displaying overall profit after the user presses the stop button. Also, it will output the results to the screen so that the user could make conclusions about the strategy used.

Interfaces

Firstly, my simulation will have a starting screen, which will have a text 'Trading simulation' in the centre of the screen and a button 'Start trading'. The prototype of the starting page is shown in Figure 16.

Figure 16

Secondly, my simulation will have a screen, where the user will have to choose level they want to play. There will be 3 levels with information about each one. Levels with easy difficulty will have a green colour, while levels with a medium difficulty will have an orange colour. The prototype of the screen with levels is shown in Figure 17.

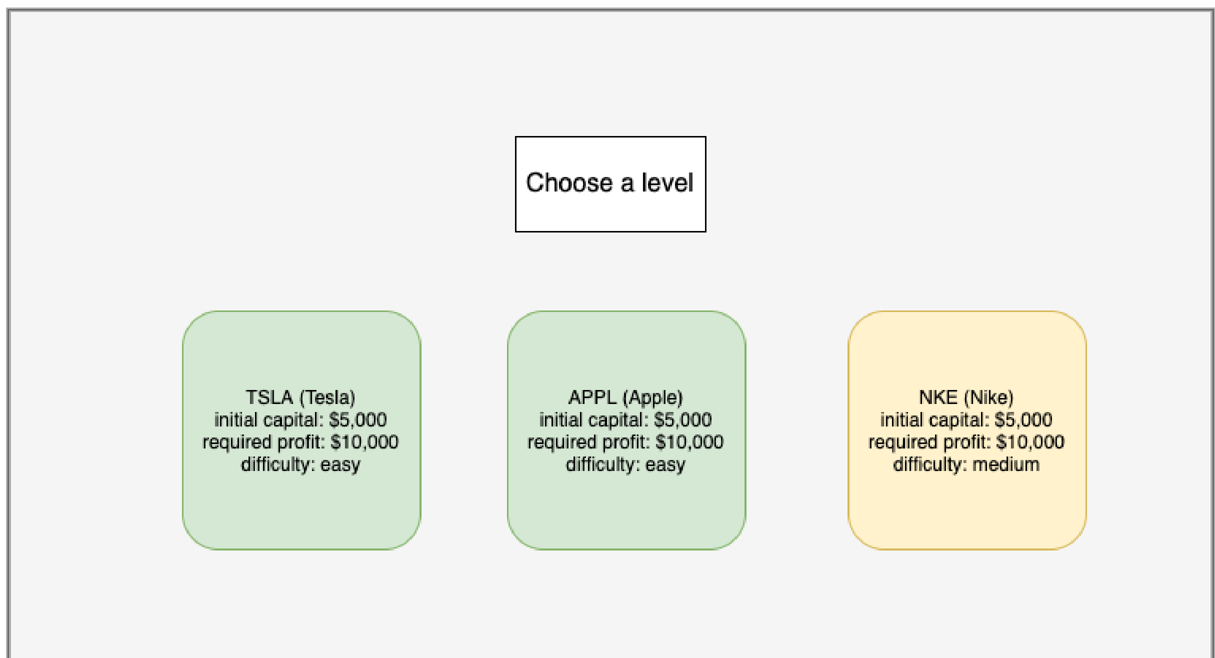


Figure 17

The main screen that my simulation will have is shown in Figure 18. It has the stock name displayed at the top of the screen, because the user needs to be clear which stock they are trading. Also, various charts will be shown on the left side of the screen with the minimum width of 500px to ensure the user can easily see all

features of the charts. On the right hand side there will be virtual wallet, information about the chosen level, input fields to buy/sell stocks, input fields for moving averages parameters and checkbox to show/hide pattern recognition. All of the above information should fit on one screen to improve usability so that the user did not have to scroll to find whatever information they need.

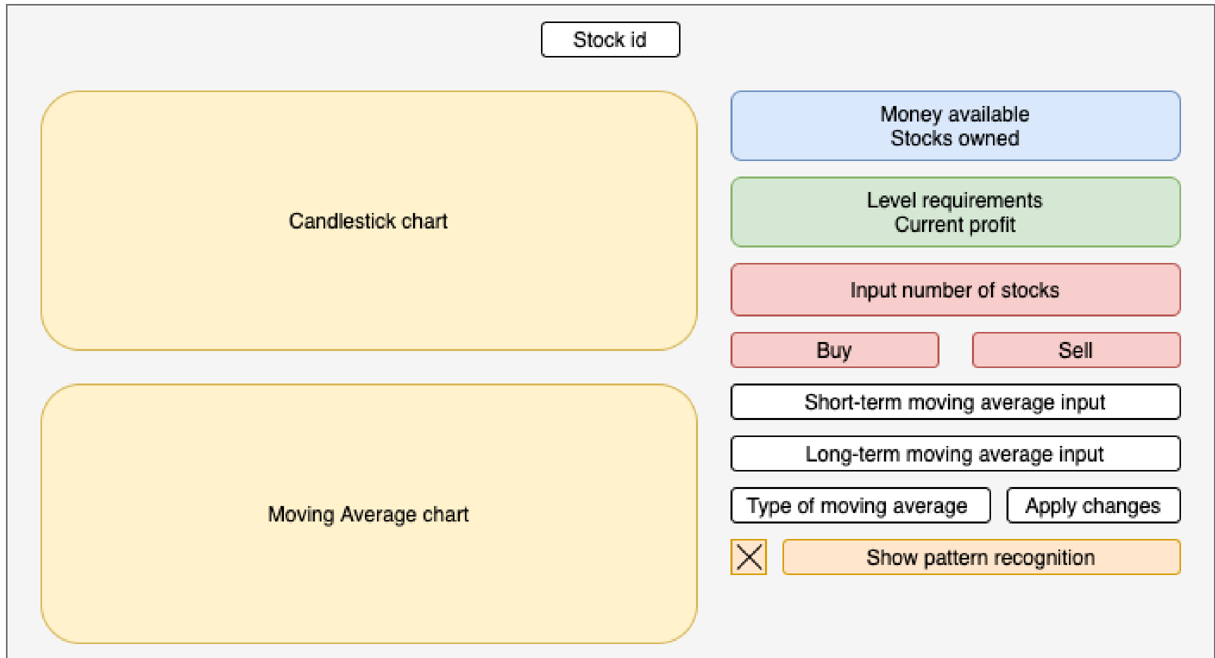


Figure 18

Validation of inputs

In my simulation, validation will be needed to prevent users from spending money they do not own, resulting in a negative virtual wallet balance. The program will calculate the price user has to pay for the entered number of stocks. If the price is higher than the available money, the transaction will not happen and the user will see a message saying 'not enough funds'. If the price is less than or equal to the available money, the transaction will happen, available money will decrease by the price and owned stocks will increase by the number of stocks the user entered. If this validation check did not exist, the user would be able to buy an infinite number of stocks despite insufficient funds.

Also, the program needs to eliminate the case, when the user wants to sell more stocks than they own. The entered number of stocks will be compared to the number of owned stocks. If the number of stocks the user owns is greater than or equal to the number of stocks they want to sell, the transaction will happen, the owned stocks will decrease by the number of stocks entered and the available money will increase by the number of stocks multiplied by the currect stock price. However, if the number of owned stocks is less than the number of stocks entered, then the user will see an error message saying 'not enough stocks'. If this validation check did not exist, the user would be able to sell millions of stocks that they do not own and become a billionaire.

Moreover, the moving average parameter must be an integer, because it shows how many data points are taken into account when calculating an average. It has to be positive, as it is impossible to have a negative number of data points. Furthermore, traders usually set the short-term moving average between 5 and 14 and the long-term moving average between 15 and 50. I chose the same ranges, so that my simulation was as professional and close to the real life as possible. There will be some presets: short-term moving average will be set to 5 and the long-term one will be set to 15 at the beginning of the simulation. The user will be able to change those values during the simulation if needed. Also, the type of the moving average will be set to Simple Moving Average, but the user will be able to choose Weighted Moving Average during the simulation if needed. The choice of moving average type does not require validation, as the user will only be able to set the moving average type by clicking one of the options from the dropdown box, that I will create.

Input	Validation check
Short-term moving average	Integer in range 5-14 days
Long-term moving average	Integer in range 15-50 days
Number of stocks	• Integer greater than zero • Number of stocks to buy multiplied by the current stock price must be less than or equal to the available money • Number of stocks to sell must be less than or equal to the owned stocks

External files

As discussed earlier, the stock data for each level will be stored in separate JSON files in a different directory called 'json'. Names of JSON files will be in format 'stockID.json'. Each JSON file will contain an array of objects and those objects will have the following attributes:

Attribute	Data type	Purpose and justification
open	number	This attribute represents the open price of the stock.
high	number	This attribute represents the highest price of the stock.
low	number	This attribute represents the lowest price of the stock.
close	number	This attribute represents the close price of the stock.
date	string	This attribute represents the date when the stock had the above prices. The format will be yyyy-mm-dd.

All of those attributes are needed to plot a candlestick chart, perform calculations to find candlestick patterns, calculating Simple and Weighted Moving Averages. Stock data in these files is essential for my simulation to work.

Data structures

stock

This object will store all information related to the chosen stock and it will contain the following attributes:

Attribute	Data type	Purpose and justification
id	string	This attribute will hold the id of the chosen stock. It will be needed to create a valid url to the stock data, as well as to display the id to the user so they know which stock they are trading.
url	string	This attribute will hold the url to the stock data in a different directory. It will be needed to retrieve correct stock data for the chosen stock.
data	array	This attribute will hold stock data for the chosen stock. It will be needed to plot candlestick and moving average charts, as well as to trade the chosen stock.
price	number	This attribute will hold the price of the chosen stock. It will be needed to calculate the current profit, as well as to trade the chosen stock.
count	number	This attribute will hold the number of days passed since the start of the simulation. It is needed to access correct data inside various arrays.

wallet

This object will store all information related to the virtual wallet and it will contain the following attributes:

Attribute	Data type	Purpose and justificaiton
money_available	number	This attribute will hold the available money that the user can use to buy more stocks. It is needed to not let the user buy more stocks than they can afford.
owned_stocks	number	This attribute will hold the number of stocks the user owns. It is needed to calculate the current profit, as well as to show the user how many stocks they can sell.
profit	number	This attribute will hold the current profit. It is needed to determine if the user has passed or failed the chosen level and to show the user how efficient their trading strategy is.

levels

This array will contain three objects, which are three different levels for my simulation. Each of those objects will contain the following attributes:

Attribute	Data type	Purpose and justificaiton
name	string	This attribute will hold the name of the chosen stock. It is needed to let the user know which stock they will trade if they select a particular level.
id	string	This attribute will hold the id of the chosen stock. It is needed to fetch stock data from a correct JSON file.
initial_capital	number	This attribute will hold the initial capital for the level. It is needed to let the user know how much money they will have at the beginning of the level.
required_profit	number	This attribute will hold the required profit for the level. It is needed to let the user know how much profit they need to make in order to pass the level.
difficulty	string	This attribute will hold the difficulty of the level. It is needed to correctly set the background colour for the level and to let the user choose the difficulty they want.

level

This object will store all information related to the chosen level and it will contain the following attributes:

Attribute	Data type	Purpose and justification
initial_capital	number	This attribute will hold the initial capital for the chosen level. It is needed to set the value of the 'money_available' variable at the beginning of the level.
required_profit	number	This attribute will hold the required profit for the chosen level. It is needed to display the required profit to the user so that they know how much profit they need to make in order to pass the level.

candlestick

This object will store all information related to the candlestick chart and will contain the following attributes:

Attribute	Data type	Purpose and justification

container	string	This attribute will hold the name of the HTML element where the candlestick chart will be displayed. It is needed to display the candlestick chart in a correct place to allow the user see it clearly.
data	array	This attribute will hold the data for the candlestick chart. It will be an array that contains one object. It would be impossible to plot a candlestick chart without this attribute.
layout	object	This attribute will hold the configuration settings for the candlestick chart. It is needed to modify the appearance of the candlestick chart to highlight patterns and annotate their names and suggestions whether to buy or sell a stock.

The data array will contain an object that will hold the actual data to plot the candlestick chart. The object will contain the following attributes:

Attribute	**Data type**	**Purpose and justification**
x	array	This attribute will hold dates when the stock was available to trade. It is needed to annotate the x-axis of the candlestick chart correctly.
open	array	This attribute will hold the opening prices of the stock. It is needed to correctly plot the candlestick chart.
high	array	This attribute will hold the highest prices of the stock every day it was available to trade. It is needed to correctly plot the candlestick chart.
low	array	This attribute will hold the lowest prices of the stock every day it was available to trade. It is needed to correctly plot the candlestick chart.
close	array	This attribute will hold the closing prices of the stock. It is needed to correctly plot the candlestick chart.
type	string	This attribute will hold the type of the chart. In this case, its value will be set to 'candlestick', as we are aiming to plot a candlestick chart.

The layout object will contain several attributes as well. The most important ones will be shapes and annotations, which are both arrays of objects, and they will be used to highlight and annotate candlestick patterns. Other attributes are related to the appearance of the candlestick chart and I will decide what they will be during the development stage while creating prototypes.

moving_average

This object will store all information related to the moving average chart and will contain the following attributes:

Attribute	**Data type**	**Purpose and justification**
container	string	This attribute will hold the name of the HTML element where the moving average chart will be displayed. It is needed to display the moving average chart in a correct place to allow the user see it clearly.
data	array	This attribute will hold the data for the moving average chart. It will be an array that contains two objects. It would be impossible to plot a moving average chart without this attribute.
layout	object	This attribute will hold the configuration settings for the moving average chart. It is needed to modify the appearance of the moving average chart to highlight intersections and

		annotate the type of the intersection and suggestions whether to buy or sell a stock.

The data array will contain two objects: one for short-term moving average and one for long-term moving average. Each of them will contain the following attributes:

Attribute	Data type	Purpose and justificaiton
x	array	This attribute will hold dates when the stock was available to trade. It is needed to correctly annotate the x-axis on the moving average chart.
y	array	This attribute will hold the y-values for both line graphs. Y-values will depend on the type of the moving average selected. They are needed to correctly plot the moving average chart.
type	string	This attribute will hold the type of the graph we want to display, which is ‘scatter’ graph in this case.
mode	string	This attribute will hold the mode of the graph we want to display, which is ‘lines’ in this case.
name	string	This attribute will hold the name for each of the moving averages. It will be one of the following: ‘Short-term SMA’, ‘Long-term SMA’, ‘Short-term WMA’, ‘Long-term WMA’.

The layout object will contain several attributes as well. The most important ones will be shapes and annotations, which are both arrays of objects, and they will be used to highlight and annotate the intersections of the moving averages. Other attributes are related to the appearance of the moving average chart and I will decide what they will be during the development stage while creating prototypes.

Pattern

This object will store all information related to candlestick patterns and contain the following attributes:

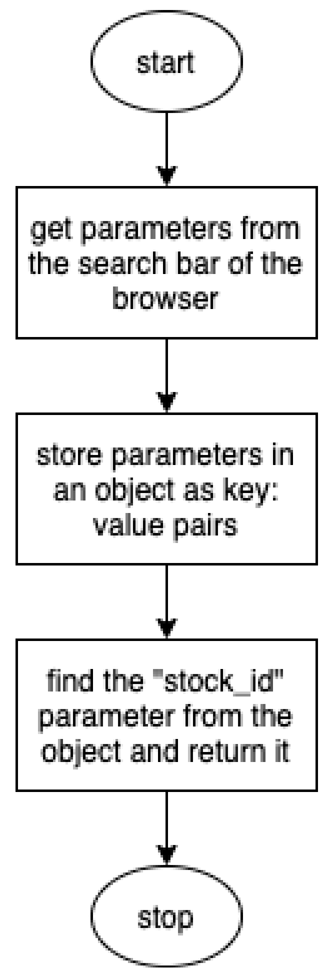
Attribute	Data type	Purpose and justificaiton
show	boolean	This attribute will hold ‘true’ if the user wants to see highlighted and annotated candlestick patterns and ‘false’ otherwise.
prev	object	This attribute will hold the candlestick data for the previous day when the stock was available to trade. It is needed to find patterns in the stock data.
curr	object	This attribute will hold the candlestick data for the current day of trading the stock. It is needed to find patterns in the stock data.
name	string	This attribute will hold the name of the last candlestick pattern found. It is needed to annotate the candlestick chart correctly.

The prev and curr objects will represent current and previous day and contain such data as close, high, low, open and date for the stock. They are needed to find candlestick patterns in the stock data.

Algorithms

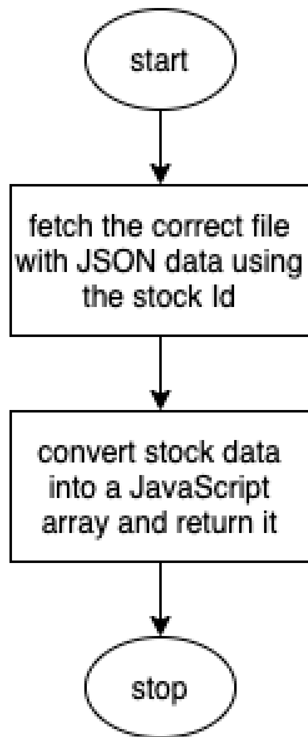
Getting id of the chosen stock from the url

The ‘choose-level.html’ page will insert the id of the chosen stock into the search bar of the browser as a parameter. This algorithm will get all parameters from the search bar of the browser, look for the parameter called ‘stock_id’ and return it. This algorithm is needed because without it the program will not know which stock the user has chosen.



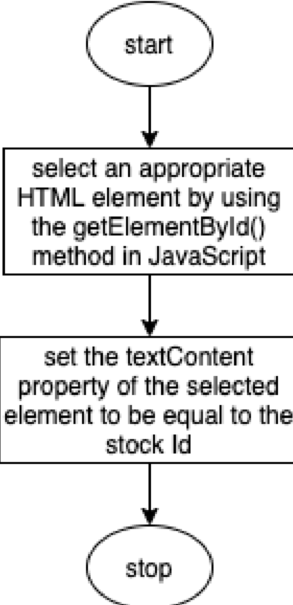
Fetching stock data

There will be three JSON files with stock data in a directory called ‘json’. The files will have names in the format ‘stockId.json’, and in order to fetch the correct file with stock data, the program will need to concatenate the path to the JSON directory, the correct stock id and the file extension. For example, ‘./json/APPL.json’ will be a valid path to the Apple stock data. This algorithm will fetch stock data from a correct file using the id of the chosen stock, convert JSON into a Javascript array and return the array with stock data. This algorithm is needed because without it the program will not have access to the stock data of the chosen stock.



Displaying id of the chosen stock

This algorithm will select the correct HTML element and set its 'textContent' property to be the stock Id. This algorithm is needed for the user to know which stock they are trading.



Getting current price of the chosen stock

As explained in the data structures section, there will be an object called 'stock' that will have several attributes. Some of the attributes will need to be constantly updated. For example, the price of the stock will change every day. In order to keep track of how many days have passed, there will be a counter. This

algorithm will get the current close price of the stock using the counter. This algorithm is needed to calculate profit that the user has made.

```
FUNCTION getStockPrice
        return stock.data[stock.count].close
ENDFUNCTION
```

Updating the counter

This algorithm will update the counter and will be called with a constant frequency by a different function in the main part of the program. This algorithm is needed to keep track of how many days have passed since the start of the simulation.

```
FUNCTION updateStockCounter
        stock.count = stock.count + 1
ENDFUNCTION
```

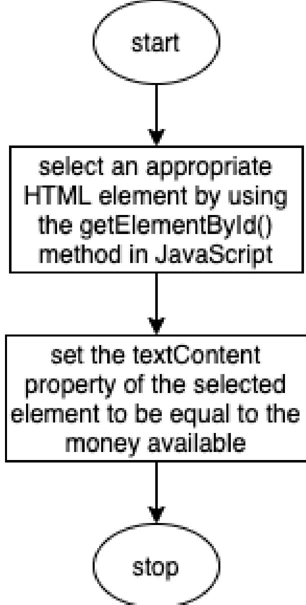
Storing stock data

This algorithm will call some of the above functions and store their returns in appropriate variables. This function will be called once at the beginning of the simulation and some of its attributes will change throughout the simulation. For example, the price of the stock will be updated every day.

```
FUNCTION getStockData
        stock.id = getStockId()
        stock.url = './json/' + stock.id + '.json'
        stock.data = fetchStockData()
        stock.price = getStockPrice()
ENDFUNCTION
```

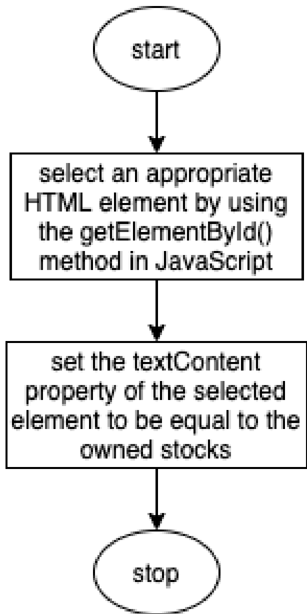
Display available money

This algorithm will select the correct HTML element and set its 'textContent' property to be the money available. This algorithm is needed for the user to know how much money they have.



Displaying owned stocks

This algorithm will select the correct HTML element and set its 'textContent' property to be the owned stocks. This algorithm is needed for the user to know how many stocks they own.



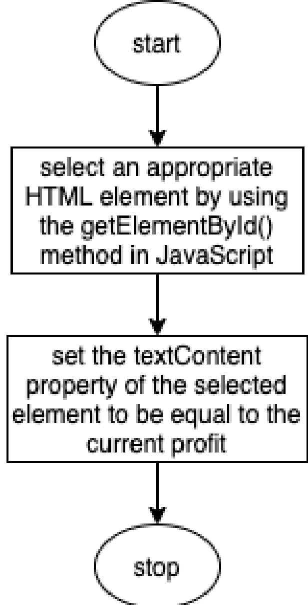
Calculating current profit

This algorithm will calculate current profit by taking into account initial capital, money available, owned stocks and the current price of the stock. This function is needed to find out if the user has met the requirements of the level.

```
FUNCTION
        return wallet.money_available + wallet.owned_stocks * stock.price – level.initial_capital
ENDFUNCTION
```

Displaying current profit

This algorithm will select the correct HTML element and set its ‘textContent’ property to be the current profit. This algorithm is needed for the user to know how much profit they made and if they met the requriements of the level.



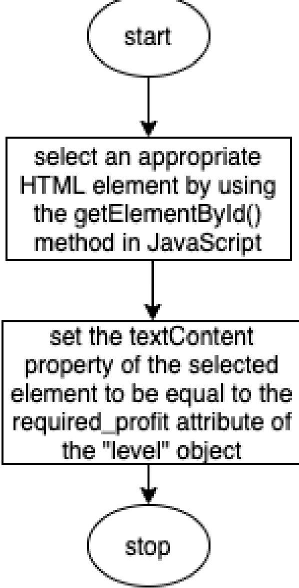
Getting required profit for the level

As discussed in the data structures section, there will be two objects related to completing the level. The ‘levels’ array will contain information about all levels. The ‘level’ object will contain information about the current level. This algorithm will set the ‘required_profit’ attribute of the ‘level’ object to the correct value depending on the id of the chosen stock by looping through the ‘levels’ array. This algorithm is needed because it is easier to save level settings once but in a different object than loop through the ‘levels’ array every time the program needs to get some information about the chosen level.

```
FUNCTION getRequiredProfit
        FOR i IN RANGE FROM 0 TO length(levels)
                IF stock.id == levels[i].id
                        level.required_profit = levels[i].required_profit
                ENDIF
        ENDFOR
ENDFUNCTION
```

Displaying required profit for the level

The user should know how much profit they need to make to pass the level. Therefore, this algorithm will select an appropriate HTML element and set its ‘textContent’ property to be equal to the ‘required_profit’ attribute of the ‘level’ object.



Getting initial capital for the level

As discussed in the data structures section, there will be two objects related to completing the level. The ‘levels’ array will contain information about all levels. The ‘level’ object will contain information about the current level. This algorithm will set the ‘initial_capital’ attribute of the ‘level’ object to the correct value depending on the id of the chosen stock by looping through the ‘levels’ array. This algorithm is needed because it is easier to save level settings once but in a different object than loop through the ‘levels’ array every time the program needs to get some information about the chosen level.

```
FUNCTION getInitialCapital
        FOR i IN RANGE FROM 0 TO length(levels)
                IF stock.id == levels[i].id
                        level.initial_capital = levels[i].initial_capital
                ENDIF
        ENDFOR
ENDFUNCTION
```

Drawing a new chart

Using an official Plotly.js documentation (https://plotly.com/javascript/), I found a method that creats a plot that takes in 3 parameters: container, data and layout. The name of the method is newPlot. The container is just an HTML element that will wrap the whole chart. The data is an array containing an object with various attributes. The layout is configuration options of the plot. At this stage of the design, I am not sure which layout options I will choose. The algorithm for creating a new plot is essential to my simulation, as I will need to have two plots: candlestick and moving average charts. However, the pseudocode for this algorithm is the following:

```
FUNCTION drawNewPlot
        Plotly.newPlot(container, data, layout)
ENDFUNCTION
```

Extending candlestick chart

Having created a new plot for the candlestick chart, I will need to add new data to it every few seconds to extend it throughout the simulation. In this algorithm, I will select various attributes, which are all arrays, and append some new data to their ends. As discussed in the data structures section, candlestick charts have five attributes: x, which is date, open, high, low and close.

```
FUNCTION extendCandlestickChart
        candlestick.data[0].x.append(stock.data[stock.count].date)
        candlestick.data[0].open.append(stock.data[stock.count].open)
        candlestick.data[0].high.append(stock.data[stock.count].high)
        candlestick.data[0].low.append(stock.data[stock.count].low)
        candlestick.data[0].close.append(stock.data[stock.count].close)
ENDFUNCTION
```

Extending moving average chart

Having created a new plot for the moving average chart, I will need to add new data to it every few seconds to extend it throughout the simulation. In this algorithm, I will select various attributes, which are all arrays, and append some new data to their ends. According to the documentation, in order to create a line graph, objects in the data array will need to have two main attributes: x and y. The first object in the data array will represent the short-term moving average and the second object will represent the long-term moving average. Later, I will create a getSimpleMovingAverage(day) function that will take in one parameter, which is the time period in days. Finally, I will create a getWeightedMovingAverage(day) function if the user decides to use another type of moving average. X-coordinates will represent dates. In order to set y-coordinates, the program will call one of those functions depending on the type of moving average the user chooses.

```
FUNCTION extendMovingAverageChart
        moving_average.data[0].x.append(stock.data[stock.count].date)
        moving_average.data[0].y.append(getSimpleMovingAverage(day_short))
        moving_average.data[1].x.append(stock.data[stock.count].date)
        moving_average.data[1].y.append(getSimpleMovingAverage(day_long))
ENDFUNCTION
```

Updating a chart

Having created a prototype where I extend candlestick and moving average charts, I encountered a problem: the charts would not update. Then, having checked some examples from the documentation, I realised that I need to update those charts. It turns out, there is a method in the library that does exactly what I need. The method is called ‘update’ and it takes in three parameters: container, data and layout. With this method, I was able to make the charts update as soon as I add a new piece of data.

```
FUNCTION updatePlot
        Plotly.update(container, data, layout)
ENDFUNCTION
```

Calculating simple moving average

In order to plot a moving average chart, I will need to get y-coordinates for the line graphs. If the user selects Simple Moving Average in the settings, I will need to call a function called getSimpleMovingAverage and set y-coordinate of the line graph to be equal to the return of that function. The function will take in one parameter which is a time period in days. Other variables will be global and will not change, so there will be no need to pass them to the function as parameters.

```
FUNCTION getSimpleMovingAverage(day)
        sum = 0
        average = 0

        FOR i IN RANGE FROM stock.count – day + 1 TO stock.count
                sum = sum + stock.data[i].close
        ENDFOR

        average = sum / day
        return average
ENDFUNCTION
```

Calculating weighted moving average

If the user selects Weighted Moving Average in the settings, I will need to call a function called getWeightedMovingAverage and set y-coordinate of the line graph to be equal to the return of that function. The function will take in one parameter which is a time period in days. Other variables will be global and will not change, so there will be no need to pass them to the function as parameters.

```
FUNCTION getWeightedMovingAverage(day)
        sum = 0
        average = 0
        weight = 0
        total_weight = 0

        FOR i IN RANGE FROM stock.count – day + 1 TO stock.count
                weight = weight + 1
                sum = sum + weight * stock.data[i].close
                total_weight = total_weight + weight
        ENDFOR

        average = sum / total_weight
        return average
ENDFUNCTION
```

Calculating slope of a line

At some point, the program will need to calculate slopes of lines to find out if the intersection is a golden cross or a death cross. The function will take in four parameters, which are four points and calculate the gradient. If the x-coordinates are the same, it is impossible to calculate the gradient, as we will be dividing by 0.

```
FUNCTION getSlope(x1, y1, x2, y2)
        IF x1 != x2
                return (y2 – y1) / (x2 – x1)
        ENDIF

        return false
ENDFUNCTION
```

Checking if two lines intersect given two points for each line

In order to detemine whether to buy or sell a stock, the program will need to search for intersections of moving averages. Every time there is a new piece of data added to the moving average chart, I will check if line segments intersect by taking in the coordinates of four points. Those points will be the two new added pieces of data to each line graph and the ones before them. The algorithm for this is the following:

```
FUNCTION doIntersect(x1, y1, x2, y2, x3, y3, x4, y4)
        a1 = y2 - y1
        b1 = x1 - x2
```

```
        c1 = x2 * y1 - x1 * y2

        r3 = a1 * x3 + b1 * y3 + c1
        r4 = a1 * x4 + b1 * y4 + c1

        IF r3 != 0 AND r4 != 0 AND isSameSign(r3, r4)
                return false
        ENDIF

        a2 = y4 - y3
        b2 = x3 - x4
        c2 = x4 * y3 - x3 * y4

        r1 = a2 * x1 + b2 * y1 + c2
        r2 = a2 * x2 + b2 * y2 + c2

        IF r1 != 0 AND r2 != 0 AND isSameSign(r1, r2)
                return false
        ENDIF

        denominator = a1 * b2 - a2 * b1

        IF denominator == 0
                return true
        ENDIF

        return true
ENDFUNCTION
```

Checking if intersection is a Golden Cross or Death Cross

In order to make a prediction about the direction of the price of a stock, the program will need to compare gradients of the two line graphs if there is an intersection. This algorithm will return 'true' if the intersection is a Golden Cross and 'false' if the intersection is a Death Cross.

```
FUNCTION isGoldenCross(slope_short, slope_long)
        return slope_short > slope_long
ENDFUNCTION
```

Checking if two numbers have the same sign

The program will need to check if two gradients have the same sign and in order to do so, I will use a built-in Math library. It has a method called 'sign' that I will use in the following way:

```
FUNCTION isSameSign(a, b)
        return Math.sign(a) == Math.sign(b)
ENDFUNCTION
```

Checking if two candles form a Bearish Kicker pattern

In order to predict the direction of the price of a stock, the program will need to check if there is a pattern in the two newest pieces of data in the candlestick chart. The algorithm to check if there is a Bearish Kicker pattern is the following:

```
FUNCTION isBearishKicker
        return pattern.prev.open < pattern.prev.close AND
                pattern.curr.open > pattern.curr.close AND
                pattern.prev.open > pattern.curr.open
ENDFUNCTION
```

Checking if two candles form a Bullish Kicker pattern

The algorithm to check if there is a Bullish Kicker pattern is the following:

```
FUNCTION isBullishKicker
        return pattern.prev.open > pattern.prev.close AND
                pattern.curr.open < pattern.curr.close AND
```

```
                pattern.prev.open < pattern.curr.open
ENDFUNCTION
```

Checking if two candles form a Shooting Star pattern

The algorithm to check if there is a Shooting Star pattern is the following:

```
FUNCTION isShootingStar
        return pattern.prev.open < pattern.prev.close AND
                pattern.curr.open > pattern.curr.close AND
                pattern.prev.close < pattern.curr.close AND
                pattern.curr.high - pattern.curr.close > 2 * (pattern.curr.open - pattern.curr.close)
AND
                pattern.curr.open - pattern.curr.close > pattern.curr.close - pattern.curr.low
ENDFUNCTION
```

Updating pattern data

The ‘pattern’ object will need to be updated every time there is a new piece of data added to the candlestick chart. Its attributes, ‘curr’ and ‘prev’ will be set to different values, as the stock count updates every few seconds.

```
FUNCTION updatePatternData
        pattern.prev = stock.data[stock.count - 1]
        pattern.curr = stock.data[stock.count]
ENDFUNCTION
```

Buying a stock

The user should be able to buy stocks if they have enough money available. The program will check what number the user put into the ‘number of stocks’ input field, check if the user has enough funds and make the transaction if the validation check is passed. The algorithm for this is the following:

```
FUNCTION buy
        number = value in the ‘number of stocks’ input field
        total_price = number * stock.price

        IF total_price < wallet.money_available
                wallet.owned_stocks = wallet.owned_stocks + number
                wallet.money_available = wallet.money_available – total_price
        ELSE
                display an error message saying the user does not have enough funds
        ENDIF

        update the wallet
ENDFUNCTION
```

Selling a stock

The user should be able to sell stocks if they own any. The program will check what number the user put into the ‘number of stocks’ input field, check if the user has enough stocks and make the transaction if the validation check is passed. The algorithm for this is the following:

```
FUNCTION sell
        number = value in the ‘number of stocks’ input field
        total_price = number * stock.price

        IF wallet.owned_stocks >= number
                wallet.owned_stocks = wallet.owned_stocks – number
                wallet.money_available = wallet.money_available – total_price
        ELSE
                display an error message saying the user does not have enough stocks
        ENDIF
ENDFUNCTION
```

Highlighting candlestick patterns

The program will need to highlight candlestick patterns. In order to do so, there will be a ‘shapes’ array in the ‘layout’ object of the ‘candlestick’ object. This algorithm will just append an object with several attributes to the ‘shapes’ array. Values for the attributes were found in the documentation of Plotly.js.

```
FUNCTION updateCandlestickShapes
        IF isBearishKicker OR isBullishKicker OR isShootingStar
                candlestick.layout.shapes.append({
                        type: 'rect',
                        xref: 'x',
                        yref: 'paper',
                        x0: pattern.prev.date,
                        y0: 0,
                        x1: pattern.curr.date,
                        y1: 1,
                        fillcolor: '#ffff00',
                        opacity: 0.4,
                        line: { width: 0 }
                })
        ENDIF
ENDFUNCTION
```

Annotating candlestick patterns

The program will need to annotate candlestick patterns. In order to do so, there will be an ‘annotations’ array in the ‘layout’ object of the ‘candlestick’ object. This algorithm will just append an object with several atributes to the ‘annotations’ array. Values for the attributes were found in the documentation of Plotly.js.

```
FUNCTION updateCandlestickAnnotations
        IF isBearishKicker THEN text = ‘Bearish Kicker’
        ELSE IF isBullishKicker THEN text  = ‘Bullish Kicker’
        ELSE IF isShootingStar THEN text = ‘Shooting Star’

        IF isBearishKicker OR isBullishKicker OR isShootingStar
                candlestick.layout.annotations.append({
                        x: pattern.prev.date,
                        y: 1,
                        xref: 'x',
                        yref: 'paper',
                        text: text,
                        font: { color: 'black', size: 10 },
                        showarrow: false,
                        xanchor: 'left',
                        ax: 0,
                        ay: 0
                })
        ENDIF
ENDFUNCTION
```

Highlighting moving average intersections

The program will need to highlight intersections of moving averages. In order to do so, there will be a ‘shapes’ array in the ‘layout’ object of the ‘moving_average’ object. This algorithm will just append an object with several attributes to the ‘shapes’ array. Values for the attributes were found in the documentation of Plotly.js.

```
FUNCTION updateMovingAverageShapes
        IF doIntersect
                moving_average.layout.shapes.append({
                        type: 'rect',
                        xref: 'x',
                        yref: 'paper',
                        x0: pattern.prev.date,
                        y0: 0,
                        x1: pattern.curr.date,
                        y1: 1,
                        fillcolor: '#ffff00',
                        opacity: 0.4,
                        line: { width: 0 }
                })
```

```
        ENDIF
ENDFUNCTION
```

Annotating moving average intersections

The program will need to annotate moving average intersections. In order to do so, there will be an 'annotations' array in the 'layout' object of the 'moving_average' object. This algorithm will just append an object with several atributes to the 'annotations' array. Values for the attributes were found in the documentation of Plotly.js.

```
FUNCTION updateMovingAverageAnnotations
        IF doIntersect
                IF isGoldenCross THEN suggestion = 'BUY'
                ELSE suggestion = 'SELL'

                moving_average.layout.annotations.append({
                        x: pattern.prev.date,
                        y: 1,
                        xref: 'x',
                        yref: 'paper',
                        text: suggestion,
                        font: { color: 'black', size: 10 },
                        showarrow: false,
                        xanchor: 'left',
                        ax: 0,
                        ay: 0
                })
        ENDIF
ENDFUNCTION
```

Test data for development

Milestones that my program needs to achieve

1. Creating the main interfaces
2. Selecting a level
3. Displaying the wallet and level requirements for each stock
4. Drawing a candlestick chart
5. Drawing a moving average chart
6. Highlighting and annotating candlestick patterns
7. Highlighting and annotating intersections of moving averages
8. Allow the user to trade the chosen stock
9. Allow the user to hide/show candlestick patterns
10. Allow the user to stop the simulation and view results
11. Allow the user to change the type of moving average and its parameters

Milestone №1

Test №	Test and input	Output
1	The user can see the starting screen of the simulation, screen with a selection of levels and the main screen of the simulation.	The user can see all screens clearly and all information that needs to be visible is visible.

Milestone №2

Test №	Test and input	Output
1	The user can select one of three levels and they will be redirected to the correct web page.	The three levels are displayed to a user with some information about them and each of them redirects the user to a correct web page.

Milestone №3

Test №	Test and input	Output
1	When the user is redirected to the main screen of the simulation, they can see the virtual wallet and level requirements for each stock.	When the user selects a level, they are redirected to the main screen of the simulation, that displays the virtual wallet and level requirements for the chosen level.

Milestone №4

Test №	Test and input	Output
1	The user can see the candlestick chart with stock data on the main screen of the simulation.	The user can see the candlestick chart with stock data, as well as interact with the chart with their mouse on the main screen of the simulation.

Milestone №5

Test №	Test and input	Output
1	The user can see the moving average chart with correctly calculated values.	The user can clearly see the moving average chart and hover over each data point with their mouse to see exact values.

Milestone №6

Test №	Test and input	Output
1	Candlestick patterns are highlighted and annotated on the candlestick chart with a suggestion whether to buy or sell the stock.	The user can see various highglighted candlestick patterns on the candlestick chart with annotations and suggestion whether to buy or sell the stock.

Milestone №7

Test №	Test and input	Output
1	Intersection points of the moving averages are highlighted and annotated with a suggestion whether to buy or sell the stock.	The user can clearly see the highlighted intersection points of the moving averages with an annotation suggesting whether to buy or sell the stock.

Milestone №8

Test №	Test and input	Output
1	The user can input how many stocks they want to buy or sell and if the validation checks are passed, the transaction will happen and the wallet will be updated.	The user can input the number of stocks they want to buy or sell. Then, if the validation checks are passed, ie the user has enough funds to buy the stock or enough stocks to sell, the transaction will take place and the user's wallet will be updated.
2	The validation checks were described in the Validation of inputs section and the test data is presented below.	If the validation of inputs is not passed, the user receives an error message saying exactly what is wrong. If the validation checks are passed, the transaction happens and the user's wallet is updated.
3	money_available: $1000 stocks_owned: 5 stock.price: $100 Valid inputs for buying stocks: 0 – 10	If 0 is inputted, nothing will happed, so we can consider it as a valid input.

	Valid inputs for selling stocks: 0 - 5	In this case, the user can only buy maximum of 10 stocks, as 10 * $100 = $1000, which is how much money the user has. In this case, the user can only sell up to 5 stocks, as they own only 5 stocks and they are not allowed to sell more stocks than they own.

Milestone №9

Test №	Test and input	Output
1	The user can choose whether to show or hide candlestick patterns. If they choose to hide them, no highlights, annotations and suggestions will be displayed on the candlestick chart. However, if the user chooses to show candlestick patterns, they can see highglights, annotations and suggestions as usual.	The user can hide or show candlestick patterns and the candlestick chart is updated according to the option chosen by the user.

Milestone №10

Test №	Test and input	Output
1	The user can stop the simulation at any moment and view the results.	When the user presses the 'Stop simulation' button, they are redirected to a different page where the results of the simulation are shown.

Milestone №11

Test №	Test and input	Output
1	The user can change the type of moving average and its parameters	When the user changes the type of moving average or parameters, the program deletes all the previous data from the moving average chart and new data points are being added.

Test data for post-development

Integration testing

№	Test and input	Expected output
1	Enter the url in the search bar of the browser	The starting screen of the simulation can be seen
2	Press the 'start trading' button	The user is redirected to a different page where they can see a selection of levels
3	Select the first level	The user is redirected to the main screen of the simulation. They can see the charts being updated, virtual wallet, level requirements, boxes for entering parameters and changing settings and all the elements of the interface
4	Candlestick patterns are highlighted and annotated	The user can see highlighted and annotated candlestick patterns
5	Moving average intersections are highlighted and annotated	The user can see highglighted and annotated intersections of moving averages
6	Try buying the stock with the following cases: 1. valid number of stocks and the user has enough funds	1. money available decreases by the total price and owned stocks increases by the number of stocks 2. error message 'not enough funds'

	2. valid number of stocks and the user does not have enough funds 3. invalid number of stocks and the user has enough funds 4. invalid number of stocks and the user does not have enough funds	3. error message ‘must be an integer greater than zero’ 4. error message ‘must be an integer greater than zero’, if resolved then ‘not enough funds’
7	Try selling the stock with the following cases: 1. valid number of stocks and the user has enough stocks 2. valid number of stocks and the user does not have enough stocks 3. invalid number of stocks and the user has enough stocks 4. invalid number of stocks and the user does not have enough stocks	1. money available is increased by the total price and owned stocks is decreased by the number of stocks 2. error message ‘not enough stocks’ 3. error message ‘must be an integer greater than zero’ 4. error message ‘must be an integer greater than zero’, if resolved then ‘not enough stocks’
8	Try changing moving average parameters with the following cases: 1. Only type 2. Only short-term average 3. Only long-term average 4. Both averages 5. Type and both averages	In all cases, the data from the moving average chart must be deleted. 1. Values are calculated using a formula for the chosen type of the moving average 2. Short-term average values will be calculated using more or less data points depending on the input 3. Long-term average will be calculated using more or less data points depending on the input 4. Both averages will be calculated using more or less data points depending on the inputs 5. Both averages will be calculated using a formula for the chosen type of the moving avreage and both will be calculated using more or less data points depending on the inputs
9	Check the pattern recognition checkbox	The user can see highlighted and annotated candlestick patterns
10	Uncheck the pattern recognition checkbox	The user no longer can see highlighted and annotated candlestick patterns
11	Profit is calculated correctly	1. Take the current price of the stock 2. Multiply it by the number of stocks owned 3. Add money available 4. Subtract initial capital 5. Round to the whole number 6. Check is the value the program calculates is the same
12	Press the ‘stop simulation’ button	The user is redirected to a different page and they can see their results calculated correctly. If their profit is less than the required profit, the level is not passed. If their profit is greater than the required profit, the level is passed.

Alpha testing

Having tested the program myself, I will present the complete solution to my stakeholders, who will test the program by verifying that each success criteria from the Analysis section is complete.

Development

Project setup

I decided to use Visual Studio Code as my IDE, because it supports HTML, CSS, JavaScript code, I am familiar with the IDE and I have it installed on my computer. Then, I created a folder *trading-simulation* and opened it in the IDE.

Firstly, I created three HTML files:

- *index.html* – the starting page of my simulation
- *choose-level.html* – the page where the user would select a level
- *trade-stock.html* – the page where the user would trade the chosen stock

Secondly, I created a CSS file *style.css*, where I will write code for styling my project.

Thirdly, I created a JavaScript file *app.js*, where I will write only the main logic for my project, and a folder *modules*, where I will create additional JavaScript files with various functions to keep the code structured and more readable.

Finally, I created a folder *json* and pasted all the JSON files with stock data:

- *APPL.json* - Apple
- *TSLA.json* - Tesla
- *NKE.json* – Nike

A problem I encountered while downloading stock data is that YAHOO Finance only allowed me to download stock data in csv format. I solved this problem by using a csv to json converter.

Milestone №1 – creating the main interfaces

In this milestone, I will create the main interfaces of my project. I will try to make them similar to what I planned in the design section, but they may deviate a little bit.

Code listing

index.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Trading simulation</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <div id="index">
      <h1>Trading simulation</h1>
      <button id="start-trading-btn">start trading</button>
    </div>
  </body>
</html>
```

choose-level.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Choose level</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <div id="choose-level">
      <h1>Choose a level</h1>
      <div id="levels"></div>
    </div>
  </body>

  <script type="module">
    import { levels } from './modules/level.js'

    (function() {
      let levels_html = ``

      for (let i = 0; i < levels.length; i++) {
        let name = levels[i].name
        let id = levels[i].id
        let initial_capital = levels[i].initial_capital
        let required_profit = levels[i].required_profit
        let difficulty = levels[i].difficulty

        let level_html = `
          <a class="level ${difficulty}" id="${id}">
            <h2>${name} (${id})</h2>
            <h3>Initial capital: $${initial_capital}</h3>
            <h3>Required profit: $${required_profit}</h3>
            <h3>Difficulty: ${difficulty}</h3>
          </a>`

        levels_html = levels_html + level_html
      }

      document.getElementById('levels').innerHTML = levels_html
    })()
  </script>

</html>
```

trade-stock.html

```
<!DOCTYPE html>
<html>

  <head>
```

```html
    <title>Trade stock</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <div id="trade-stock">
      <div id="charts">
        <div id="candlestick-chart"></div>
        <div id="moving-average-chart"></div>
      </div>

      <div id="information">
        <h1 id="stock-id"></h1>
        <div>
          <h2>Wallet</h2>
          <h3 id="money-available"></h3>
          <h3 id="owned-stocks"></h3>
        </div>

        <div>
          <h2>Level</h2>
          <h3 id="required-profit"></h3>
          <h3 id="profit"></h3>
        </div>

        <div>
          <h2>Buy / Sell</h2>
          <div class="input-container">
            <label for="number-of-stocks">Number of stocks</label>
            <input type="number" id="number-of-stocks" required>
          </div>
          <button id="buy-btn">BUY</button>
          <button id="sell-btn">SELL</button>
          <p id="error-message"></p>
        </div>

        <div>
          <h2>Settings</h2>
          <div class="input-container">
            <label for="short-term-input">Short-term input</label>
            <input type="number" id="short-term-input" value="5">
          </div>
          <div class="input-container">
            <label for="long-term-input">Long-term input</label>
            <input type="number" id="long-term-input" value="15">
          </div>
          <select id="options">
            <option value="sma">Simple Moving Average</option>
            <option value="wma">Weighted Moving Average</option>
          </select>
          <button id="change-settings-btn">APPLY</button>
```

```
        </div>

        <div>
          <h2>Pattern Recognition</h2>
          <div class="checkbox-container">
            <input type="checkbox" id="pattern-recognition-checkbox" checked>
            <label for="pattern-recognition-checkbox">show</label>
          </div>
        </div>

        <div>
          <button id="stop-simulation-btn">Stop simulation</button>
        </div>
      </div>
    </div>
  </body>

  <script src='https://cdn.plot.ly/plotly-latest.min.js'></script>
  <script src="app.js" type="module"></script>

</html>
```

style.css

```
* {
  margin: 0;
  padding: 0;
}

h1 {
  font-family: 'Roboto';
  font-weight: 600;
  font-size: 48px;
  line-height: 52px;
  color: #000;
}

h2 {
  font-family: 'Roboto';
  font-weight: 400;
  font-size: 24px;
  line-height: 28px;
  color: #000;
}

h3 {
  font-family: 'Roboto';
  font-weight: 400;
  font-size: 16px;
  line-height: 20px;
  color: #000;
}

button {
  margin-top: 7px;
```

```
  padding: 5px 10px;
  border: 1px solid #000;
  border-radius: 5px;
  font-family: 'Roboto';
  font-size: 14px;
  font-weight: 400;
  color: #000;
}

button:hover {
  cursor: pointer;
}

/* INDEX */
#index {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);

  display: flex;
  flex-direction: column;
  align-items: center;
}

#index button {
  margin-top: 25px;
  padding: 10px 15px;
  font-size: 20px;
  background-color: rgb(80, 202, 80);
}

/* CHOOSE-LEVEL */
#choose-level {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);

  display: flex;
  flex-direction: column;
  align-items: center;
}

#levels {
  display: flex;
}

.level {
  margin: 10px;
  padding: 15px;
  border: 1px solid #000;
  border-radius: 10px;
  text-decoration: none;
```

```
  color: #000;
}

.level h2 {
  margin-bottom: 15px;
}

.level:hover {
  cursor: pointer;
}

.easy {
  background-color: rgb(80, 202, 80);
}

.medium {
  background-color: orange;
}

/* TRADE-STOCK */
#stock-id {
  margin-bottom: 10px;
  text-align: center;
}

#trade-stock {
  display: flex;
}

#charts {
  width: calc(100% - 350px);
  max-width: 1000px;
}

#information {
  padding: 5px 20px;
  position: fixed;
  top: 0;
  right: 0;
  width: 300px;
  height: 100vh;
  background-color: rgb(239, 239, 239);
}

#information > div {
  margin-bottom: 15px;
}

input {
  width: 100%;
}

.checkbox-container {
  display: flex;
```

```
  justify-content: flex-start;
  align-items: center;
}

#pattern-recognition-checkbox {
  margin-right: 5px;
  width: fit-content;
}
```

level.js

```
export const levels = [
  {
    name: 'Apple',
    id: 'APPL',
    initial_capital: 1000,
    required_profit: 100,
    difficulty: 'easy'
  },
  {
    name: 'Tesla',
    id: 'TSLA',
    initial_capital: 1000,
    required_profit: 200,
    difficulty: 'easy'
  },
  {
    name: 'Nike',
    id: 'NKE',
    initial_capital: 10000,
    required_profit: 5000,
    difficulty: 'medium'
  }
]
```

candlestick.js

```
export let candlestick = {
  container: 'candlestick-chart',
  data: [{
    x: [],
    open: [],
    high: [],
    low: [],
    close: [],
    type: 'candlestick'
  }],
  layout: {
    dragmode: 'zoom',
    title: 'Candlestick Chart',
    shapes: [],
    annotations: [],
    xaxis: {
      autorange: true,
      type: 'date',
      rangeslider: { visible: false }
```

```
    },
    yaxis: {
      autorange: true,
      type: 'linear',
      title: 'Price($)'
    }
  }
}
```

moving-average.js

```
export let moving_average = {
  day_short: 5,
  day_long: 15,
  type: 'sma',
  container: 'moving-average-chart',
  data: [
    {
      x: [],
      y: [],
      type: 'scatter',
      mode: 'lines',
      name: 'Short-term SMA'
    },
    {
      x: [],
      y: [],
      type: 'scatter',
      mode: 'lines',
      name: 'Long-term SMA'
    }
  ],
  layout: {
    dragmode: 'zoom',
    title: 'Moving Average Chart',
    shapes: [],
    annotations: [],
    xaxis: {
      autorange: true,
      type: 'date',
      rangeslider: { visible: false },
    },
    yaxis: {
      autorange: true,
      type: 'linear',
      title: 'Price($)'
    }
  }
}
```

plot.js

```
export function drawNewPlot(plot) {
  Plotly.newPlot(plot.container, plot.data, plot.layout);
}
```

app.js

```
import { candlestick } from './modules/charts/candlestick.js'
import { moving_average } from './modules/charts/moving-average.js'
import { drawNewPlot } from './modules/charts/plot.js'

(async () => {
  // charts
  drawNewPlot(candlestick)
  drawNewPlot(moving_average)
})()
```

Explanation of the code

Firstly, I created three HTML files *index.html, choose-level.html and trade-stock.html.* Then, I created the CSS file *style.css* and linked it to all of the HTML files so that I can style those pages. Also, I connected some external fonts from Google Fonts (https://fonts.google.com/) to all of the HTML pages. After that I filled those pages with content by referring to DevDocs documentation (https://devdocs.io/).

Initially, in the *choose-level.html* file I had all the levels listed as HTML code, but all the information about the levels was hard-coded, which I did not like. Therefore, I wrote some code to solve this problem. I created a *level.js* file inside the *modules* folder and created an array called *levels*. The structure of the array is the same as described in the Data Scructures section. Then, in the *choose-level.html* file I wrote some code that iterates through the *levels* array and inserts the same HTML code as before into the HTML element with *id="levels".* After writing this code, I had the same result as before, when I just used HTML, but the information about the levels is now not hard-coded. It means that if I ever want to change some level requirements, I only need to change the *levels* array, which improves the usability of the code.

Also, I used the Plotly.js documentation (https://plotly.com/javascript/) to create empty charts for the candlestick and moving average charts. I created a folder *charts* and two files *candlestick.js* and *moving-average.js* inside the folder.

Inside the *candlestick.js* file, I created an object called *candlestick*. Inside the *moving-average.js* file, I created an object *moving-average*. They both have a lot of attributes which I found in the documentation. The only attribute I added to both objects is the name of the HTML elements inside which I want the charts to appear.

In the *plot.js* file, I created a function *drawNewPlot,* which calles a method *newPlot* that I found in the documentation. It takes in three parameters, which are *container, data, layout* and displays the chart to the screen.

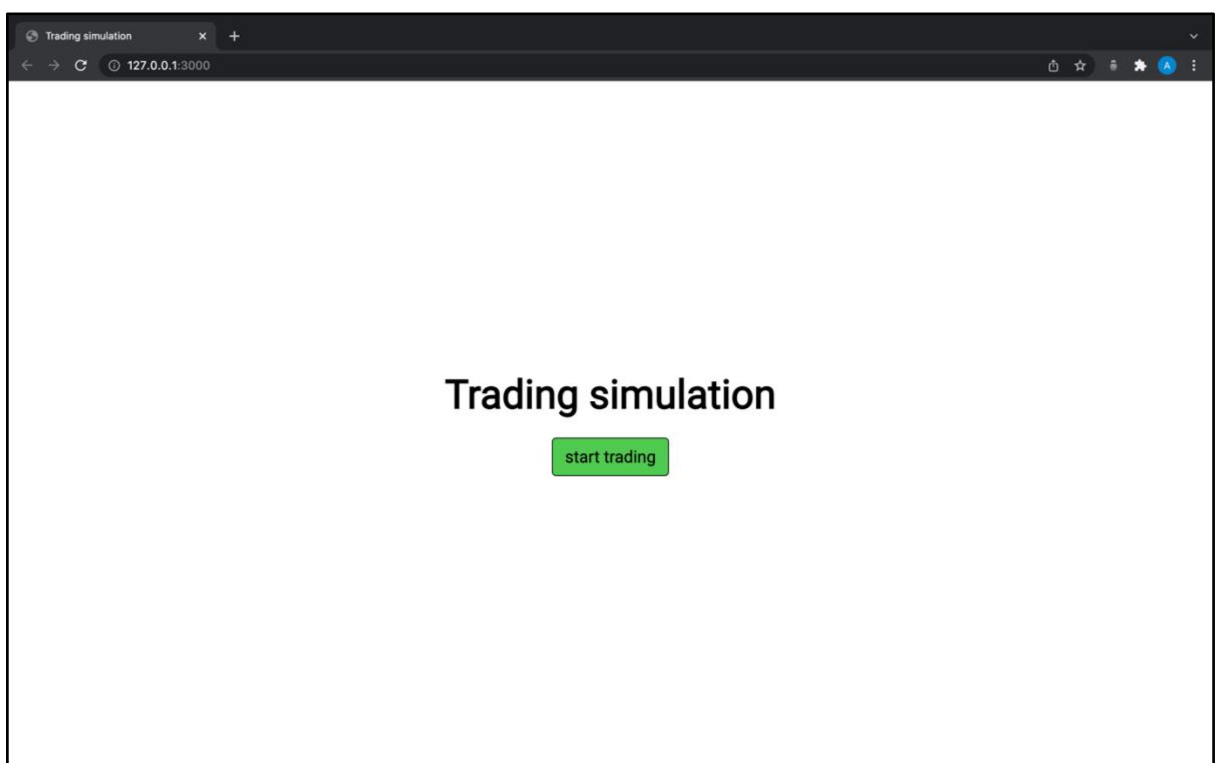
In the *app.js* file, I imported the *candlestick* and *moving_average* objects, as well as the *drawNewPlot* function. Then, I created an immediately invoked function expression, that would execute automatically as soon as the file loads. Inside it, I called the *drawNewPlot* function twice to display both charts to the screen.

Finally, I linked the *app.js* file as well as the link to code that *Plotly.js* requires in the *trade-stock.html* file.

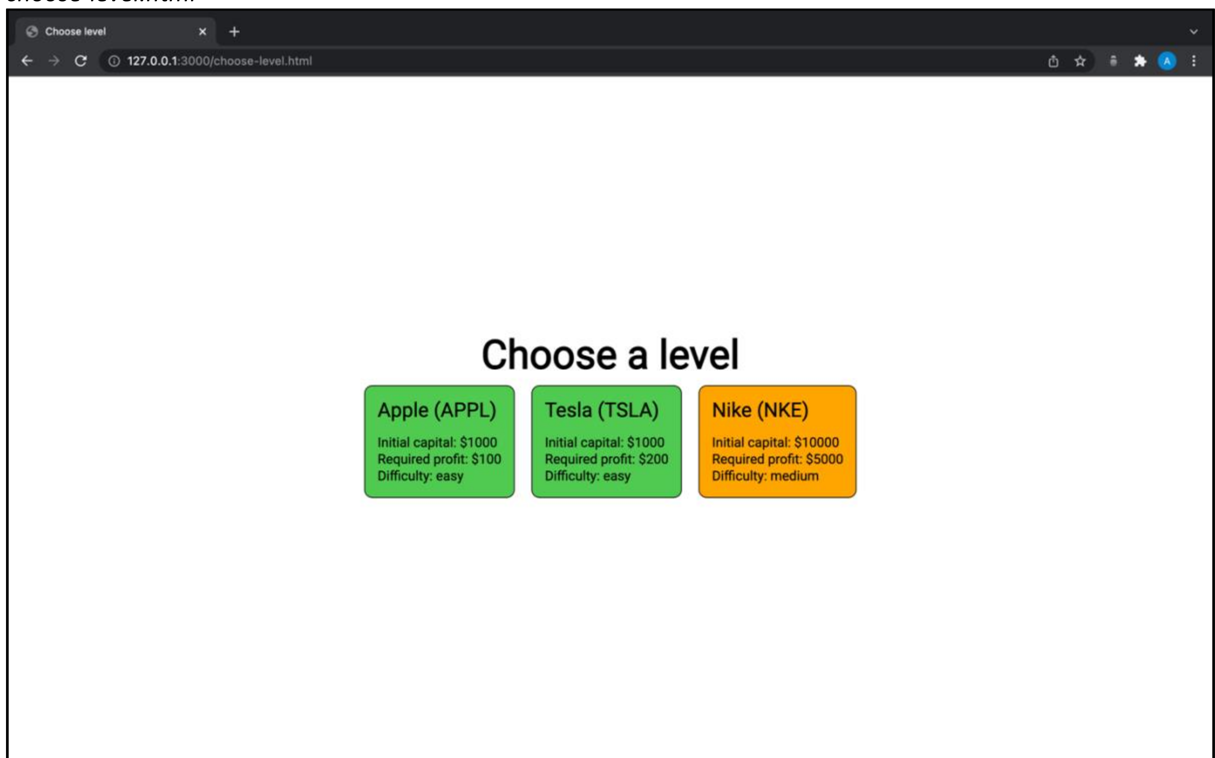
Testing

I tested my code by manually adding some data to the *trade-stock.html* file. Other files had all the required data. I got the following results:

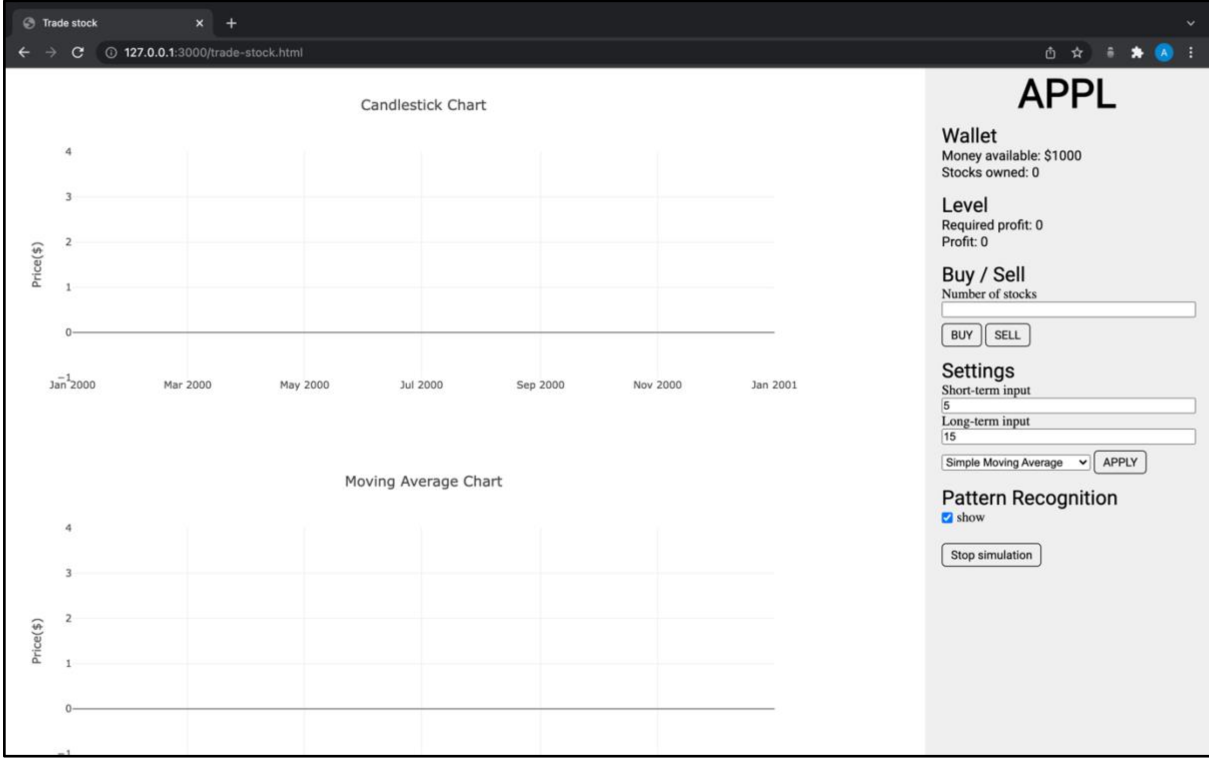
index.html



choose-level.html



trade-stock.html



The interfaces look very similar to those in the Design section. However, in the Design section, I forgot to add the 'stop simulation' button at the right bottom corner of the screen. I will have a meeting with my stakeholders in a couple of days, when I will ask for their opinion on the design of the main interfaces.

Reflection

While creating the interfaces, I tried to be as close as possible to the interfaces I created in the Design section. Also, I improved the usability of the code by importing the information about the levels to the *choose-level.html* file from the *levels* array. By doing that, adding new levels in the future will be much easier and the program will be more maintainable.

Milestone №2 – selecting a level

In this milestone, I will implement page redirections and selecting a level. There is one redirection from *index.html* to *choose-level.html* and another one from *choose-level.html* to *trade-stock.html*.

Code listing

index.html

```
<script>
  document.getElementById('start-trading-btn').addEventListener('click', () => { window.open('choose-level.html', '_self') })
</script>
```

choose-level.html

```
let links = document.getElementsByTagName('a')

for (let i = 0; i < links.length; i++) {
  links[i].addEventListener('click', () => { window.open(`trade-stock.html?stock_id=${links[i].id}`, '_self')
})
}
```

Explanation of the code

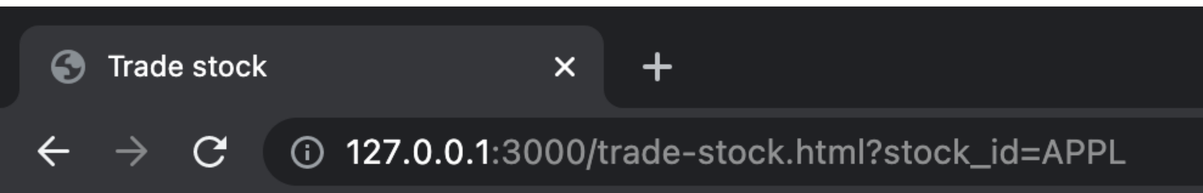
In order to redirect the user from *index.html* to *choose-level.html*, I added an event listener to the button on the starting screen of the simulation. When the button is clicked, a function *window.open()* will redirect the user to the *choose-level.html* page.

In order to redirect the user from *choose-level.html* to *trade-stock.html*, I selected all *<a>* elements on the *choose-level.html* page and stored them inside the *links* array. Then, I iterated through the array and added an event listener to each of the elements. Since the program needs to pass which level the user has selected, I decided to pass this information as a parameter in the url. The program will know which level the user has selected, because each *<a>* element has an *id*, which is unique to each stock.

Testing

I tested the first redirection by simply clicking the *start trading* button on the starting screen and I was correctly redirected to the *choose-level.html* page.

I tested the second redirection by clicking all the levels one by one and looking at the url. Each time I was correctly redirected to the page *trade-stock.html*, but there was also a parameter in the url, such as *stock_id=APPL*. The evidence of this is shown below:



Reflection

At first, I was planning to add an additional attribute each level in the *levels* array, where I would store a link. However, I thought it would not be necessary, as the only thing that changes is the parameter in the url and it is the same as the id of the stock. So I came up with the idea of adding a link to each level using JavaScript.

Milestone №3 - displaying the wallet and level requirements for each stock

In this milestone, I will make sure that the correct information is shown for each level, such as the wallet and level requirements.

Code listing

stock.js

```
export let stock = {
  id: '',
  url: '',
  data: [],
  price: 0,
  count: 0,
  getId: function() {
    let queryString = window.location.search
    let urlParams = new URLSearchParams(queryString)
    this.id = urlParams.get('stock_id')
  },
  displayId: function() {
```

```
    document.getElementById('stock-id').textContent = this.id
  },
  getUrl: function() {
    this.url = './json/' + this.id + '.json'
  },
  getData: async function() {
    let response = await fetch(stock.url)
    this.data = await response.json()
  },
  getPrice: function() {
    this.price = this.data[this.count].close
  }
}
```

level.js

```
import { stock } from './stock.js'

export let level = {
  initial_capital: 0,
  required_profit: 0,
  getInitialCapital: function() {
    for (let i = 0; i < levels.length; i++) {
      if (stock.id === levels[i].id) {
        this.initial_capital = levels[i].initial_capital
      }
    }
  },
  getRequiredProfit: function() {
    for (let i = 0; i < levels.length; i++) {
      if (stock.id === levels[i].id) {
        this.required_profit = levels[i].required_profit
      }
    }
  },
  displayRequiredProfit: function() {
    document.getElementById('required-profit').textContent = 'Required profit: ' + this.required_profit
  }
}
```

wallet.js

```
import { stock } from "./stock.js"
import { level } from "./level.js"

export let wallet = {
  money_available: 0,
  owned_stocks: 0,
  profit: 0,
  setMoneyAvailable: function() {
    this.money_available = level.initial_capital
  },
  displayMoneyAvailable: function() {
    document.getElementById('money-available').textContent = 'Money available: ' +
Math.round(this.money_available)
  },
```

```
  displayOwnedStocks: function() {
    document.getElementById('owned-stocks').textContent = 'Stocks owned: ' + this.owned_stocks
  },
  getProfit: function() {
    this.profit = Math.round(this.money_available + this.owned_stocks * stock.price - level.initial_capital)
  },
  displayProfit: function() {
    document.getElementById('profit').textContent = 'Profit: ' + this.profit
  },
  update: function() {
    this.displayMoneyAvailable()
    this.displayOwnedStocks()
    this.getProfit()
    this.displayProfit()
  }
}
```

app.js

```
...
import { stock } from './modules/stock.js'
import { level } from './modules/level.js'
import { wallet } from './modules/wallet.js

(async () => {
  // stock.js
  stock.getId()
  stock.displayId()
  stock.getUrl()
  stock.getData()
    .then(() => { stock.getPrice() })

  // level.js
  level.getInitialCapital()
  level.getRequiredProfit()
  level.displayRequiredProfit()

  // wallet.js
  wallet.setMoneyAvailable()
  wallet.update()

  ...
})()
```

Explanation of the code

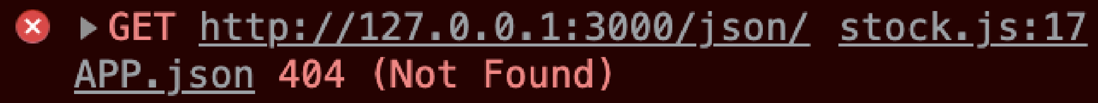
- *stock.js* – as discussed in the Data sctructures section, I created the *stock* object. It has the same attributes as I planned in the Design section. However, instead of writing functions outside the object, I decided to write them as methods of the *stock* object. In this file, I used the *getStockId, fetchStockData, displayStockId, getStockPrice, getStockData* algorithms from the Algorithms section, slightly modified and renamed them. However, the functionality of those algorithms did not change significantly. The method *getId* would get the *id* of the chosen stock from the url and store it in the *id* attribute of the *stock* object. The *displayId* method would paste the id of the stock into the appropriate HTML element. The *getUrl* method would simply concetenate the path to the *json* folder, the id of the stock and the file extension to get a valid path to the file that stores the data for the chosen stock and store it in the *url* attribute. The *getData* method would store the stock data from

the correct file in the *data* attribute. The *getPrice* method would return the current price of the chosen stock by accessing the correct element in the *data* array. I had some problems implementing the *getData method*, but I was able to resolve them after reading the MDN documentation about the fetch function (https://developer.mozilla.org/en-US/docs/Web/API/fetch).

- *level.js* – firstly, I created the *level* object with the same attributes as discussed in the Data Sctructures section. Then, I wrote the *getInitialCapital, getRequiredProfit, displayRequiredProfit* methods by referring to the corresponding functions in the Algorithms section. The *getInitialCapital* and *getRequiredProfit* functions iterate through the *levels* array and search for an object with the same id as the id of the chosen stock. When they find such an object, they copy the values of the *initial_capital* and *required_profit* attributes to the *level* object.
- *wallet.js* – firstly, I imported *stock* and *level* variables into the *wallet.js* file, as I will need to use some information from those variables. Then, I created the *wallet* object, as intended in the Data Structures section. The *setMoneyAvailable method* will be called once at the beginning of the simulation to set *money_available* to the same value as the *initial_capital*. The *getProfit* method calculates the current profit and the *displayMoneyAvailable*, *displayOwnedStocks* and *displayProfit* methods display information into appropriate HTML elements. The *update* method simply updates all attributes of the *wallet* object. I will need to update the wallet frequently, so I decided to create this function to save some lines of code and decrease the chance of encountering a bug.
- *app.js* – I imported the *stock, level, wallet* objects. Then, I called their methods in the specific order: methods that do not depend on other objects are called first. For example, as I import the *stock* object into the *level.js* file, the methods of the *level* object are called after the methods of the *stock* object.

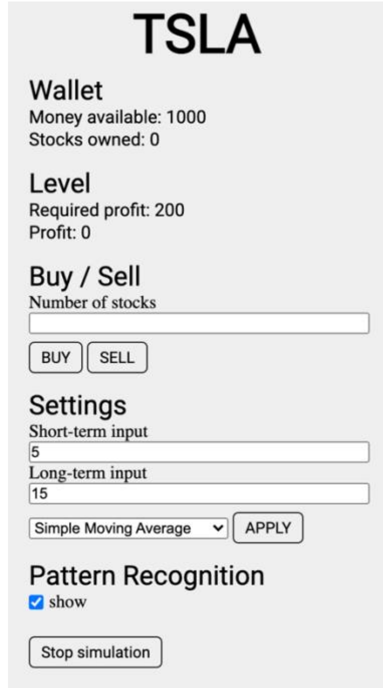
Testing

Firstly, I checked what would happen if I intentionally passed an incorrect *stock_id* in the url and got the following result in the browser console:



Now, as a developer, I would know when an incorrect *stock_id* is passed in the url, which can save me time during debugging in the future.

Secondly, I ran the code again with the correct *stock_id=TSLA* and got the following result:



I checked that all information is correct for each of the stocks.

Thirdly, I checked that the JSON data is fetching correctly by outputting it to the browser console:

```
▼data: Array(252)
  ▶[0 … 99]
  ▶[100 … 199]
  ▶[200 … 251]
   length: 252
 id: "APPL"
 price: 82.875
 url: "./json/APPL.json"
```

After expanding the *data* attribute, I checked that all of the JSON data matches the stock data in the corresponding file, which means all data is fetched correctly.

Reflection

While designing the program, I did not think that fetching stock data might take some time. Due to that reason, I encountered a problem that the program would throw an error, because it was trying to get the price of the stock and the data has not been received yet. Therefore, I had to use asynchronous JavaScript, which means I can control in which order functions execute. To solve the problem I encountered, I had to get the price of the stock only when the stock data has been fetched by the program with the use of *then()* method.

Milestone №4 – drawing a candlestick chart

In this milestone, I will implement drawing and updating the candlestick chart with the correct stock data.

Code listing

stock.js

```
export let stock = {
  ...
  incrementCount: function() {
    this.count = this.count + 1
  }
}
```

candlestick.js

```
import { stock } from "../stock.js"

export let candlestick = {
  ...
  extend: function() {
    if (this.data[0].x.length >= 12) {
      this.data[0].x.shift()
      this.data[0].open.shift()
      this.data[0].high.shift()
      this.data[0].low.shift()
      this.data[0].close.shift()
    }
    this.data[0].x.push(stock.data[stock.count].date)
    this.data[0].open.push(stock.data[stock.count].open)
    this.data[0].high.push(stock.data[stock.count].high)
    this.data[0].low.push(stock.data[stock.count].low)
    this.data[0].close.push(stock.data[stock.count].close)
  }
}
```

plot.js

```
export function updatePlot(plot) {
  Plotly.update(plot.container, plot.data, plot.layout)
}
```

app.js

```
...
import { updatePlot } from './modules/charts/plot.js'

(async () => {
  ...
  let interval_constant = 1000

  let interval = setInterval(() => {
    // update stock price
    stock.getPrice()

    // update candlestick chart
    candlestick.extend()
    updatePlot(candlestick)

    // update stock count
    stock.incrementCount()

  }, interval_constant)
```

```
})()
```

Explanation of the code

Firstly, as discussed in the Algorithms section, I created a method that would update the counter and called it *incrementCount.*

Secondly, in the *candlestick.js* file, I imported the *stock* object and created a new method called *extend.* Thid method would control the number of candles that is shown on the graph and append new candles to it by appending and removing items to and from the *data* array.

Thirdly, in the *plot.js* file, I created a new function that would update a chart and called it *updatePlot.*

Finally, in the *app.js* file, I imported the new *updatePlot* function. Then, I created an interval, which is a code block that executes with a constant rate. To control the rate, I created a variable called *interval_constant* and set its value to 1Hz (1 execution per second). Inside the interval, I update the price of the stock, append new data to the candlestick chart, update the candlestick chart and increment the counter.

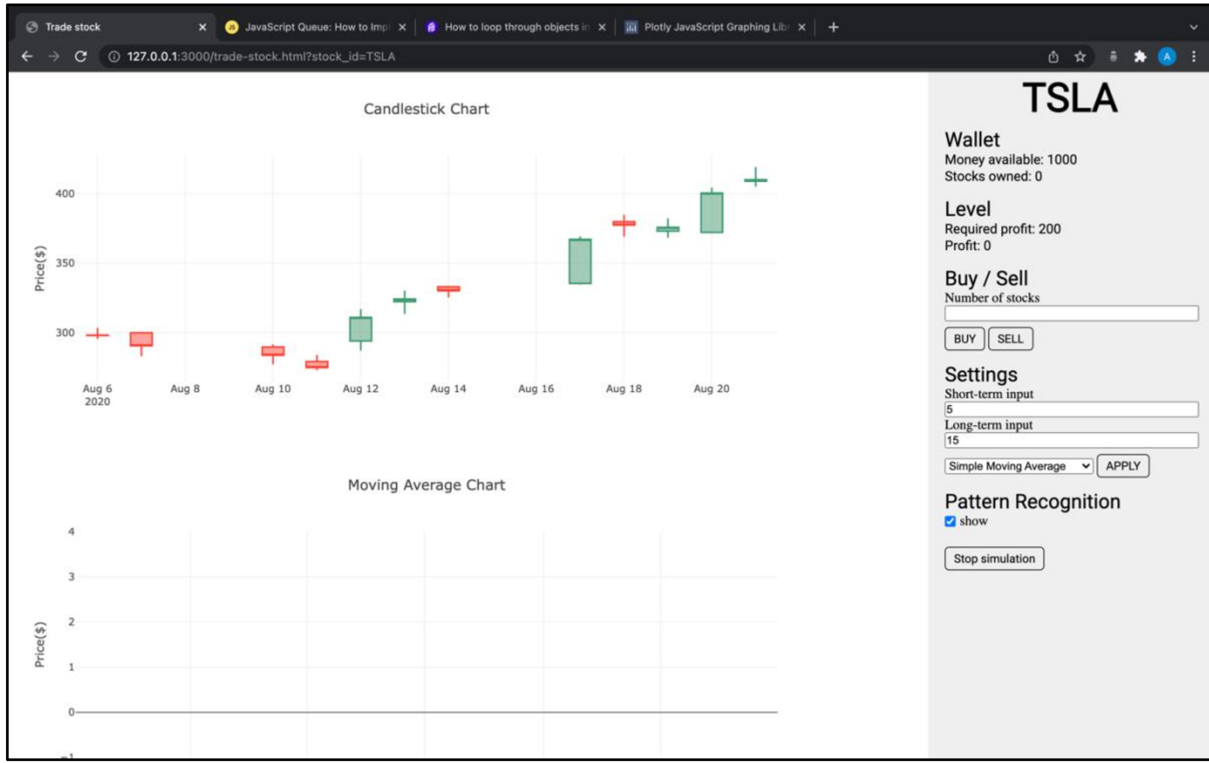
I tried implementing *x, close, open, high, low* arrays as queues, but it did not work and I am not sure why. I will try to come back to this improvement later if I have time.

```
function Queue() {
  this.elements = []
}

Queue.prototype.enqueue = function (e) {
  this.elements.push(e)
}

Queue.prototype.dequeue = function () {
   return this.elements.shift()
}

Queue.prototype.isEmpty = function () {
   return this.elements.length == 0
}

Queue.prototype.peek = function () {
   return !this.isEmpty() ? this.elements[0] : undefined
}

Queue.prototype.length = function() {
   return this.elements.length
}
```

```
export let candlestick = {
  container: 'candlestick-chart',
  data: [{
    x: new Queue(),
    open: new Queue(),
    high: new Queue(),
    low: new Queue(),
    close: new Queue(),
    type: 'candlestick'
  }],
```

Testing

I ran the code and got the following result:



The candlestick chart works as intended and when the number of candles is greater than or equal to 12, the first candle is deleted, keeping the chart clean. Also, I checked the dates and values of several candles and they match the contents of the *TSLA.json* file, which means the data is appended correctly and the chart is updated correctly as well. I repeated the same procedure for each level and everything seems to work correctly.

Reflection

Even though in the Design section I planned to use built-in JavaScript arrays to store the stock data, during the development I came up with an idea to use queues, as it would be a more efficient solution. However, I did not manage to implement queues and I will try to make this improvement at the end of the development process if I will have any remaining time.

Milestone №5 – drawing a moving average chart

In this milestone, I will implement drawing and updating the moving average chart with the correct stock data.

Code listing

average.js

```
import { stock } from '../stock.js'

export function getSimpleMovingAverage(day) {
  let sum = 0
  let average = 0

  for (let i = stock.count - day + 1; i <= stock.count; i++) {
    sum = sum + stock.data[i].close
  }

  average = sum / day
```

```
  return average
}

export function getWeightedMovingAverage(day) {
  let sum = 0
  let average = 0
  let weight = 0
  let total_weight = 0

  for (let i = stock.count - day + 1; i <= stock.count; i++) {
    weight++
    sum = sum + weight * stock.data[i].close
    total_weight = total_weight + weight
  }

  average = sum / total_weight
  return average
}
```

moving-average.js

```
import { stock } from "../stock.js"
import { getSimpleMovingAverage, getWeightedMovingAverage } from './average.js'

export let moving_average = {
  ...
  extendShortMovingAverage: function() {
    if (this.data[0].x.length >= 14) {
      this.data[0].x.shift()
      this.data[0].y.shift()
    }

    if (stock.count >= this.day_short) {
      this.data[0].x.push(stock.data[stock.count].date)

      let y = (this.type === 'sma')
        ? getSimpleMovingAverage(this.day_short)
        : getWeightedMovingAverage(this.day_short)

      this.data[0].y.push(y)
    }
  },
  extendLongMovingAverage: function() {
    if (this.data[1].x.length >= 14) {
      this.data[1].x.shift()
      this.data[1].y.shift()
    }

    if (stock.count >= this.day_long) {
      this.data[1].x.push(stock.data[stock.count].date)

      let y = (this.type === 'sma')
        ? getSimpleMovingAverage(this.day_long)
        : getWeightedMovingAverage(this.day_long)
```

```
      this.data[1].y.push(y)
    }
  }
}
```

app.js

```
...

(async () => {
  ...

  let interval = setInterval(() => {
    ...
    // update moving average chart
    moving_average.extendShortMovingAverage()
    moving_average.extendLongMovingAverage()
    updatePlot(moving_average)
    ...
  }, interval_constant)
})()
```

Explanation of the code

First of all, I created the functions *getSimpleMovingAverage* and *getWeightedMovingAverage*. Each of these functions take in a parameter *day.* I found the steps for calculating simple moving average and weighted moving average on Trading View (https://www.tradingview.com/):

Simple Moving Average (SMA)

Simple Moving Average is an unweighted Moving Average. This means that each day in the data set has equal importance and is weighted equally. As each new day ends, the oldest data point is dropped and the newest one is added to the beginning.

CALCULATION

An example of a 3 period SMA

```
Sum of Period Values / Number of Periods

Closing Prices to be used: 5, 6, 7, 8, 9

First Day of 3 Period SMA: (5 + 6 + 7) / 3 = 6
Second Day of 3 Period SMA: (6 + 7 + 8) / 3 = 7
Third Day of 3 Period SMA: (7 + 8 + 9) /3 = 8
```

Weighted Moving Average (WMA)

Weighted Moving Average is similar to the SMA, except the WMA adds significance to more recent data points. Each point within the period is assigned a multiplier (largest multiplier for the newest data point and then descends in order) which changes the weight or significance of that particular data point. Then, just like the SMA, once a new data point is added to the beginning, the oldest data point is thrown out.

Calculation

Is similar to the SMA except it adds a weight (multiplier) to each period. The most recent period has the most weight). An example of a 5 period WMA

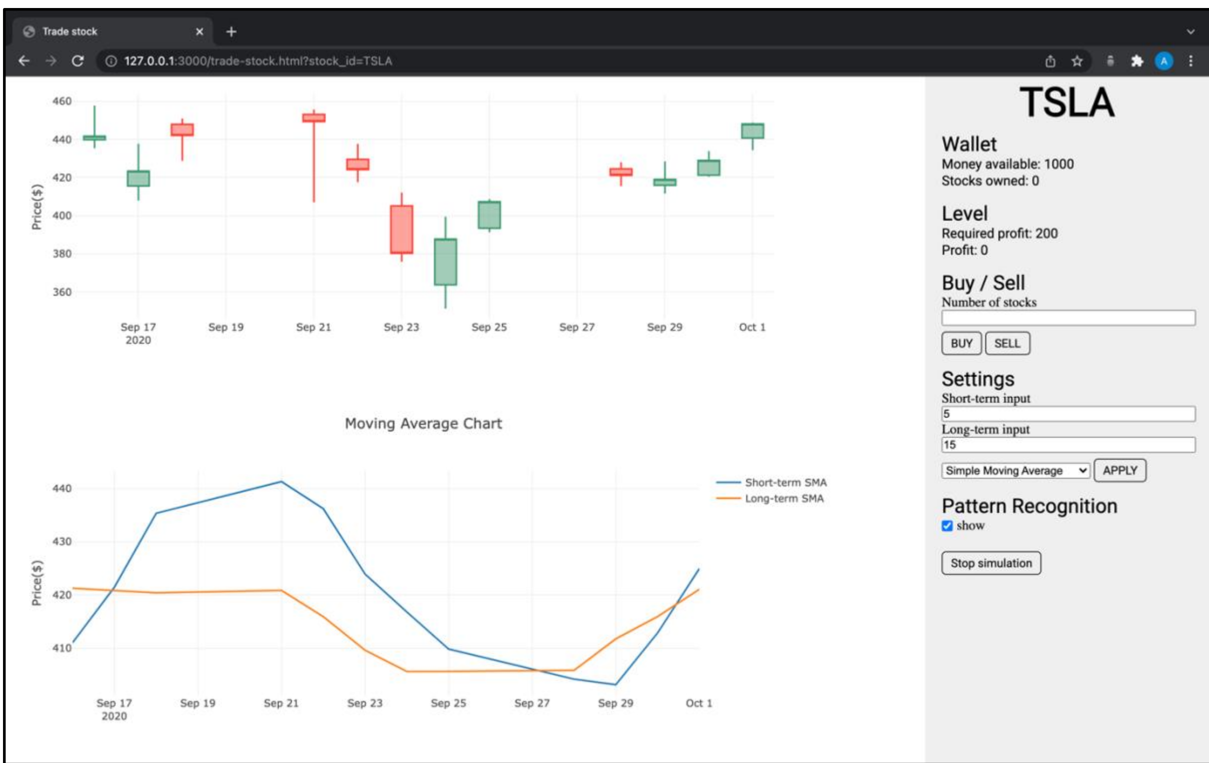
```
  Sum of Period Values / Number of Periods

Closing Prices to be used: 5, 6, 7, 8, 9

Period        Weight          Price      Weighted Average
    1           1       X       5              5
    2           2       X       6              12
    3           3       X       7              21
    4           4       X       8              32
    5           5       X       9              45

TOTAL    =      15      X      35      =      525

WMA = (Sum of Weighted Averages) / (Sum of Weight)

WMA   115 / 15 = 7.6667
```

Then, I created the methods *extendShortMovingAverage* and *extendLongMovingAverage*. Those functions delete points from the graph if the number of points is greater than or equal to 12 to keep the chart clean and readable. Then, they append new points to the chart for both short-term and long-term moving averages, taking into account which type of moving average the user has selected.

Finally, in the *app.js*, I imported *extendShortMovingAverage* and *extendLongMovingAverage*, called them inside the interval and after that added a line that would update the moving average chart.

Testing

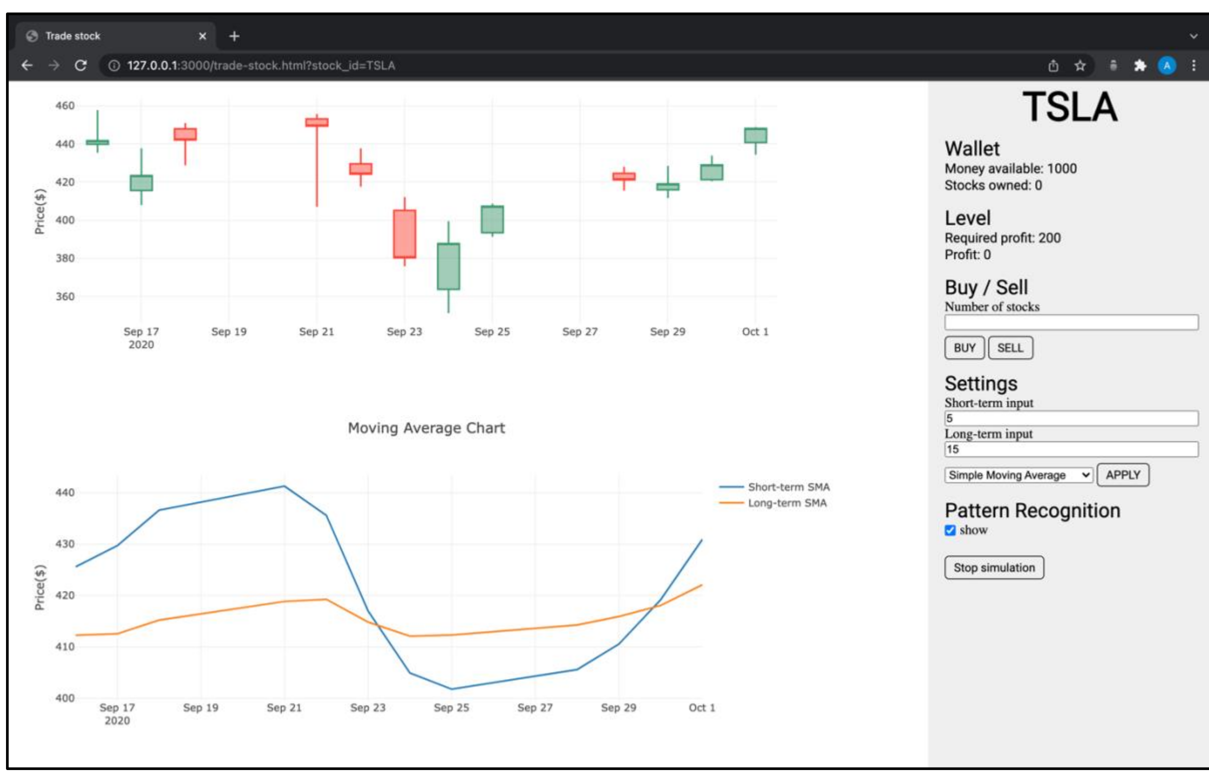
After I ran the code I got the following result:



In the screenshot, a simple moving average is selected and seems to be working correctly. I checked the first few values for the simple moving average and they match the values on the chart. However, to test the milestone fully, I need a way of changing the type of moving average. I do not have such a functionality yet, but I changed the *type* of moving average in the code to *wma*:

```
export let moving_average = {
  day_short: 5,
  day_long: 15,
  type: 'wma',
```

After I ran the code again, I got the following result:



This time I got a different graph for the moving average. Having manually calculated the first few values for the weighted moving average, the values on the graph matched the calculated values, which means all the functions work correctly.

Reflection

For this milestone, I also wanted to make a use of queues. However, as it did not work in the previous milestone, I decided that I will not waste time now, as it might not work again, and will try to make this improvement at the end of the development process.

Meeting with stakeholders

Having completed half of the milestones, I decided to meet with my stakeholders to tell them about the progress that has been made and ask for their opinion on the finished features. Firstly, they like how the interfaces look: 'very similar to the ones we agreed on in the Design section'. Secondly, they like the way the charts are drawn and being updated. Also, they do not seem to dislike anything so far and are looking forward to seeing the rest of the features. However, they pointed out that the charts are being updated too quickly and it is difficult to keeep up with the pace. Therefore, I decided to increase the *interval_constant* from 1 second to 2 seconds. They seemed to like the idea. We agreed on having another meeting when I finish all the features.

Milestone №6 - highlighting and annotating candlestick patterns

In this milestone, I will implement highlighting and annotating candlestick patterns on the candlestick chart.

Code listing

candlestick-patterns.js

```
export function isBearishKicker(prev, curr) {
  return prev.open < prev.close &&
```

```
        curr.open > curr.close &&
        prev.open > curr.open
}

export function isBullishKicker(prev, curr) {
  return prev.open > prev.close &&
        curr.open < curr.close &&
        prev.open < curr.open
}

export function isShootingStar(prev, curr) {
  return prev.open < prev.close &&
        curr.open > curr.close &&
        prev.close < curr.close &&
        curr.high - curr.close > 2 * (curr.open - curr.close) &&
        curr.open - curr.close > curr.close - curr.low
}
```

new-annotation.js

```
export function newAnnotation(startDate, text) {
  return {
    x: startDate,
    y: 1,
    xref: 'x',
    yref: 'paper',
    text: text,
    font: { color: 'black', size: 8 },
    showarrow: false,
    xanchor: 'left',
    ax: 0,
    ay: 0
  }
}
```

new-shape.js

```
export function newShape(startDate, endDate) {
  return {
    type: 'rect',
    xref: 'x',
    yref: 'paper',
    x0: startDate,
    y0: 0,
    x1: endDate,
    y1: 1,
    fillcolor: '#ffff00',
    opacity: 0.4,
    line: { width: 0 }
  }
}
```

pattern.js

```
import { isBearishKicker, isBullishKicker, isShootingStar } from './candlestick-patterns.js'
import { stock } from '../stock.js'

```

```
export let pattern = {
  show: true,
  prev: {},
  curr: {},
  name: '',
  updatePrev: function() {
    this.prev = stock.data[stock.count - 1]
  },
  updateCurr: function() {
    this.curr = stock.data[stock.count]
  },
  updateName: function() {
    if (isBearishKicker(this.prev, this.curr)) {
      this.name = 'Bearish Kicker'
    } else if (isBullishKicker(this.prev, this.curr)) {
      this.name = 'Bullish Kicker'
    } else if (isShootingStar(this.prev, this.curr)) {
      this.name = 'Shooting Star'
    } else {
      this.name = ''
    }
  }
}
```

candlestick.js

```
import { pattern } from '../pattern-recognition/pattern.js'
import { newShape } from '../new-layout/new-shape.js'
import { newAnnotation } from '../new-layout/new-annotation.js'

export let candlestick = {
  ...
  updateShapes: function() {
    if (this.layout.shapes.length > 12) this.layout.shapes.shift()
    let new_shape = pattern.name ? newShape(pattern.prev.date, pattern.curr.date) : null
    this.layout.shapes.push(new_shape)
  },
  updateAnnotations: function() {
    if (this.layout.annotations.length > 12) this.layout.annotations.shift()
    let new_annotation = pattern.name ? newAnnotation(pattern.prev.date, pattern.name) : null
    this.layout.annotations.push(new_annotation)
  }
}
```

app.js

```
...
import { pattern } from './modules/pattern-recognition/pattern.js'

(async () => {
  ...

  let interval = setInterval(() => {
    ...
    // update candlestick chart
    candlestick.extend()
```

```
    if (stock.count > 1) {
      pattern.updatePrev()
      pattern.updateCurr()
      pattern.updateName()

      if (pattern.show) {
        candlestick.updateShapes()
        candlestick.updateAnnotations()
      }
    }

    updatePlot(candlestick)
    ...
  }, interval_constant)
})()
```

Explanation of the code

Firstly, I created the functions *isBearishKicker, isBullishKicker, isShootingStar* in the *candlestick-pattern.js* file. These functions are needed to check whether there is a pattern in the two given data points. Each function returns true if it has found a pattern and false otherwise.

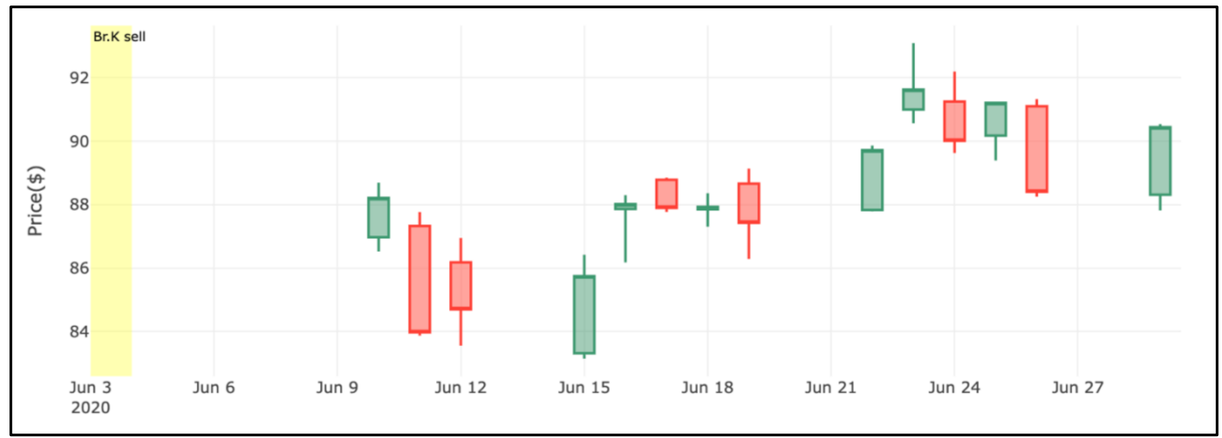
Secondly, I created the functions *newAnnotation* and *newShape*. These functions return objects that should be inserted in the *shapes* or *annotations* arrays of the *layout* object to highlight and annotate candlestick patterns on the chart.

Thirdly, I created the *pattern* object in the *pattern.js* file. As discussed in the Data Structures section, it has four attributes. It also has three methods. The *updatePrev* and *updateCurr* methods simply update the *prev* and *curr* attributes depending on the counter. The *updateName* method updates the *name* attribute depending on which function from the *candlestick-patterns.js* returns true.

Then, in the *candlestick.js* file, I imported the above objects and created two new methods *updateShapes* and *updateAnnotations.* These methods delete the old highlights and annotations from the graph and append new ones depending on the information in the *pattern* object.

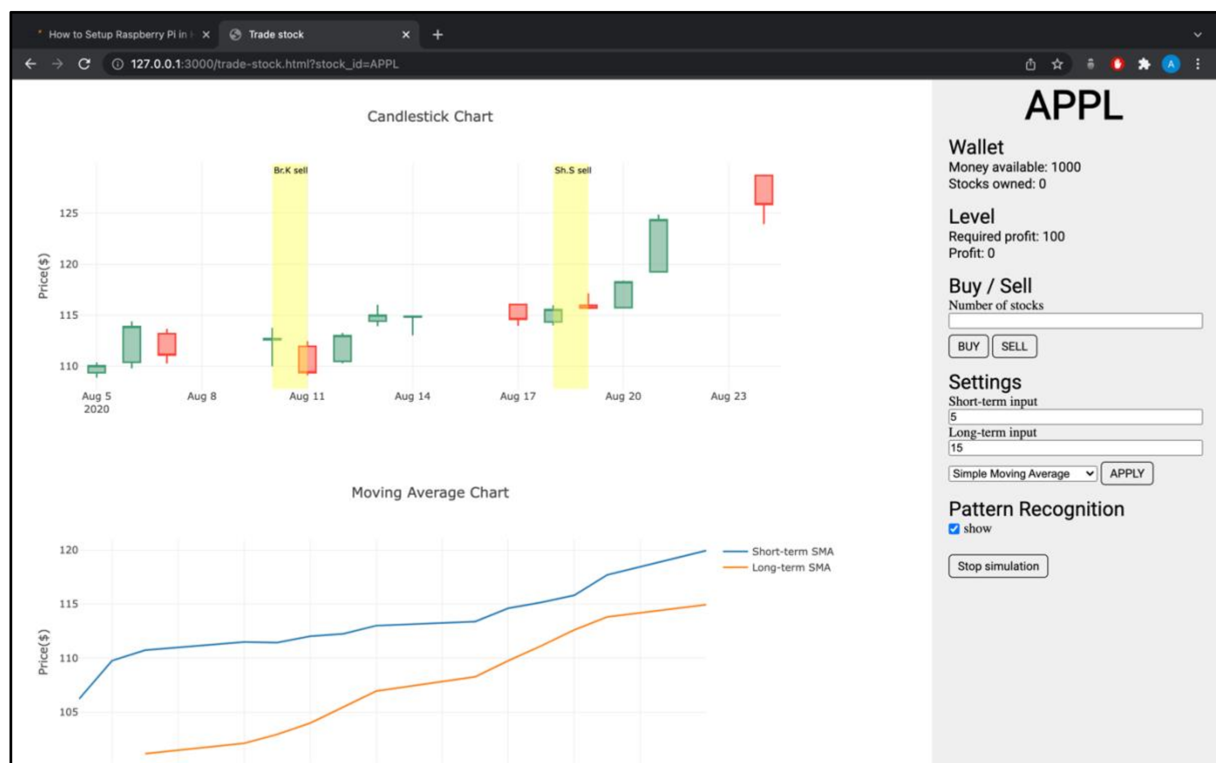
Finally, in the *app.js* file, I update attributes of the *pattern* object and if the attribute *show* is true, then the program calls the *updateShapes* and *updateAnnotations* methods of the *candlestick* object to update the highlights and annotations on the chart. However, before updating attributes in the *pattern* object, I check that the counter is greater than 1, because otherwise the *prev* attribute will not find the information it needs, as there is no element with an index of -1 in any array, and an error will be thrown.

If no pattern were found then *null* will be added to the *shapes* and *annotations* arrays. I decided to add *null* when no patterns were found just to keep track when to delete the old candlestick patterns. If I did not delete old candlestick patterns, the chart would look like this and the yellow rectangle with text will never disappear from the chart:



Testing

I ran the code, waited until the program has found any patterns, and got the following result:



In order to test the feature, I used Trading View. I selected a stock called 'Apple' and scrolled until I found the same date I have, which is 5/8/2020. Then, I selected which patterns I am looking for and Trading View found the same patterns as my program, which means there is a high chance that the feature works correctly. In order to fully test the feature, I will need to ask my stakeholders to check if my program finds all candlestick patterns that it is searching for.

Reflection

I decided to slightly deviate from the design and break this problem down even further. I created two separate functions *newShape* and *newAnnotation*, because the code was becoming unreadable. Other than that, I followed the algorithms and data structured from the Design section.

Milestone №7 – highlighting and annotating intersections of moving averages

In this milestone, I will implement highlighting and annotating intersections of moving averages on the moving average chart.

Code listing

get-slope.js

```
export function getSlope(y1, y2) {
  return (y2 - y1)
}
```

is-golden-cross.js

```
export function isGoldenCross(slope_short, slope_long) {
  return slope_short > slope_long
}
```

is-same-sign.js

```
export function isSameSign(a, b) {
  return Math.sign(a) == Math.sign(b)
}
```

do-intersect.js

```
import { isSameSign } from './is-same-sign.js'
import { stock } from '../stock.js'

export function doIntersect(prev, curr) {
  let x1 = stock.count - 1, y1 = prev.short.y
  let x2 = stock.count, y2 = curr.short.y
  let x3 = stock.count - 1, y3 = prev.long.y
  let x4 = stock.count, y4 = curr.long.y
  let a1, b1, c1, a2, b2, c2
  let r1, r2, r3, r4
  let denom

  a1 = y2 - y1
  b1 = x1 - x2
  c1 = x2 * y1 - x1 * y2

  r3 = a1 * x3 + b1 * y3 + c1
  r4 = a1 * x4 + b1 * y4 + c1

  if (r3 !== 0 && r4 !== 0 && isSameSign(r3, r4)) return false

  a2 = y4 - y3
  b2 = x3 - x4
  c2 = x4 * y3 - x3 * y4

  r1 = a2 * x1 + b2 * y1 + c2
  r2 = a2 * x2 + b2 * y2 + c2

  if (r1 !== 0 && r2 !== 0 && isSameSign(r1, r2)) return false

  denom = a1 * b2 - a2 * b1
```

```
  if (denom === 0) return true

  return true
}
```

point.js

```
import { doIntersect } from './do-intersect.js'
import { moving_average } from '../charts/moving-average.js'

export let point = {
  prev: {
    short: {},
    long: {}
  },
  curr: {
    short: {},
    long: {}
  },
  intersect: false,
  updatePrev: function() {
    this.prev.short = {
        x: moving_average.data[0].x[moving_average.data[0].x.length - 2],
        y: moving_average.data[0].y[moving_average.data[0].y.length - 2]
    }
    this.prev.long = {
        x: moving_average.data[1].x[moving_average.data[1].x.length - 2],
        y: moving_average.data[1].y[moving_average.data[1].y.length - 2]
    }
  },
  updateCurr: function() {
    this.curr.short = {
        x: moving_average.data[0].x[moving_average.data[0].x.length - 1],
        y: moving_average.data[0].y[moving_average.data[0].y.length - 1]
    }
    this.curr.long = {
        x: moving_average.data[1].x[moving_average.data[1].x.length - 1],
        y: moving_average.data[1].y[moving_average.data[1].y.length - 1]
    }
  },
  updateIntersect: function() {
    this.intersect = doIntersect(this.prev, this.curr)
  }
}
```

moving-average.js

```
import { newShape } from '../new-layout/new-shape.js'
import { newAnnotation } from '../new-layout/new-annotation.js'
import { point } from '../intersection/point.js'
import { getSlope } from '../intersection/get-slope.js'
import { isGoldenCross } from '../intersection/is-golden-cross.js'

export let moving_average = {
  ...
  updateShapes: function() {
```

```
    if (this.layout.shapes.length > 12) this.layout.shapes.shift()

    let new_shape = point.intersect ? newShape(point.prev.short.x, point.curr.short.x) : null

    this.layout.shapes.push(new_shape)
  },
  updateAnnotations: function() {
    if (this.layout.annotations.length > 12) this.layout.annotations.shift()

    let slope_short = getSlope(point.prev.short.y, point.curr.short.y)
    let slope_long = getSlope(point.prev.long.y, point.curr.long.y)
    let text = isGoldenCross(slope_short, slope_long) ? 'Buy' : 'Sell'

    let new_annotation = point.intersect ? newAnnotation(point.prev.short.x, text) : null

    this.layout.annotations.push(new_annotation)
  }
}
```

app.js

```
...
import { point } from './modules/intersection/point.js'

(async () => {
  ...

  let interval = setInterval(() => {
    ...
    // update moving average chart
    moving_average.extendShortMovingAverage()
    moving_average.extendLongMovingAverage()

    if (moving_average.data[1].x.length > 1) {
      point.updatePrev()
      point.updateCurr()
      point.updateIntersect()

      moving_average.updateShapes()
      moving_average.updateAnnotations()
    }

    updatePlot(moving_average)
    ...
  }, interval_constant)
})()
```

Explanation of the code

The *getSlope* function calculates the slope between two points. There is no need to consider x-coordinates, as the difference of x-coordinates for short and long moving averages is the same.

The *isGoldenCross* function returns *true* if the slope of the short-term moving average is greater than the slope of the long-term moving average and false otherwise.

The *isSameSign* function returns true if the signs of the parameters are the same and false otherwise.

The doIntersect function takes in two parameters: *prev* and *curr.* They are previous and current data points of the moving average chart. The function finds the equation between two points of the short and long-term moving averages. Then, it checks whether two lines intersect considering their Cartesian equations. If the program finds that two lines intersect, it returns *true.* Otherwise, it returns *false.*

In the *point.js* file, I created an object called *point*. It has three attributes:

- *prev* stores the previous data point of the moving average chart.
- *curr* stores the current data point of the moving average chart
- *intersect* holds *true* if the two lines intersect and *false* otherwise

It also has the following methods:

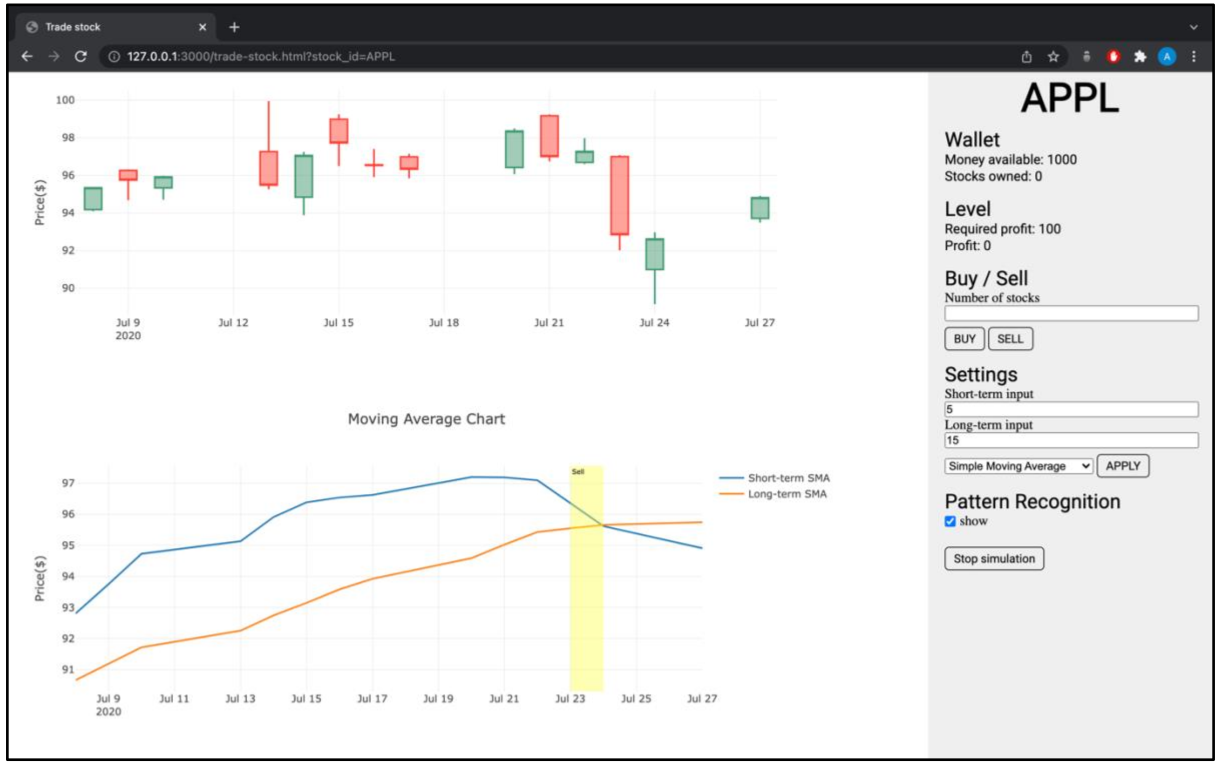
- *updatePrev* updates the data for the previous data point of the moving average chart.
- *updateCurr* updates the data for the current data point of the moving average chart.
- *updateIntersect* updates the *intersect* attribute by checking if two lines constructed from *prev* and *curr* intersect.

In the *moving-average.js* file, I created new methods *updateShapes* and *updateAnnotations*. The *updateshapes* method deletes old highlights from the moving average chart. Then, it adds a new object in the *shapes* array to highlight the intersection. If there is no intersection, *null* is added to the array to keep. track of the number of highlights on the chart. The *updateAnnotations* method deletes old annotations from the. moving average chart. Then, it calculates the slopes of the short and long term moving averages and checks if the intersection if a golden or death cross. If the intersection is a golden cross, the annotation will say *Buy* and if it is a death cross, the annotation will say *Sell.* finally, the functions updates the contents of the *annotations* array to add the annotation to the chart.

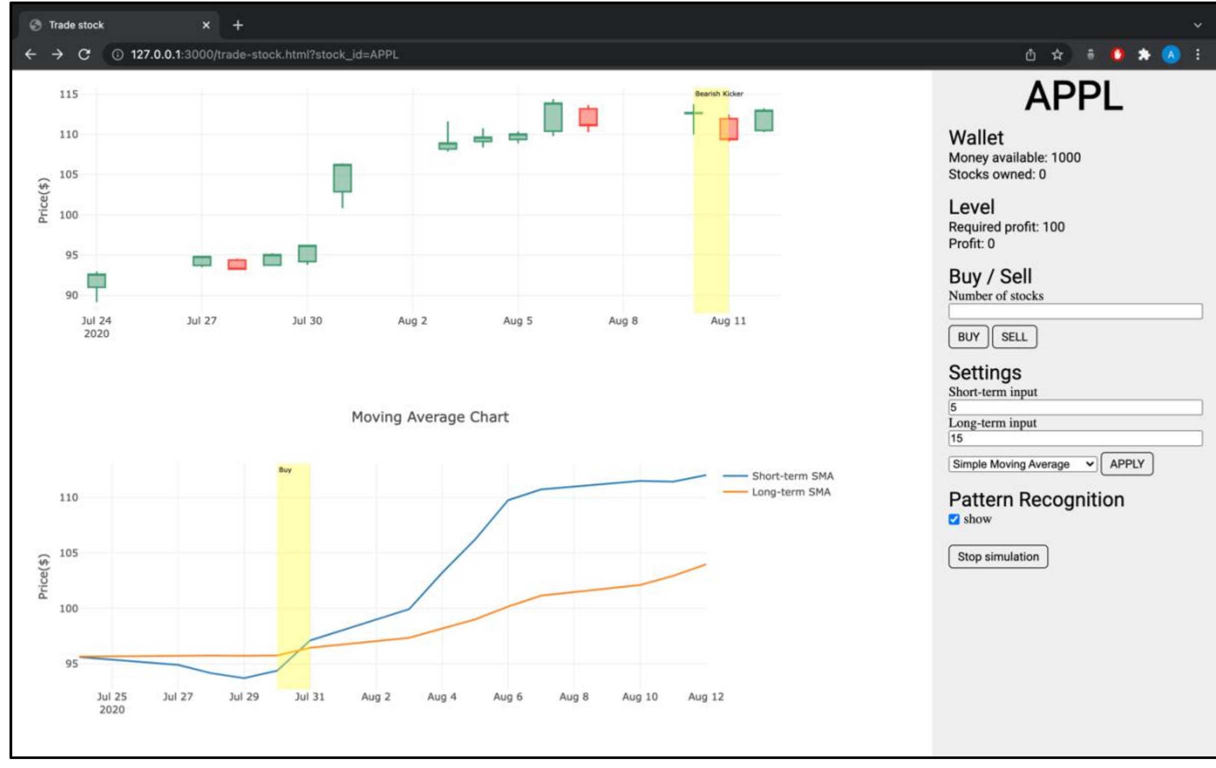
In the *app.js* file, the program checks if there are at least two data points on the moving average chart, as it is impossible to draw a line with just one point. Then, I update the *curr* and *prev* attributes of the *point* object by calling *updatePrev* and *updateCurr* methods. After that, I check if the lines intersect by calling a *updateIntersect* method. Finally, I update the *shapes* and *annotations* arrays to highglight and annotate intersections on the moving average chart.

Testing

As it can be seen on the moving average chart, the program has highglighted the region of intersection and anntated the intersection as ‘Sell’, because the slope of the long-term average is greater than the slope of the short-term average:



Also, when there is an intersection and the slope of the short-term average is greater than the long-term average, the program highlights the intersection and annotated it as 'Buy':



Reflection

In the *doIntersect* function I used a mathematical formula and could not come up with reasonable variables names, so the code for that function might not be readable to other developers. Therefore, I am planning to add a few comments explaining the formula.

Milestone №8 – allow the user to trade the chosen stock

In this milestone, I will implement buying and selling the chosen stock by the user.

Code listing

message.js

```
export let message = {
  message: '',
  displayMessage: function() {
    document.getElementById('error-message').textContent = this.message
  },
  notEnoughFunds: function() {
    this.message = 'Not enough funds.'
    this.displayMessage()
  },
  notEnoughStocks: function() {
    this.message = 'Not enough stocks.'
    this.displayMessage()
  },
  incorrectInput: function() {
    this.message = 'Must be an integer greater than 0.'
    this.displayMessage()
  },
  removeErrorMessage: function() {
    this.message = ''
    this.displayMessage()
  }
}
```

trade.js

```
import { stock } from "./stock.js"
import { wallet } from './wallet.js'
import { message } from './message.js'

export let trade = {
  number_of_stocks: 0,
  total_price: 0,
  getNumberOfStocks: function() {
    this.number_of_stocks = Number(document.getElementById('number-of-stocks').value)
  },
  isValidNumber: function() {
    return (this.number_of_stocks > 0 && Number.isInteger(this.number_of_stocks))
  },
  getTotalPrice: function() {
    this.total_price = this.number_of_stocks * stock.price
  },
  buyStock: function() {
    this.getNumberOfStocks()
    if (!this.isValidNumber()) return message.incorrectInput()
    this.getTotalPrice()

    if (this.total_price < wallet.money_available) {
```

```
      wallet.owned_stocks += this.number_of_stocks
      wallet.money_available -= this.total_price
      message.removeErrorMessage()
    }
    else {
      message.notEnoughFunds()
    }
    wallet.update()
  },
  sellStock: function() {
    this.getNumberOfStocks()
    if (!this.isValidNumber()) return message.incorrectInput()
    this.getTotalPrice()

    if (wallet.owned_stocks >= this.number_of_stocks) {
      wallet.owned_stocks -= this.number_of_stocks
      wallet.money_available += this.total_price
      message.removeErrorMessage()
    }
    else {
      message.notEnoughStocks()
    }
    wallet.update()
  }
}
```

app.js

```
...
import { trade } from './modules/trade.js'

(async () => {
  ...
  // event listeners
  document.getElementById('buy-btn').addEventListener('click', () => { trade.buyStock() })
  document.getElementById('sell-btn').addEventListener('click', () => { trade.sellStock() })

  let interval = setInterval(() => {
    ...
    // update profit
    wallet.getProfit()
    wallet.displayProfit()
    ...
  }, interval_constant)
})()
```

Explanation of the code

In the *message.js* file, I created several methods for displaying error messages to the screen. I created an attribute called *message* which holds the text of the message. The *displayMessage* method selects an HTML element and sets its *textContent* property to be the same as the value of the *message* attribute. Other methods change the value of the *message* attribute and call the *displayMessage* method.

In the *trade.js* file, I created two attributes: *number_of_stocks* and *total_price*. The *getNumberOfStocks* method gets the value from the input field and stores it in the *number_of_stocks* attribute. The *isValidNumber*

method checks if the entered number is in valid format, returns *true* if this is the case and *false* otherwise. The *getTotalPrice* method calculates the total price of the transaction and stores it in the *total_price* attribute.

The *buyStock* method gets the number of stocks as the input from the user. If the input is invalid the error message 'Must be an integer greater than 0' is displayed. If the input is valid, the total price of the transaction is calculated. If the total price is less than or equal to the money available, the transaction takes place:

- *owned_stocks* is increased by the number of stocks inputted by the user
- *money_available* is decreased by the total price of the transaction
- error message is removed

If the total price is more than the money available, the error message 'Not enough funds' is displayed. Finally, the wallet is updated at the end of the method.

The *sellStock* method gets the number of stocks as the input from the user. If the input is invalid the error message 'Must be an integer greater than 0' is displayed. If the input is valid, the total price of the transaction is calculated. If the number of stocks inputted by the user is less than or equal to the *owned_stocks*, the transaction takes place:

- *owned_stocks* is increased by the number of stocks inputted by the user
- *money_available* is decreased by the total price of the transaction
- error message is removed

If the number of stocks inputted by the user is greater than the *owned_stocks*, the error message 'Not enough stocks' is displayed. Finally, the wallet is updated at the end of the method.

In the *app.js* file, I added two event listeners to the buttons named 'buy' and 'sell'. When the 'buy' button is clicked, the *buyStock* method is called. If the 'sell' button is clicked, the *sellStock* method is called. Finally, inside the interval, I added two lines of code to get the current profit and display it to the screen.

Testing

Firstly, I tried buying too many stocks. The program should output 'Not enough funds':

Wallet
Money available: 1000
Stocks owned: 0

Level
Required profit: 100
Profit: 0

Buy / Sell
Number of stocks
100
BUY SELL
Not enough funds.

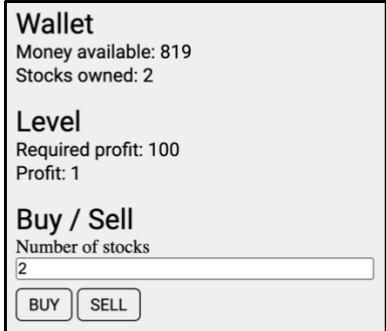
Secondly, I tried buying a fraction of a stock. The program should output 'Must be an integer greater than 0':

Wallet
Money available: 1000
Stocks owned: 0

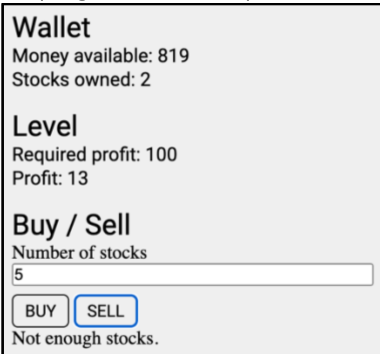
Level
Required profit: 100
Profit: 0

Buy / Sell
Number of stocks
2.5
BUY SELL
Must be an integer greater than 0.

Thirdly, I tried inputting a valid number and buying the stock. The program should allow it and update the wallet:



Then, I tried selling too many stocks. The program should ouput 'Not enough stocks':



Finally, I tried inputting a valid number and selling the stock. The program should allow it and update the wallet:

Reflection

I did not design how error messages will be shown in the Deisgn section, but I managed to implement the feature. Also, I had to change the algorithms for buying and selling a stock to include validation check and getting the input from the user.

Milestone №9 - allow the user to hide/show candlestick patterns

In this milestone, I will write code that will allow the user to hide and show candlestick patterns

Code listing

pattern.js

```
...
import { candlestick } from '../charts/candlestick.js'
```

```
export let pattern = {
  ...
  toggleShow: function() {
    this.show = !this.show
    candlestick.layout.shapes = []
    candlestick.layout.annotations = []
  }
}
```

app.js

```
...

(async () => {
  ...
  document.getElementById('pattern-recognition-checkbox').addEventListener('click', () => {
pattern.toggleShow() })
  ...
}
```

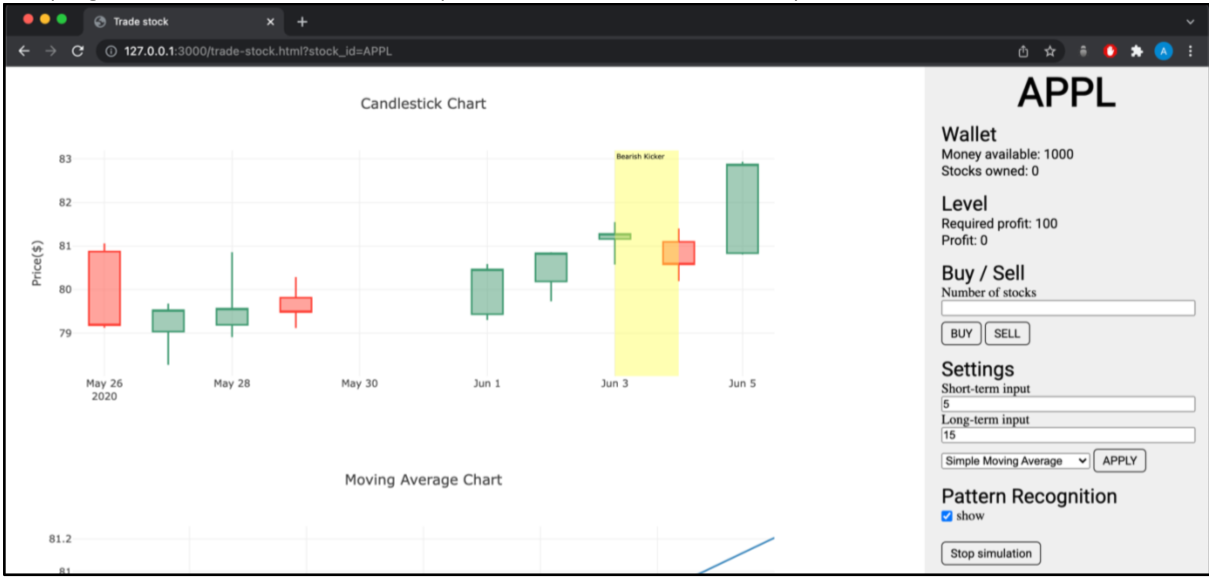
Explanation of the code

Firstly, I created a new method for the *pattern* object called *toggleShow.* This method changes the attribute called *show* to its opposite value as it is a boolean data type. Also, this method removes all objects from *shapes* and *annotations* arrays of the *candlestick* object.

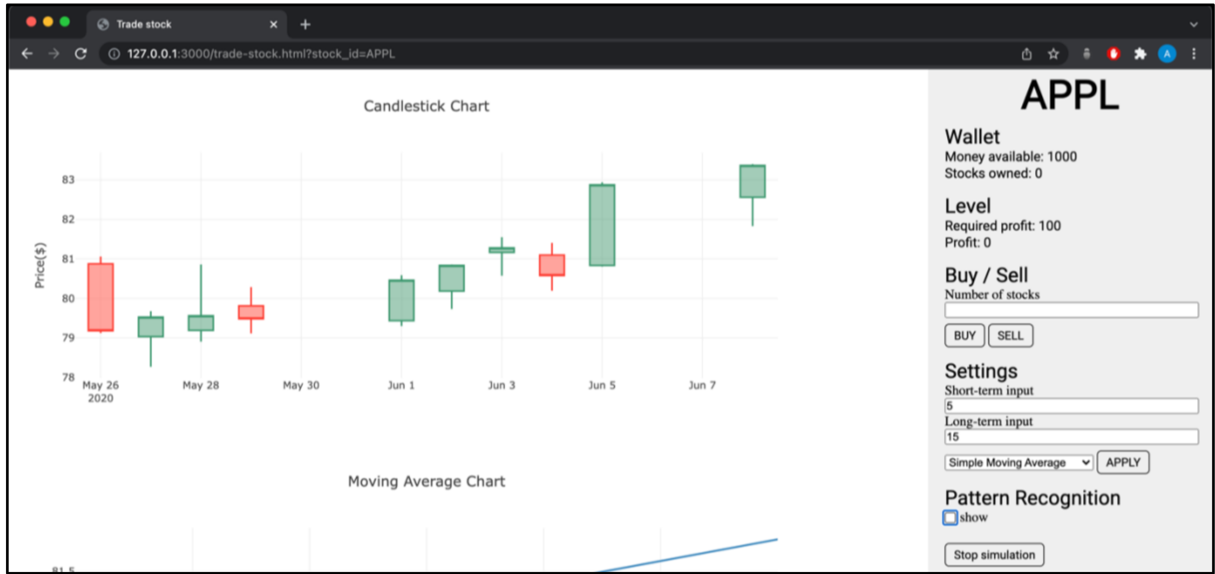
Secondly, in the *app.js* file, I add an event listener to an element with an id of *pattern-recognition-checkbox* so that when this element is clicked, the method *toggleShow* is called.

Testing

The program has found a Bearish Kicker pattern on the 3rd – 4th of January and the checkbox is checked:



Then, I clicked on the checkbox so its not checked and there were no patterns on the 3rd – 4th of January:



After some time I made sure that no further patterns are highlighted and annotated on the chart when the checkbox is not checked. Then, I clicked on the checkbox again and the program continued finding, highlighting and annotating candlestick patterns on the chart.

Reflection

As planned in the Design section, I made use of the attribute *show,* which has a boolean data type, to show and hide candlestick patterns. The only thing I did not think of in the Design section was to delete the old highlights and annotations from the *shapes* and *annotations* arrays.

Milestone №10 - allow the user to stop the simulation and view results

In this milestone, I will wrote code to allow the user to stop the simulation and view their results

Code listing

result.html

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Results</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <h1>Results</h1>

    <h2>Overall profit:</h2>
    <h3 id="results-profit"></h3>

    <h2>Required profit:</h2>
    <h3 id="results-required-profit"></h3>
```

```
    <h2>Level passed:</h2>
    <h3 id="results-level-passed"></h3>

    <button onclick="window.open('index.html')">Play again</button>
  </body>

  <script type="text/javascript">
    (function() {
      let queryString = window.location.search
      let urlParams = new URLSearchParams(queryString)

      let profit = urlParams.get('profit')
      let required_profit = urlParams.get('required_profit')
      let isPassed = urlParams.get('isPassed')

      document.getElementById('results-profit').textContent = profit
      document.getElementById('results-required-profit').textContent = required_profit
      document.getElementById('results-level-passed').textContent = isPassed
    })()
  </script>

</html>
```

result.js

```
import { wallet } from './wallet.js'
import { level } from './level.js'

export function stopSimulation() {
  let isPassed = wallet.profit >= level.required_profit

  window.open(`
    result.html?profit=${wallet.profit}&required_profit=${level.required_profit}&isPassed=${isPassed}
  `, '_self')
}
```

app.js

```
...
import { stopSimulation } from './modules/result.js'

(async () => {
  ...
 document.getElementById('stop-simulation-btn').addEventListener('click', () => { stopSimulation() })
  ...
```

Explanation of the code

Firstly, I created a new HTML page called *result.html*. In there, I created some empty elements where I will insert the results of the simulation. Also, I inserted some JavaScript code there. The code just gets parameters from the url of the browser and displays them to the screen, so that the user could see their results.

Secondly, in the *result.js* file, I created a function *stopSimulation*. I created a variable *isPassed*, which is *true* when profit is greater than the required profit and *false* otherwise. Then, it opens the *result.html* page with several parameters in the url, such as *profit, required_profit, isPassed.*

Finally, in the *app.js* file, I added an event listener to the button ‘stop simulation’ which would trigger the function *stopSimulation* when clicked.

Testing

After I played the simulation and have completed the level, I pressed the ‘stop simulation’ button and got the following result:

Results

Overall profit:

183

Required profit:

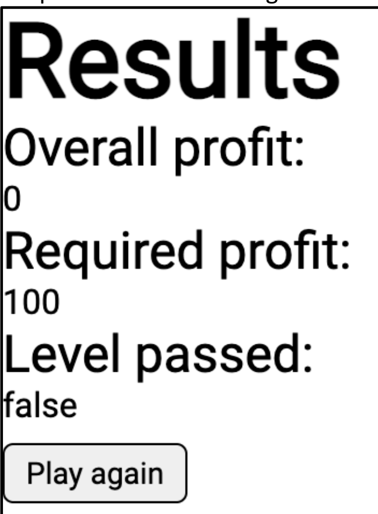
100

Level passed:

true

Play again

Then I intentionally made zero profit and pressed the button again:



Also, when I press the ‘play again’ button, I am being redirected to the starting screen of the simulation.

Reflection

In the design section I thought that the program would display the results on the main screen of the simulation. However, I decided that there is too much information on that screen already. Therefore, I decided to create a different page for this purpose.

Milestone №11 – allow the user to change the type of moving average and its parameters

In this milestone, I will write code to allow the user to change the type of moving average and its parameters.

Code listing

moving-average.js

```
...
import  { updatePlot } from './plot.js'

export let moving_average = {
  ...
  changeSettings: function() {
    let short_term_input = Number(document.getElementById('short-term-input').value)
    let long_term_input = Number(document.getElementById('long-term-input').value)
    let type_input = document.getElementById('options').value

    if (short_term_input > 15 || short_term_input < 1 || !Number.isInteger(short_term_input)) {
      alert('Short-term input must be an integer between 1 and 14')
      return this.resetInputs()
    }
    else if (long_term_input > 30 || long_term_input < 15 || !Number.isInteger(long_term_input)) {
      alert('Long-term input must be an integer between 15 and 30')
      return this.resetInputs()
    }

    this.data[0].x = []
    this.data[0].y = []
    this.data[0].name = 'Short-term ' + this.type.toUpperCase()

    this.data[1].x = []
    this.data[1].y = []
    this.data[1].name = 'Long-term ' + this.type.toUpperCase()

    this.layout.shapes = []
    this.layout.annotations = []

    updatePlot(this)
  },
  resetInputs: function() {
    document.getElementById('short-term-input').value = this.day_short
    document.getElementById('long-term-input').value = this.day_long
    document.getElementById('options').value = this.type
  }
}
```

app.js

```
...
(async () => {
  ...
  document.getElementById('change-settings-btn').addEventListener('click', () => {
moving_average.changeSettings() })


  ...
})()
```

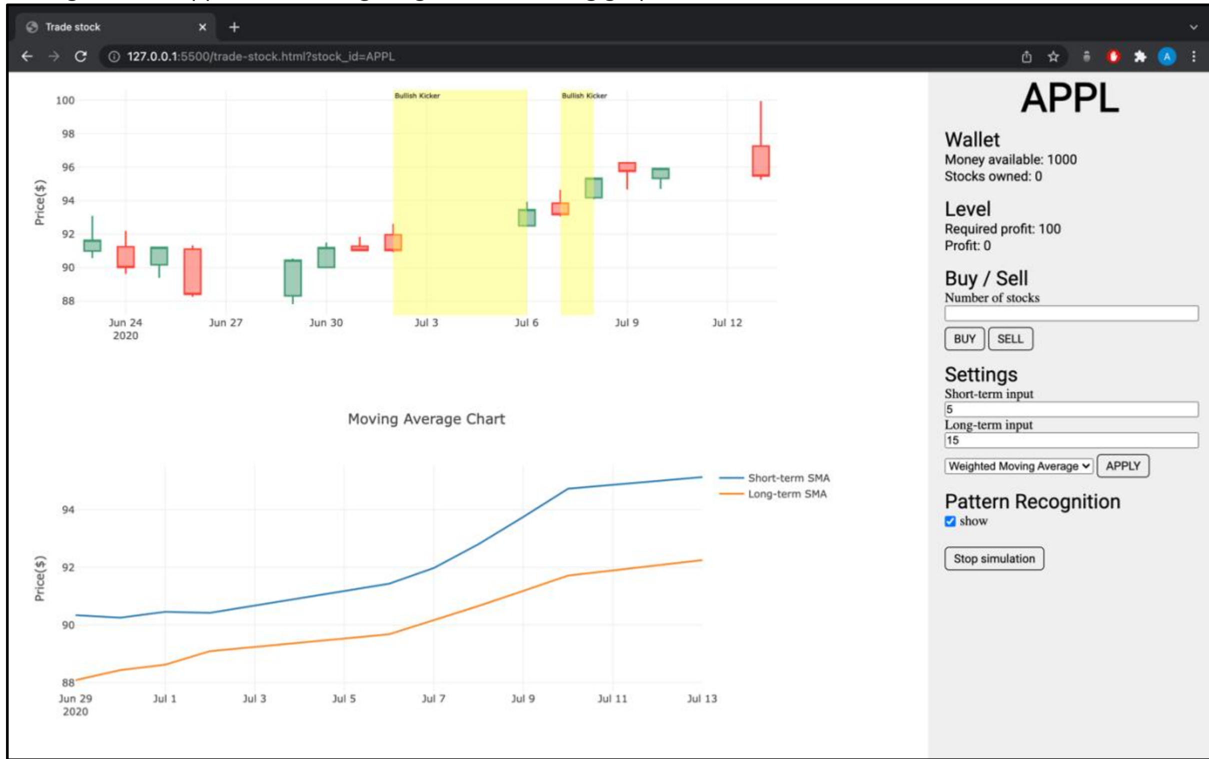
Explanation of the code

In the *moving-average.js* file, I added two methods to the *moving_average* object. The *resetInputs* method sets the value of input fields related to the moving average to their current values. This method is needed to reset the inputs in case the user enters invalid data. The *changeSettings* method gets the values of the input fields and checks them for validity. If at least one of the inputs is not valid, the inputs are reseted to what they were and the user will see an error message. Then, the method deletes all the data points from the moving average chart and renames the labels of lines if the user has changed the type of the moving average. After that, the method deletes all highlights and annotations from the moving average chart and upates the chart.

In the *app.js* file, I added an event listeners to the 'apply' button, which would call the *changeSettings* method and if all the inputs are valid, the changes will apply.

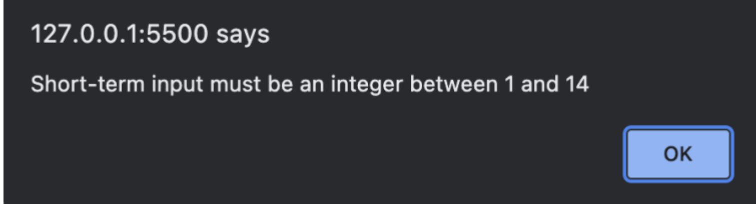
Testing

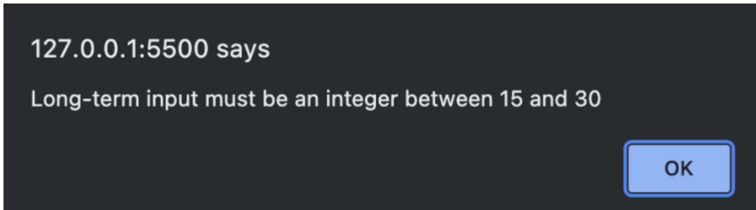
Firstly, I tried changing the type of the moving average from simple moving average to weighted moving average. After I applied the changes I got the following graphs:



I checked that the values on the moving average chart are calculated with the weighted moving average formula by manually calculating several points. However, the labels for the graphs did not change.

Secondly, I changed inputs to invalid data and got the following error messages:





Therefore, data validation works correctly. Also, as I checked, inputs are being reseted to their previous valid values.

Then, I tried inputting valid values and not changing the type of the moving average. Having manually calculated several points, I came to conclusion that it worked correctly. The same procedure was repeated with changing the type of the moving average and changing parameters to valid, but different values. Therefore, everything except for changing the labels of the graphs when changing the type of the moving average works correctly.

Reflection

I will need to review the code and find out why the labels for the graphs would not change when the user changes the type of the moving average. The next step is to meet with the stakeholders and continue improving and debugging the program.

Meeting with stakeholders

Having completed all the milestones, even though with a few bugs, I decided to meet my stakeholders once again to verify that the simulation looks similar to what they expected. Firstly, they said that the rate of updating the charts has improved, as I increased the *interval_constant* by 1 second. Secondly, they liked that error messages are displayed whenever incorrect data is inputted. Finally, they said that it would be great if I could fix the bug where changing the type of the moving average does not change the labels on the graphs. Other than that, they agreed to test my program by checking that all success criteria requirements are met.

Evaluation

Post-development testing

Link to the simulation
https://pedantic-thompson-a9ffed.netlify.app/index.html

Testing table

№	Test and input	Expected output	Result
1	Enter the url in the search bar of the browser	The starting screen of the simulation can be seen	Pass
2	Press the 'start trading' button	The user is redirected to a different page where they can see a selection of levels	Pass
3	Select a level	The user is redirected to the main screen of the simulation. They can see the charts being updated, virtual wallet, level requirements, boxes for entering parameters and changing settings and all the elements of the interface	Pass
4	Candlestick patterns are highlighted and annotated	The user can see highlighted and annotated candlestick patterns	Pass
5	Moving average intersections are highlighted and annotated	The user can see highglighted and annotated intersections of moving averages	Pass
6	Try buying the stock with the following cases: 1. valid number of stocks and the user has enough funds 2. valid number of stocks and the user does not have enough funds 3. invalid number of stocks and the user has enough funds 4. invalid number of stocks and the user does not have enough funds	1. money available decreases by the total price and owned stocks increases by the number of stocks 2. error message 'not enough funds' 3. error message 'must be an integer greater than zero' 4. error message 'must be an integer greater than zero', if resolved then 'not enough funds'	Pass
7	Try selling the stock with the following cases: 1. valid number of stocks and the user has enough stocks 2. valid number of stocks and the user does not have enough stocks 3. invalid number of stocks and the user has enough stocks 4. invalid number of stocks and the user does not have enough stocks	1. money available is increased by the total price and owned stocks is decreased by the number of stocks 2. error message 'not enough stocks' 3. error message 'must be an integer greater than zero' 4. error message 'must be an integer greater than zero', if resolved then 'not enough stocks'	Pass
8	Try changing moving average parameters with the following cases: 1. Only type 2. Only short-term average 3. Only long-term average 4. Both averages 5. Type and both averages	In all cases, the data from the moving average chart must be deleted. 1. Values are calculated using a formula for the chosen type of the moving average 2. Short-term average values will be calculated using more or less data points depending on the input 3. Long-term average will be calculated using more or less data points depending on the input	Pass

		4. Both averages will be calculated using more or less data points depending on the inputs 5. Both averages will be calculated using a formula for the chosen type of the moving avreage and both will be calculated using more or less data points depending on the inputs	
9	Check the pattern recognition checkbox	The user can see highlighted and annotated candlestick patterns	Pass
10	Uncheck the pattern recognition checkbox	The user no longer can see highlighted and annotated candlestick patterns	Pass
11	Profit is calculated correctly	1. Take the current price of the stock 2. Multiply it by the number of stocks owned 3. Add money available 4. Subtract initial capital 5. Round to the whole number 6. Check is the value the program calculates is the same	Pass
12	Press the 'stop simulation' button	The user is redirected to a different page and they can see their results calculated correctly. If their profit is less than the required profit, the level is not passed. If their profit is greater than the required profit, the level is passed.	Pass

Evidence for post-development testing

Test №1

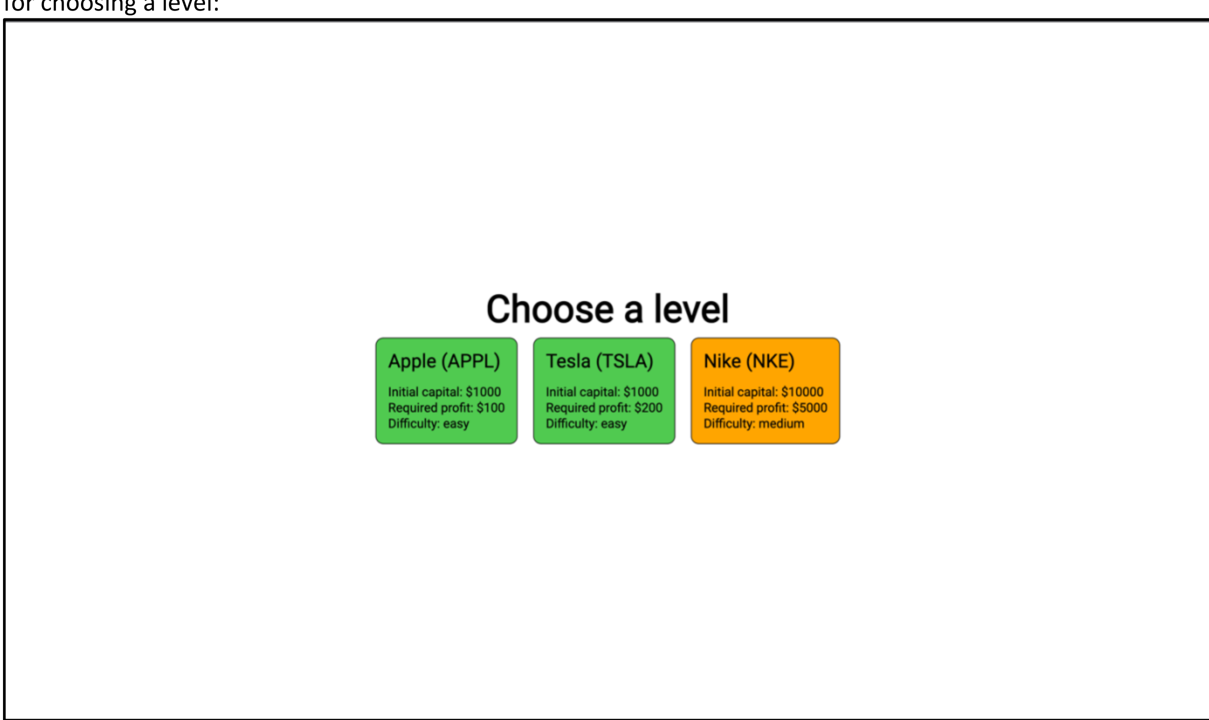
When the user enters the link to the simulation into the browser search bar and presses enter, they are redirected to the following page, which is the starting page of the simulation:

Trading simulation

start trading

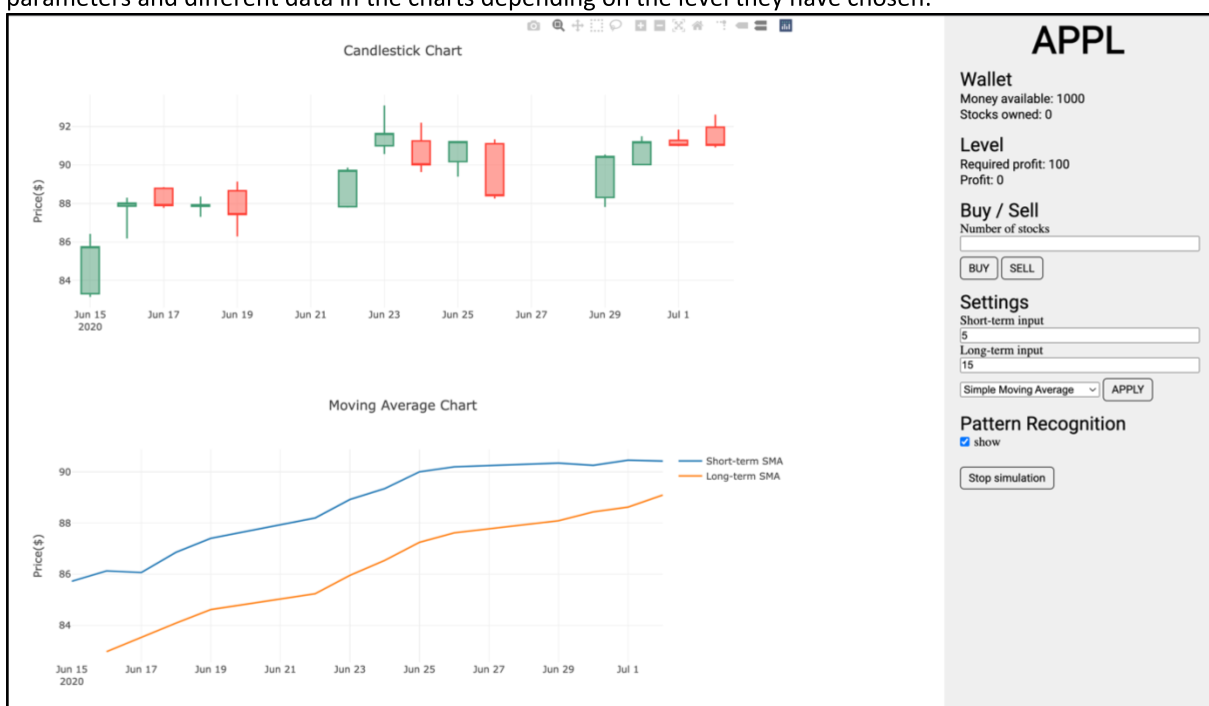
Test №2

When the user presses the 'start trading' button, they are redirected to the following page, which is the page for choosing a level:



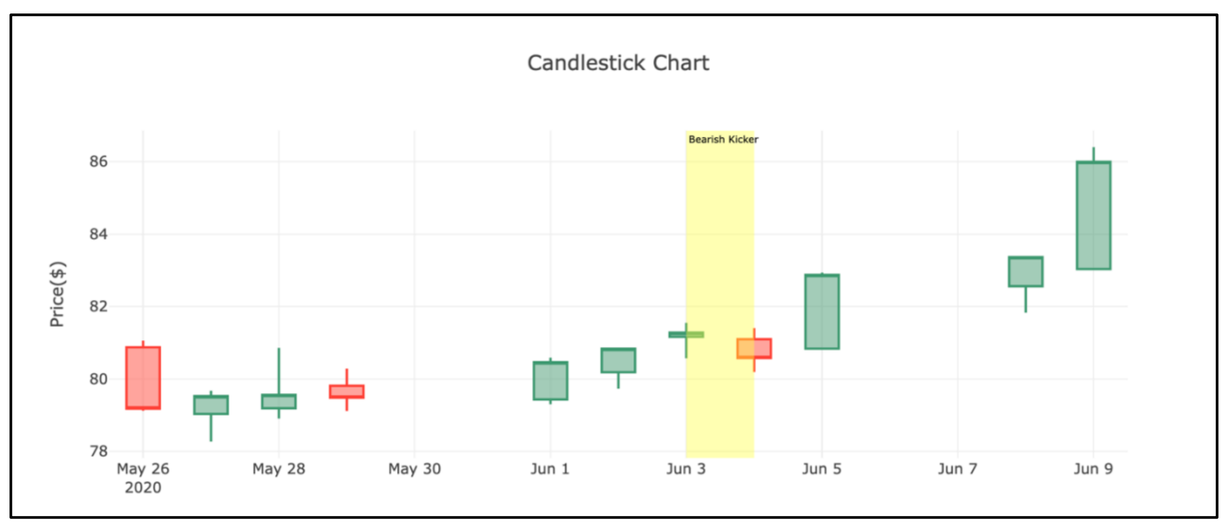
Test №3

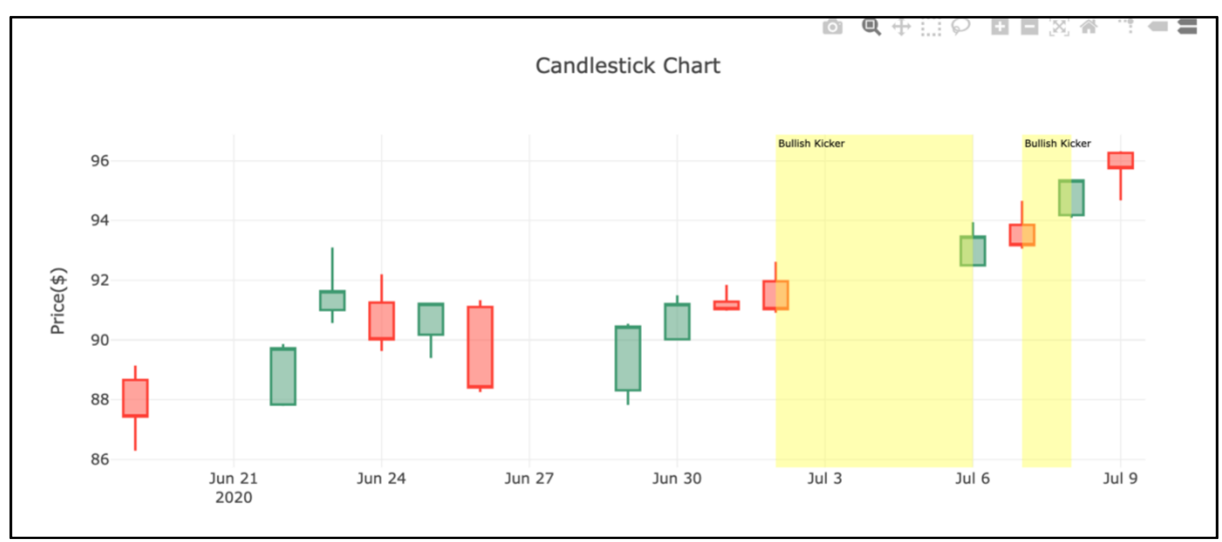
When the user selects a level from the selection of levels, they are shown different information such as initial parameters and different data in the charts depending on the level they have chosen:

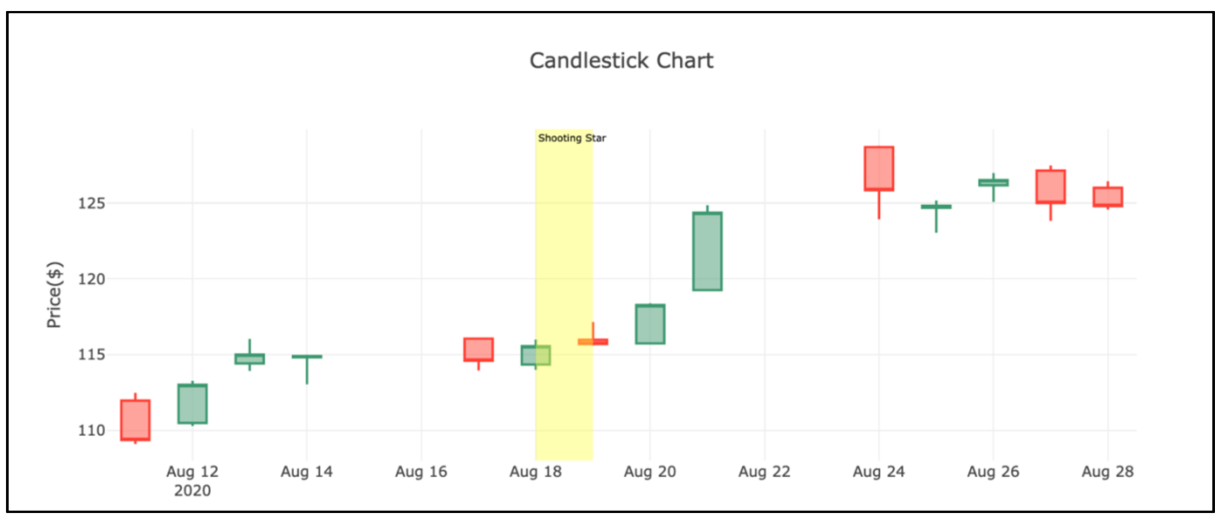


Test №4

As it can be seen from the following screenshots, the program identifies candlestick patterns and annotates them correctly using a set of predefined formulas, which represent the general shapes of those candlestick patterns:

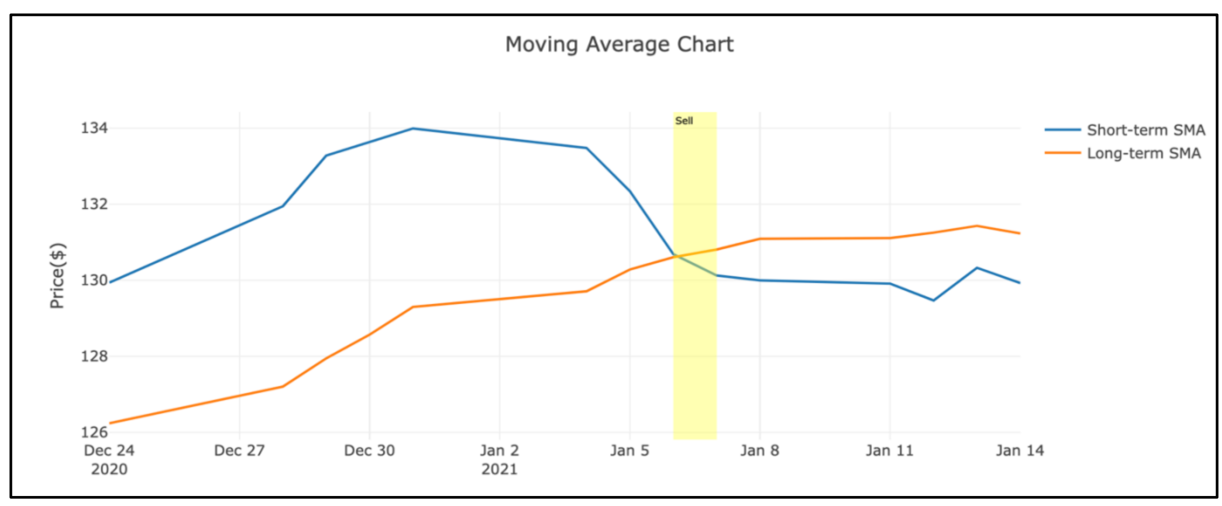


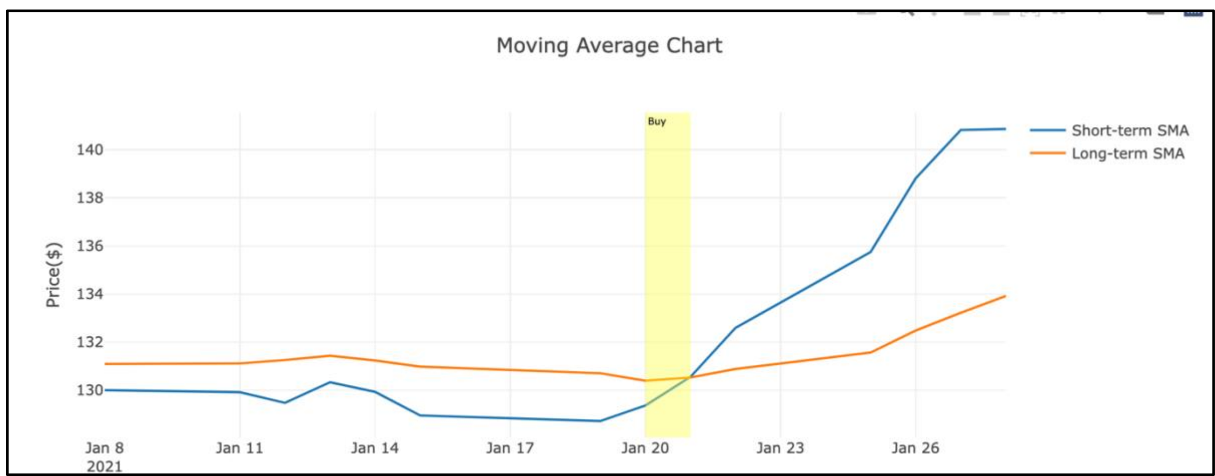




Test №5

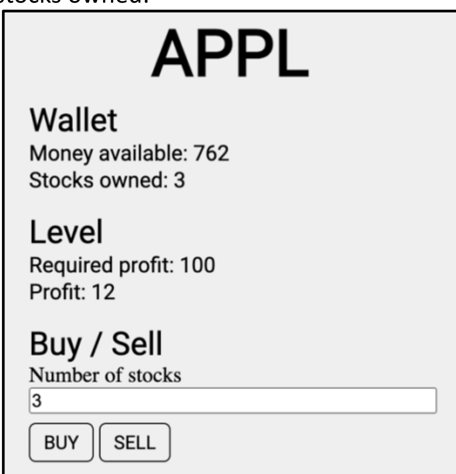
As it can be seen from the following screenshots, the program identifies intersections of moving averages and correctly annotated them as ‘buy’ when the slope of the short-term moving average is greater than the slope of the long-term moving average and ‘sell’ when the slope of the long-term moving average is greater than the slope of the short-term moving average:



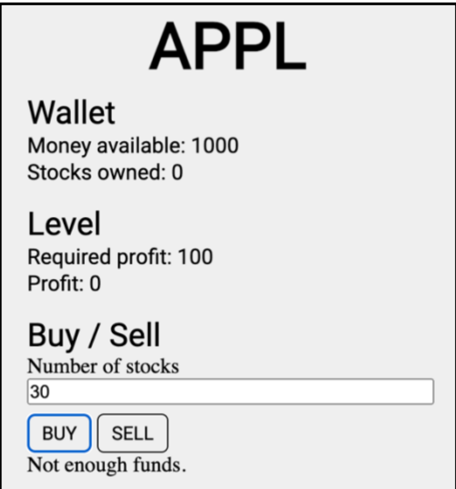


Test №6

When the user enters a valid number of stocks and has enough funds to buy them, the program allows it and updates the money available and stocks owned:

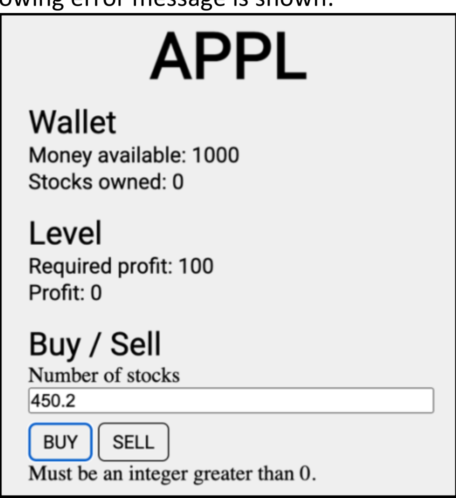


When the user enters a valid number of stocks, but does not have enough funds to buy them, the transaction is declined and the following error message is shown:



When the user enters an invalid number of stocks and has enough funds to buy them, the transaction is declined and the following error message is shown:

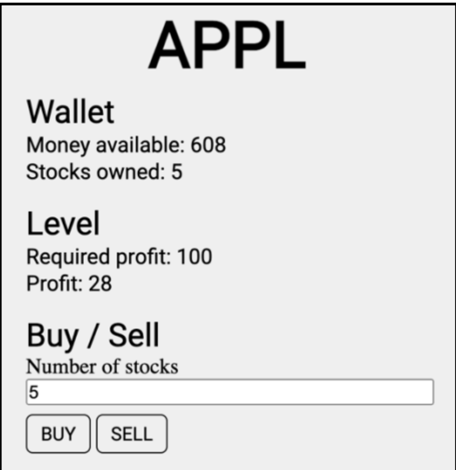
When the user enters an invalid number of stocks and does not have enough funds to buy them, the transaction is declined and the following error message is shown:



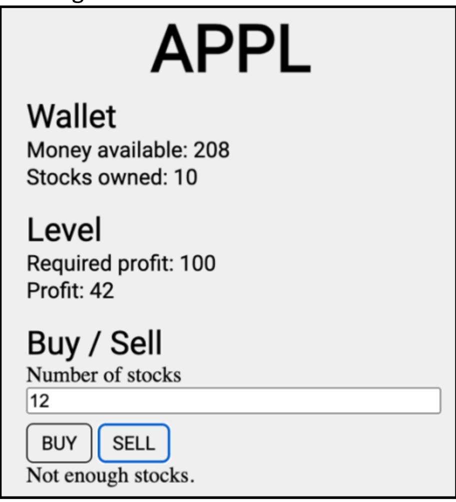
Test №7

In all of the tests below the user starts with 10 stocks and the number in the box represents the number of stocks the user wants to sell.

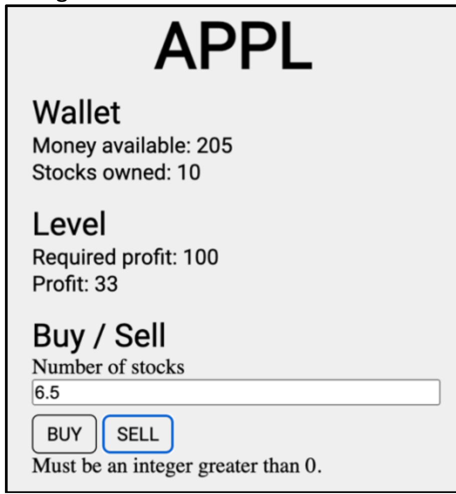
When the user enters a valid number of stocks and has enough stocks to sell them, the program allows it and updates the money available and stocks owned:



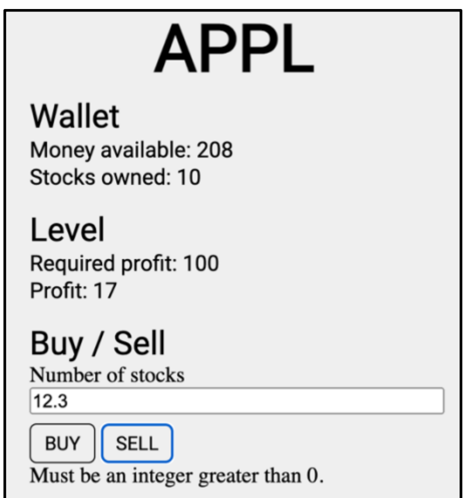
When the user enters a valid number of stocks, but does not have enough stocks to sell them, the transaction is declined and the following error message is shown:



When the user enters an invalid number of stocks and has enough stocks to sell them, the transaction is declined and the following error message is shown:



When the user enters an invalid number of stocks and does not have enough stocks to sell them, the transaction is declined and the following error message is shown:

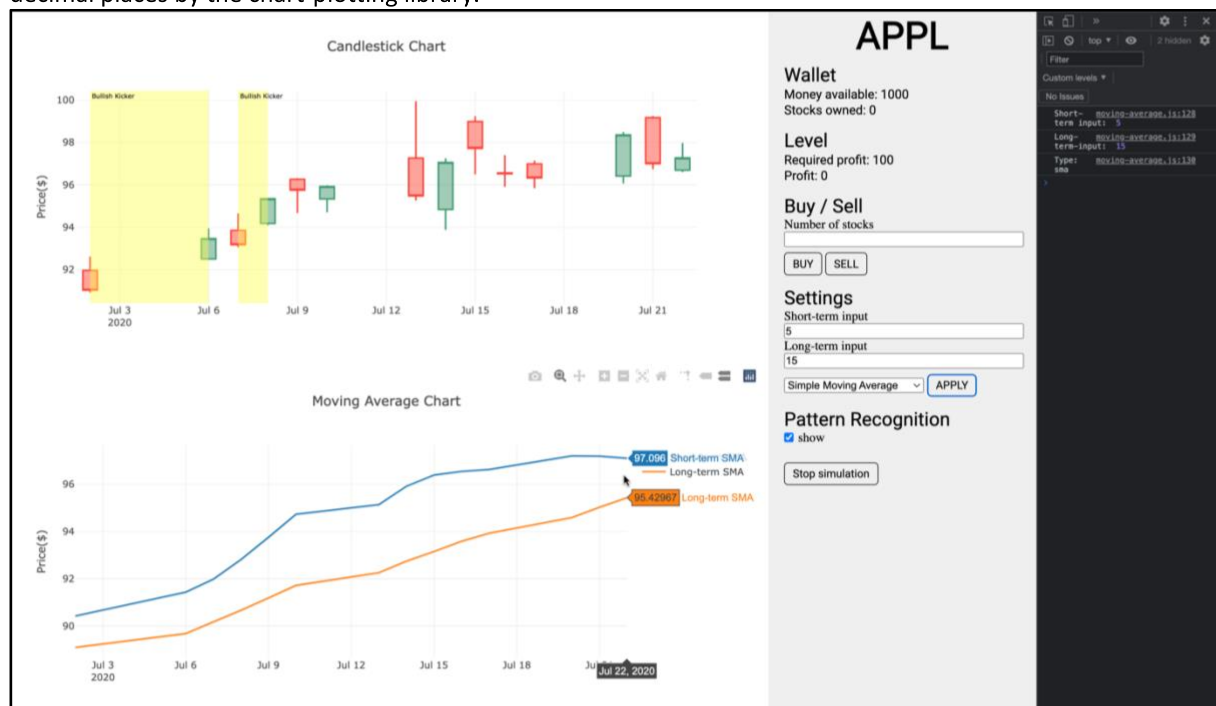


Test №8

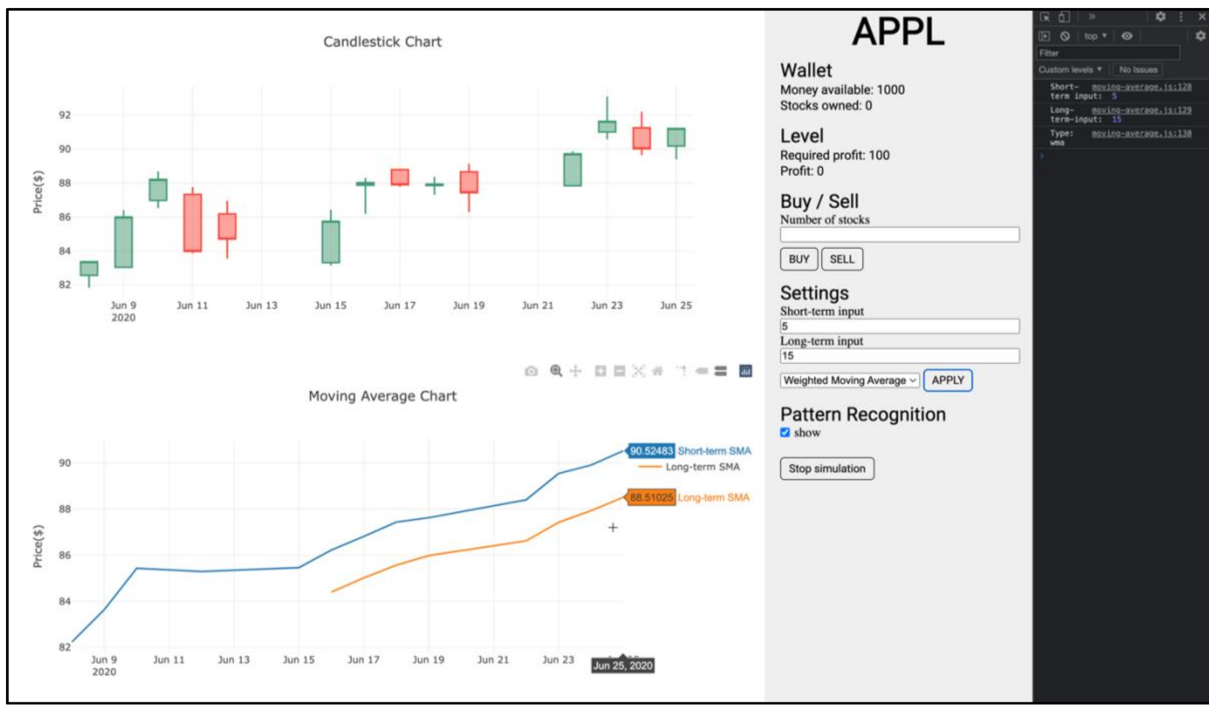
The default values for the moving average parameters, which I change during the tests:
type: 'sma'
short-term-input: 5
long-term-input: 15

Changing the type of the moving average

In the following screenshot, a simple moving average is selected. The last five closing prices were: 97.272499, 97, 98.357498, 96.327499, 96.522499. According to the simple moving average formula, the 5-day average should be 97.095999. The value shown on the graph is 97.096, which is the same value rounded to three decimal places by the chart-plotting library.



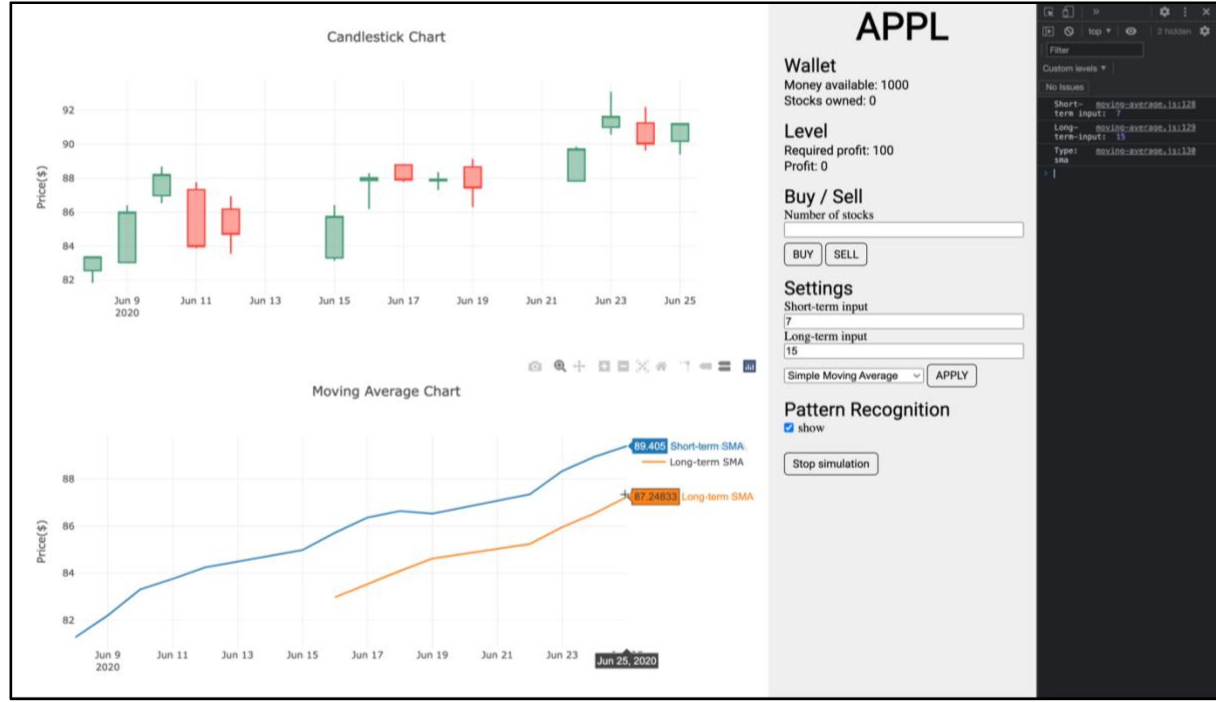
In the following screenshot, a weighted moving average is selected. The last five closing prices were: 91.209999, 90.014999, 91.6325, 89.717499, 87.43. According to the weighted moving average formula, the 5-day average should be 90.5248326. The value shown on the graph is 90.52483, which is the same value rounded to five decimal places by the chart-plotting library.



Also, the values of parameters that can be changed in the settings are shown in the console. When the type of the moving average is changed to weighted, the *type* variable changes its value to 'wma', which stands for weighted moving average. This test concludes that changing the type of the moving average works correctly and values are being calculated using a correct formula. However, I still have not fixed the bug that prevents labels from changing to 'Short-term WMA' and 'Long-term WMA'.

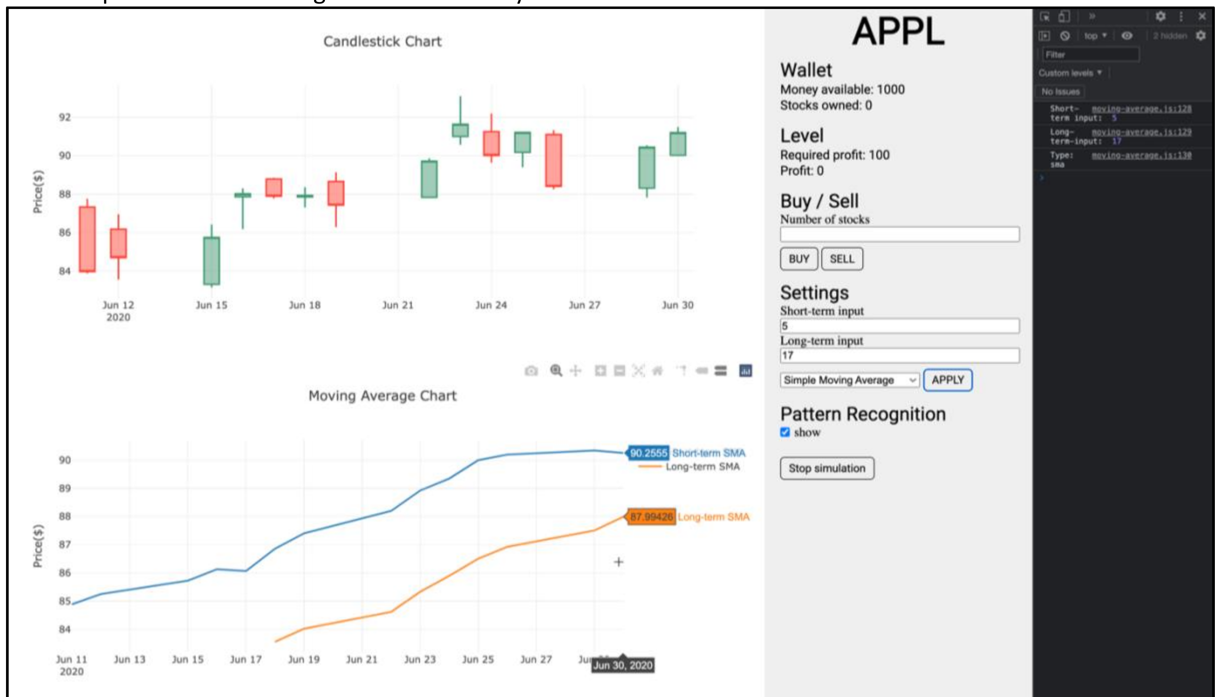
Changing the short-term average

In this test, I changed the *short-term-input* to 7. According to the 7-day simple moving average formula, the value should be 89.405. The value shown on the graph is 89.405, which is the same value. This test concludes that changing the short-term average with all other parameters unchanged works correctly.



Changing the long-term average

In this test, I changed the *long-term-input* to 17. According to the 17-day simple moving average formula, the value should be 87.99426365. The value shown on the graph is 87.99426, which is the same value rounded to five decimal places by the chart-plotting library. This test concludes that changing the long-term average with all other parameters unchanged works correctly.

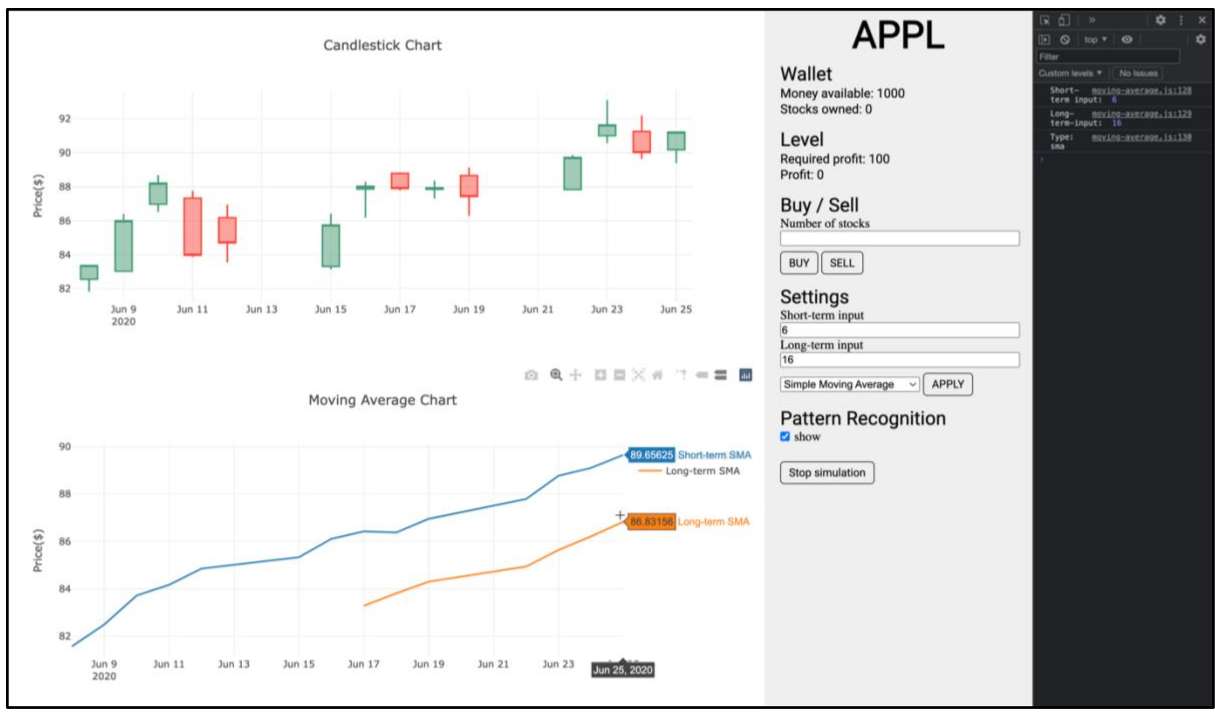


Changing both averages at the same time

In this test, I changed the *short-term-input* to 6. According to the 6-day simple moving average formula, the value should be 89.65625. The value shown on the graph is 89.65625, which is the same value.

Also, I changed the *long-term-input* to 16. According to the 16-day simple moving average formula, the value should be 86.83156163. The value shown on the graph is 86.83156, which is the same value rounded to five decimal places by the chart-plotting library.

This test concludes that changing the short-term and long-term averages at the same time with all other parameters unchanged works correctly.



Changing both averages and the type at the same time

In this test, I changed the *short-term-input* to 8. According to the 8-day weighted moving average formula, the value should be 90.53319. The value shown on the graph is 90.53319, which is the same value.

Also, I changed the *long-term-input* to 18. According to the 18-day weighted moving average formula, the value should be 89.32534. The value shown on the graph is 89.32534, which is the same value.

Also, I changed the *type* of the moving average to ‘wma’, which is shown in the console.

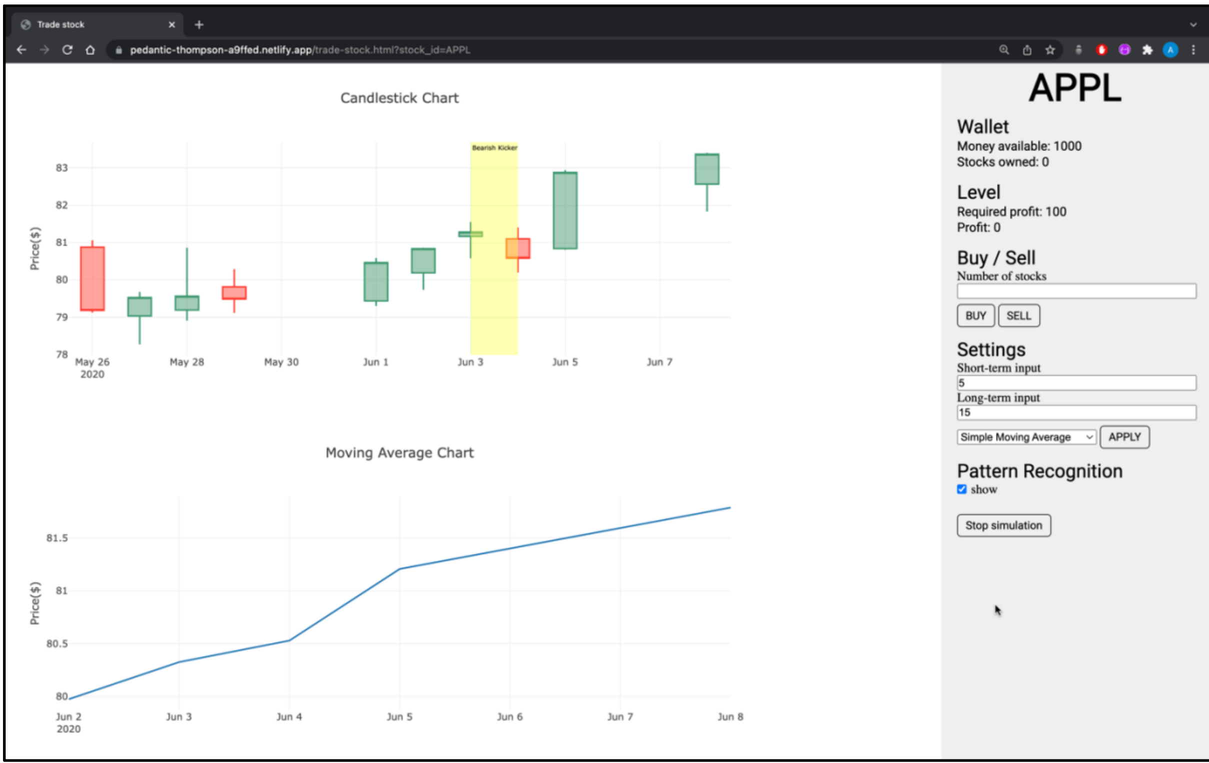
This test concludes that changing all the parameters at the same time works correctly.

All of the above tests conclude that by changing the type of the moving average, the program uses a correct formula for calculating values. Also, by changing short-term and long-term inputs, the program takes it into account and uses more data points to calculate a correct value, as it should be. Finally, changing all parameters of the moving average at the same time works as intended as well.

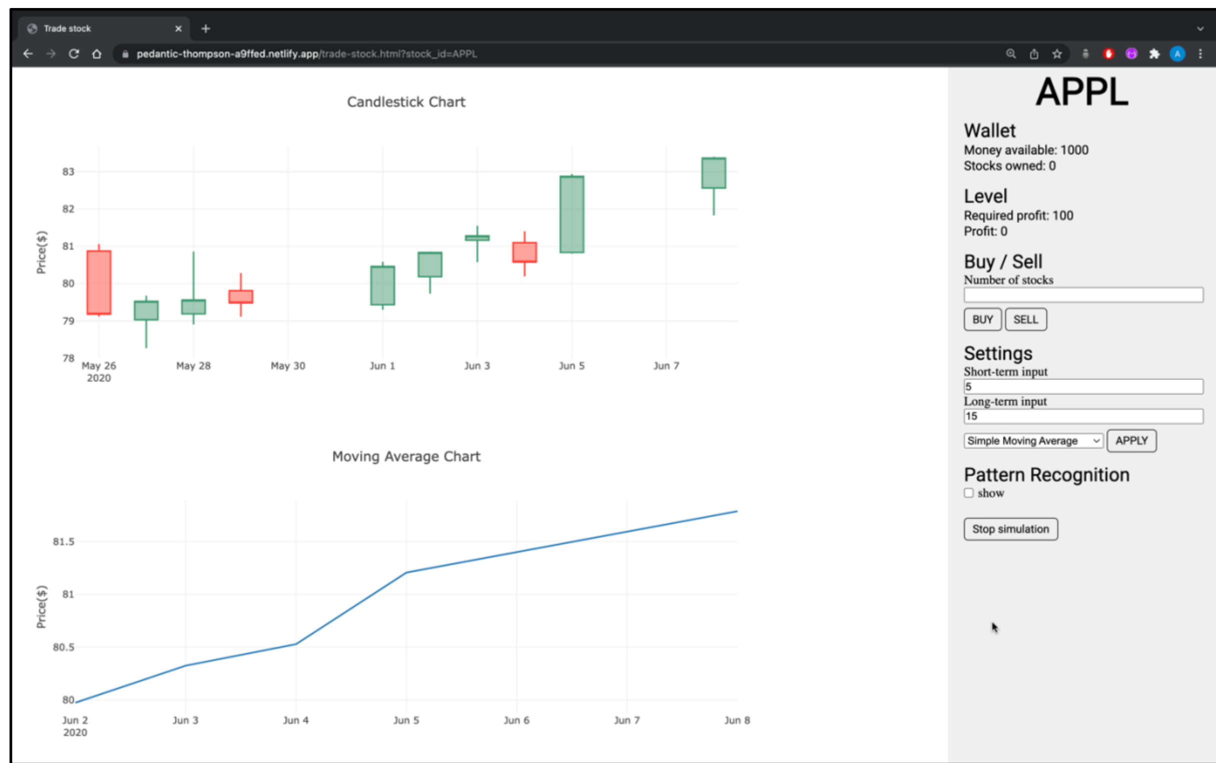
Test №9 and test №10

In order to test this feature, I will firstly keep the checkbox checked to find a candlestick pattern and take a screenshot of it. Then, I will reload the simulation, uncheck the checkbox and take a screenshot of the same place where the candlestick pattern used to be. In order for the simulation to pass this test, the candlestick from the first screenshot should not be visible in the second screenshot.

First screenshot:



Second screenshot:



As it can be seen from the screenshots above, the Bearish Kicker candlestick pattern that was found between the 3rd and 4th of January, disappeared when I unchecked the pattern recognition checkbox.

Test №11

Formula for calculating profit is the following:
*profit = money available + number of stocks owned * current price of the stock – initial capital*

Formula for calculating money available:
*money available = money available – number of stocks * current price of the stock*

The initial capital is $1000 and I had $1000 at the start of the test.

In order to test this feature, I bought 10 stocks at $80.835.

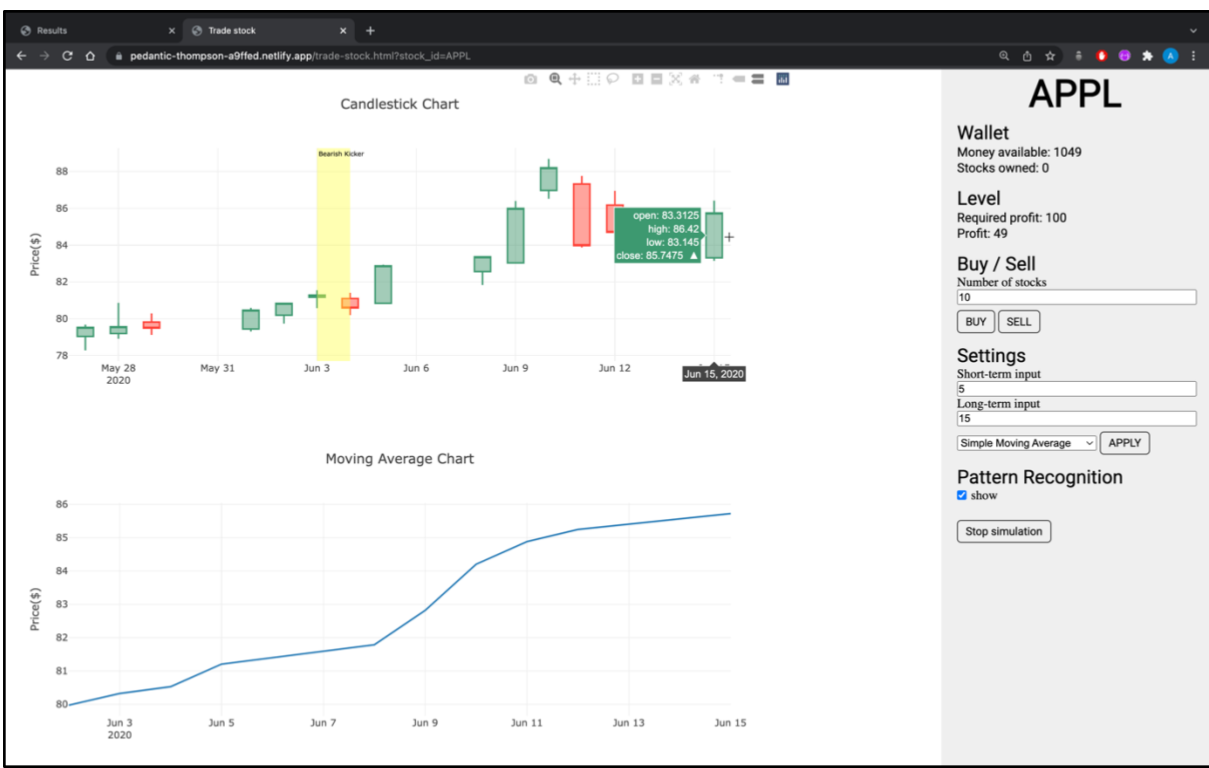
Money available became:
*1000 – 10 * 80.835 = 191.65 (192 rounded to the whole number)*

Stocks owned became 10.

Some time later, the stock price became 85.7475.

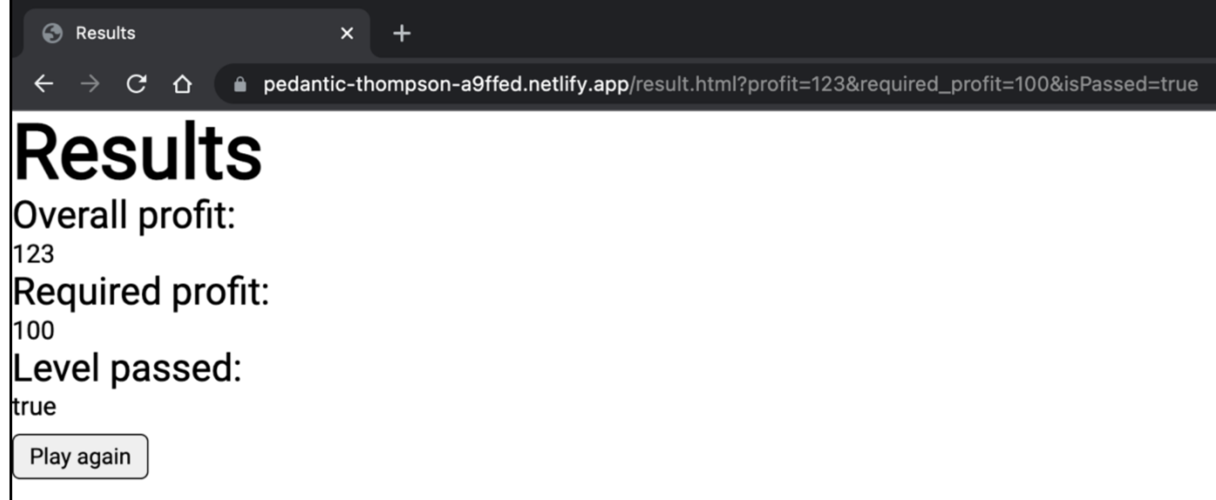
The current profit, according to the formula, should be:
*191.65 + 10 * 85.7475 – 1000 = 49.125 (49 rounded to the whole number)*

As shown in the following picture, the profit is calculated correctly by the program and is equal to $49 rounded to the whole number:



Test №12

When the user made a profit that is more than the required profit and presses the 'stop simulation' button, they are redirected to the following screen and all required data from the simulation is passed to this page through the url as parameters:



However, if the user did not manage to make enough profit to pass the level, they will see the following screen. Again, all the data from the simulation is passed to this page through the url as parameters:

Results

pedantic-thompson-a9ffed.netlify.app/result.html?profit=20&required_profit=100&isPassed=false

Results

Overall profit:

20

Required profit:

100

Level passed:

false

Play again

Success criteria evaluation

№	Requirement	Success	Evaluation
1	The starting screen is shown when the user enters the address of the website into a browser	Fully met	When the user enters the address of the website into a browser, they can see the starting screen of the simulation.
2	A screen with a selection of levels is shown after the user presses a button on the starting screen	Fully met	When the user presses the 'start trading' button on the starting screen of the simulation, they are redirected to another page with a selection of levels.
3	The program retrieves data for the chosen stock from a server	Fully met	When the user chooses a level, the stock data for that level is imported from the corresponding JSON file and is stored in the *stock* object.
4	A screen with virtual wallet and level requirements is shown	Fully met	When the user chooses a level, they are redirected to the main screen of the simulation and they can see various information about their virtual wallet and level requirements.
5	Display a candlestick chart for the chosen stock	Fully met	On the main screen of the simulation, the user can see the candlestick chart being constantly updated with an interval of 2 seconds.
6	Allow the user to interact with the candlestick chart by hovering over with their mouse	Fully met	I managed to find a graph-plotting library with such functionality and when the user hovers over a candle with their mouse, they can see the full information about that candle.
7	Display a moving average chart for the chosen stock	Fully met	On the main screen of the simulation, the user can see the moving average chart with two line graphs of moving averages, which is constantly updated every 2 seconds.
8	Highlight and annotate crossovers between moving averages	Fully met	The program is constantly looking for intersections of two moving averages and when it finds an intersection, it highlights and annotates its name on the moving average chart.
9	Highlight and annotate candlestick patterns on the candlestick chart	Fully met	The program is constantly looking for candlestick patterns in the data and when a pattern is found, it highlights and annotates its name on the candlestick chart.

10	Ability to change parameters and moving average type during the simulation	Partially met	The user can change the parameters of the moving average during the simulation and the graph is updated when the user presses the button 'apply'. However, the labels for the graphs would not change when the user changes the type of the moving average.
11	Ability to buy and sell the chosen stock	Fully met	The user can enter the number of stocks they want to buy or sell. The number entered is validated and if the user has a sufficient amount of money or stocks, the transaction takes place. Otherwise, they see an error message saying what the problem is.
12	Ability to stop simulation at any time	Fully met	There is a button 'stop simulation' on the main screen of the simulation and the user can press it any time during the simulation. When they press the button, they are redirected to a different page, where they can view their results.
13	View results at the end of the simulation	Fully met	When the user presses the 'stop simulation' button, they can view their results and the program will tell the user if they have succcessfully completed the chosen level.

Usability features

One example of a usability feature in my simulation is the use of h1-h6 HTML paragraphs. It is easy for a user to find information on the screen because of the use of headings. For example, when they want to find how much money is available, they can instantaneously see 'Wallet' heading and then look for the required information in that section.

Another example of a usability feature is that charts are big enough for the user to see all the information they need and the charts are labeled at the top. It makes it clear which chart the user is using and they are less likely to miss a feature of a graph, as the charts are big enough to spot every single detail.

Also, all the input boxes have a label on the top and some of them have a placeholder, which is text inside the text box which disappears as soon as you start typing in it. In my opinion, it is a good usability feature, as it is very clear what information the user has to type in the input box. Furthermore, if they typed anything invalid, they get an error message that explains what is wrong with their input.

In addition, highglighting candlestick patterns and intersections of moving averages will not allow the user to miss a pattern or an intersection due to the contrast of the colours. I intentionally chose yellow colour, as this colour is very easy to spot on white background.

Finally, responsive design is one of the features that could improve the usability of my simulation. It would allow the users to use my simulation on mobile devices. Undoubtedly, responsive design would be a huge improvement of my simulation as more users would be able to use it, but it would take way too much time planning the new design and writing CSS code. Therefore, I decided not to add a respoonsive design and consider it a limitation of my program.

Limitations and improvements

Firstly, keeping the old data points of the candlestick and moving average charts would be a great improvement of my simulation, as users would be able to scroll the graphs horizontally. That would allow them to view the whole picture of the graphs, find even more patterns that my simulation is not capable of finding and earn even more profit. Therefore, the ability to view old parts of the charts is a limitation of my simulation, which could be added if I had more time.

Secondly, I planned on replacing built-in JavaScript arrays for highlights and annotations with abstract data structures, such as queues. Queues would have solved the problem more efficiently and I would have written less code. Even though I studied how to implement queues in JavaScript and wrote the code for it in one of the milestones, it did not work. Since then, I thought that I would have another go after I complete the whole program. However, I did not have enough time for that, as I had to complete the Evaluation section of the coursework. Nevertheless, the simulation works correctly with the use of built-in arrays, but the implementation of queues would have been an improvement of efficiency of the program.

Also, throughout the coursework, I did not think what would happen if the program runs out of stock data. Therefore, as an improvement, I decided to add some code to stop the program when this happens. I added the following code in the *app.js* file:

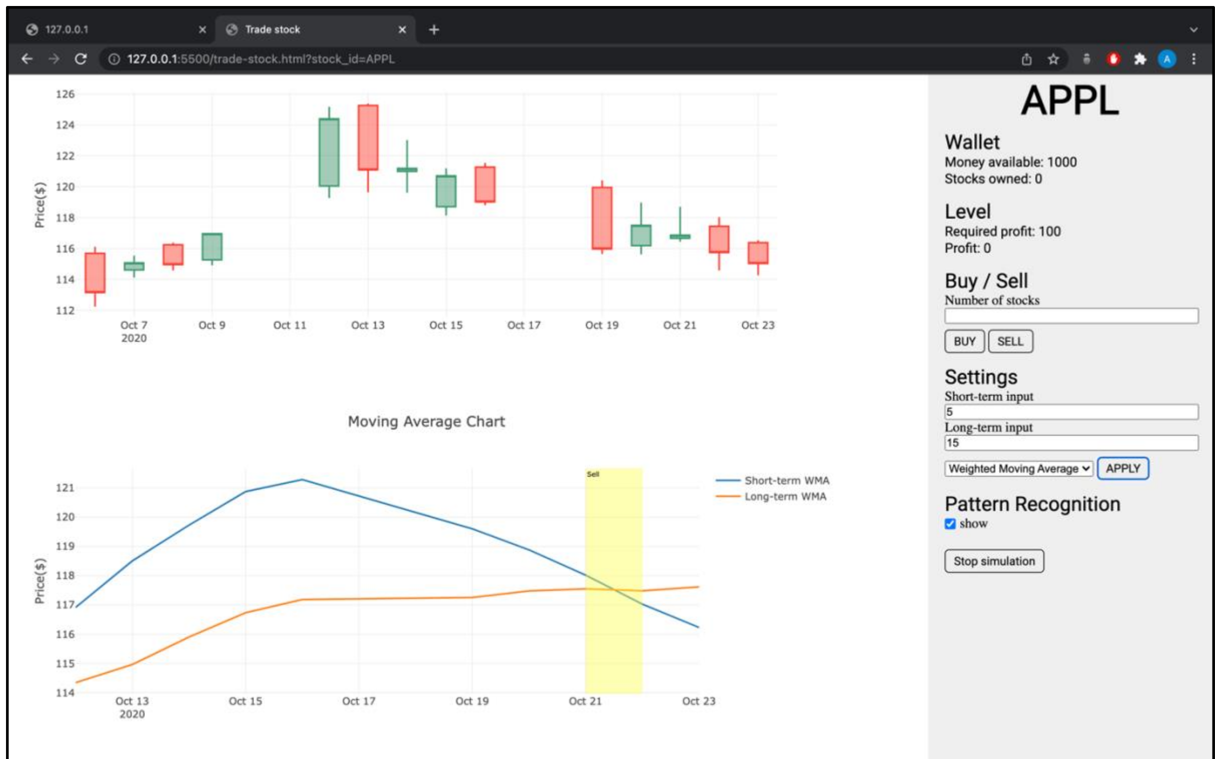
```
(async () => {
  ...
  let interval = setInterval(() => {
    ...
    if (stock.count === stock.data.length - 1) clearInterval(interval)
    ...
  }, interval_constant)
}
```

As the result of the above code, when the program runs out of stock data, the code inside the interval will stop running, so there will be no more updates to the charts, current profit and the counter. The event listeners and all buttons will still work, so the user will be able to stop the simulation by pressing the 'stop simulation' button.

In addition, I was not able to solve the problem with renaming labels for the graphs in the Development section. It turned out to have a simple solution which I did not notice. When the user changed the type of the moving average, I set the labels for the graphs to their previous values instead of the new values inputted by the user. I changed the following lines of code to solve this problem:

```
export let moving_average = {
  ...
  changeSettings: function() {
    ...
    this.data[0].name = 'Short-term ' + type_input.toUpperCase()
    ...
    this.data[1].name = 'Long-term ' + type_input.toUpperCase()
    ...
  }
}
```

I tested the improvement by changing the type of the moving average to weighted moving average. As the result, the labels for the graphs changed to WMA as well:



Finally, my program is not capable of finding every single possible candlestick pattern that exists in the data. Writing all the functions for finding various candlestick patterns would take way too much time and effort. I decided that my simulation would look for three main candlestick patterns that consist of two candles, as the simulation is aimed at beginners and they do not need to know all possible candlestick patterns. Therefore, the number of candlestick patterns detected by the program is another limitation of my program, which could have been a great improvement if I had more time for the Development stage.

Maintenance

After the user has completed the three levels, they might want to practise more. In order to add a new level to the simulation, all I will need to do is download stock data from Yahoo Finance, convert it to JSON format and add it to the *./json* directory. After that, I will need to add properties of the new level, such as *id, name, initial capital, required profit, difficulty* to the *levels* array. After that the new level should be available to the users. Easy addition of new levels was one of the key factors I kept in mind while developing the program.

Other than adding new levels, users may want the program to detect more candlestick patterns. They can do it by writing a function for detecting a pattern in the *./modules/pattern-recognition/candlestick-patterns.js* file. Then, they would need to slightly modify an *if statement* by adding one extra condition in the *updateName* method in the *./modules/pattern-recognition/pattern.js* file. After that, the added function should be used by the program to detect even more patterns. The only limitation is that patterns must consist of two candles.

Code listings

index.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Trading simulation</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <div id="index">
      <h1>Trading simulation</h1>
      <button id="start-trading-btn">start trading</button>
    </div>
  </body>

  <script>
    document.getElementById('start-trading-btn').addEventListener('click', () => { window.open('choose-
level.html', '_self') })
  </script>

</html>
```

choose-level.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Choose level</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <div id="choose-level">
      <h1>Choose a level</h1>
      <div id="levels"></div>
    </div>
  </body>

  <script type="module">
    import { levels } from './modules/level.js'

    (function() {
      let levels_html = ``
```

```
      for (let i = 0; i < levels.length; i++) {
        let name = levels[i].name
        let id = levels[i].id
        let initial_capital = levels[i].initial_capital
        let required_profit = levels[i].required_profit
        let difficulty = levels[i].difficulty

        let level_html = `
          <a class="level ${difficulty}" id="${id}">
            <h2>${name} (${id})</h2>
            <h3>Initial capital: $${initial_capital}</h3>
            <h3>Required profit: $${required_profit}</h3>
            <h3>Difficulty: ${difficulty}</h3>
          </a>`

        levels_html = levels_html + level_html
      }

      document.getElementById('levels').innerHTML = levels_html
    })()

    let links = document.getElementsByTagName('a')

    for (let i = 0; i < links.length; i++) {
      links[i].addEventListener('click', () => { window.open(`trade-stock.html?stock_id=${links[i].id}`,
'_self') })
    }
  </script>

</html>
```

trade-stock.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Trade stock</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <div id="trade-stock">
      <div id="charts">
        <div id="candlestick-chart"></div>
        <div id="moving-average-chart"></div>
      </div>

      <div id="information">
        <h1 id="stock-id"></h1>
```

```
        <div>
          <h2>Wallet</h2>
          <h3 id="money-available"></h3>
          <h3 id="owned-stocks"></h3>
        </div>

        <div>
          <h2>Level</h2>
          <h3 id="required-profit"></h3>
          <h3 id="profit"></h3>
        </div>

        <div>
          <h2>Buy / Sell</h2>
          <div class="input-container">
            <label for="number-of-stocks">Number of stocks</label>
            <input type="number" id="number-of-stocks" required>
          </div>
          <button id="buy-btn">BUY</button>
          <button id="sell-btn">SELL</button>
          <p id="error-message"></p>
        </div>

        <div>
          <h2>Settings</h2>
          <div class="input-container">
            <label for="short-term-input">Short-term input</label>
            <input type="number" id="short-term-input" value="5">
          </div>
          <div class="input-container">
            <label for="long-term-input">Long-term input</label>
            <input type="number" id="long-term-input" value="15">
          </div>
          <select id="options">
            <option value="sma">Simple Moving Average</option>
            <option value="wma">Weighted Moving Average</option>
          </select>
          <button id="change-settings-btn">APPLY</button>
        </div>

        <div>
          <h2>Pattern Recognition</h2>
          <div class="checkbox-container">
            <input type="checkbox" id="pattern-recognition-checkbox" checked>
            <label for="pattern-recognition-checkbox">show</label>
          </div>
        </div>

        <div>
          <button id="stop-simulation-btn">Stop simulation</button>
        </div>
      </div>
    </div>
  </body>
```

```
  <script src='https://cdn.plot.ly/plotly-latest.min.js'></script>
  <script src="app.js" type="module"></script>

</html>
```

result.html

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Results</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="./style.css">
  </head>

  <body>
    <h1>Results</h1>

    <h2>Overall profit:</h2>
    <h3 id="results-profit"></h3>

    <h2>Required profit:</h2>
    <h3 id="results-required-profit"></h3>

    <h2>Level passed:</h2>
    <h3 id="results-level-passed"></h3>

    <button onclick="window.open('index.html')">Play again</button>
  </body>

  <script type="text/javascript">
    (function() {
      let queryString = window.location.search
      let urlParams = new URLSearchParams(queryString)

      let profit = urlParams.get('profit')
      let required_profit = urlParams.get('required_profit')
      let isPassed = urlParams.get('isPassed')

      document.getElementById('results-profit').textContent = profit
      document.getElementById('results-required-profit').textContent = required_profit
      document.getElementById('results-level-passed').textContent = isPassed
    })()
  </script>

</html>
```

style.css

```css
* {
  margin: 0;
  padding: 0;
}

h1 {
  font-family: 'Roboto';
  font-weight: 600;
  font-size: 48px;
  line-height: 52px;
  color: #000;
}

h2 {
  font-family: 'Roboto';
  font-weight: 400;
  font-size: 24px;
  line-height: 28px;
  color: #000;
}

h3 {
  font-family: 'Roboto';
  font-weight: 400;
  font-size: 16px;
  line-height: 20px;
  color: #000;
}

button {
  margin-top: 7px;
  padding: 5px 10px;
  border: 1px solid #000;
  border-radius: 5px;
  font-family: 'Roboto';
  font-size: 14px;
  font-weight: 400;
  color: #000;
}

button:hover {
  cursor: pointer;
}

/* INDEX */
#index {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);

  display: flex;
  flex-direction: column;
```

```
  align-items: center;
}

#index button {
  margin-top: 25px;
  padding: 10px 15px;
  font-size: 20px;
  background-color: rgb(80, 202, 80);
}

/* CHOOSE-LEVEL */
#choose-level {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);

  display: flex;
  flex-direction: column;
  align-items: center;
}

#levels {
  display: flex;
}

.level {
  margin: 10px;
  padding: 15px;
  border: 1px solid #000;
  border-radius: 10px;
  text-decoration: none;
  color: #000;
}

.level h2 {
  margin-bottom: 15px;
}

.level:hover {
  cursor: pointer;
}

.easy {
  background-color: rgb(80, 202, 80);
}

.medium {
  background-color: orange;
}

/* TRADE-STOCK */
#stock-id {
  margin-bottom: 10px;
```

```
  text-align: center;
}

#trade-stock {
  display: flex;
}

#charts {
  width: calc(100% - 350px);
  max-width: 1000px;
}

#information {
  padding: 5px 20px;
  position: fixed;
  top: 0;
  right: 0;
  width: 300px;
  height: 100vh;
  background-color: rgb(239, 239, 239);
}

#information > div {
  margin-bottom: 15px;
}

input {
  width: 100%;
}

.checkbox-container {
  display: flex;
  justify-content: flex-start;
  align-items: center;
}

#pattern-recognition-checkbox {
  margin-right: 5px;
  width: fit-content;
}
```

app.js

```
import { stock } from './modules/stock.js'
import { level } from './modules/level.js'
import { wallet } from './modules/wallet.js'
import { candlestick } from './modules/charts/candlestick.js'
import { moving_average } from './modules/charts/moving-average.js'
import { drawNewPlot, updatePlot } from './modules/charts/plot.js'
import { pattern } from './modules/pattern-recognition/pattern.js'
import { point } from './modules/intersection/point.js'
import { trade } from './modules/trade.js'
import { stopSimulation } from './modules/result.js'
```

```
(async () => {
  let interval_constant = 2000

  // stock.js
  stock.getId()
  stock.displayId()
  stock.getUrl()
  stock.getData()
    .then(() => { stock.getPrice() })

  // level.js
  level.getInitialCapital()
  level.getRequiredProfit()
  level.displayRequiredProfit()

  // wallet.js
  wallet.setMoneyAvailable()
  wallet.update()

  // charts
  drawNewPlot(candlestick)
  drawNewPlot(moving_average)

  // event listeners
  document.getElementById('buy-btn').addEventListener('click', () => { trade.buyStock() })
  document.getElementById('sell-btn').addEventListener('click', () => { trade.sellStock() })
  document.getElementById('change-settings-btn').addEventListener('click', () => {
moving_average.changeSettings() })
  document.getElementById('pattern-recognition-checkbox').addEventListener('click', () => {
pattern.toggleShow() })
  document.getElementById('stop-simulation-btn').addEventListener('click', () => { stopSimulation() })

  let interval = setInterval(() => {

    // clear interval
    if (stock.count === stock.data.length - 1) clearInterval(interval)

    // update stock price
    stock.getPrice()

    // update candlestick chart
    candlestick.extend()

    if (stock.count > 1) {
      pattern.updatePrev()
      pattern.updateCurr()
      pattern.updateName()

      if (pattern.show) {
        candlestick.updateShapes()
        candlestick.updateAnnotations()
      }
    }
```

```
    updatePlot(candlestick)

    // update moving average chart
    moving_average.extendShortMovingAverage()
    moving_average.extendLongMovingAverage()

    if (moving_average.data[1].x.length > 1) {
      point.updatePrev()
      point.updateCurr()
      point.updateIntersect()

      moving_average.updateShapes()
      moving_average.updateAnnotations()
    }

    updatePlot(moving_average)

    // update profit
    wallet.getProfit()
    wallet.displayProfit()

    // update stock count
    stock.incrementCount()

  }, interval_constant)
})()
```

modules/stock.js

```
export let stock = {
  id: '',
  url: '',
  data: [],
  price: 0,
  count: 0,
  getId: function() {
    let queryString = window.location.search
    let urlParams = new URLSearchParams(queryString)
    this.id = urlParams.get('stock_id')
  },
  displayId: function() {
    document.getElementById('stock-id').textContent = this.id
  },
  getUrl: function() {
    this.url = './json/' + this.id + '.json'
  },
  getData: async function() {
    let response = await fetch(stock.url)
    this.data = await response.json()
  },
  getPrice: function() {
    this.price = this.data[this.count].close
  },
  incrementCount: function() {
```

```
    this.count = this.count + 1
  }
}
```

modules/wallet.js

```
import { stock } from "./stock.js"
import { level } from "./level.js"

export let wallet = {
  money_available: 0,
  owned_stocks: 0,
  profit: 0,
  setMoneyAvailable: function() {
    this.money_available = level.initial_capital
  },
  displayMoneyAvailable: function() {
    document.getElementById('money-available').textContent = 'Money available: ' +
Math.round(this.money_available)
  },
  displayOwnedStocks: function() {
    document.getElementById('owned-stocks').textContent = 'Stocks owned: ' + this.owned_stocks
  },
  getProfit: function() {
    this.profit = Math.round(this.money_available + this.owned_stocks * stock.price - level.initial_capital)
  },
  displayProfit: function() {
    document.getElementById('profit').textContent = 'Profit: ' + this.profit
  },
  update: function() {
    this.displayMoneyAvailable()
    this.displayOwnedStocks()
    this.getProfit()
    this.displayProfit()
  }
}
```

modules/level.js

```
import { stock } from './stock.js'

export const levels = [
  {
    name: 'Apple',
    id: 'APPL',
    initial_capital: 1000,
    required_profit: 100,
    difficulty: 'easy'
  },
  {
    name: 'Tesla',
    id: 'TSLA',
    initial_capital: 1000,
```

```
    required_profit: 200,
    difficulty: 'easy'
  },
  {
    name: 'Nike',
    id: 'NKE',
    initial_capital: 10000,
    required_profit: 5000,
    difficulty: 'medium'
  }
]

export let level = {
  initial_capital: 0,
  required_profit: 0,
  getInitialCapital: function() {
    for (let i = 0; i < levels.length; i++) {
      if (stock.id === levels[i].id) {
        this.initial_capital = levels[i].initial_capital
      }
    }
  },
  getRequiredProfit: function() {
    for (let i = 0; i < levels.length; i++) {
      if (stock.id === levels[i].id) {
        this.required_profit = levels[i].required_profit
      }
    }
  },
  displayRequiredProfit: function() {
    document.getElementById('required-profit').textContent = 'Required profit: ' + this.required_profit
  }
}
```

modules/message.js

```
export let message = {
  message: '',
  displayMessage: function() {
    document.getElementById('error-message').textContent = this.message
  },
  notEnoughFunds: function() {
    this.message = 'Not enough funds.'
    this.displayMessage()
  },
  notEnoughStocks: function() {
    this.message = 'Not enough stocks.'
    this.displayMessage()
  },
  incorrectInput: function() {
    this.message = 'Must be an integer greater than 0.'
    this.displayMessage()
  },
  removeErrorMessage: function() {
```

```
    this.message = ''
    this.displayMessage()
  }
}
```

modules/trade.js

```
import { stock } from "./stock.js"
import { wallet } from './wallet.js'
import { message } from './message.js'

export let trade = {
  number_of_stocks: 0,
  total_price: 0,
  getNumberOfStocks: function() {
    this.number_of_stocks = Number(document.getElementById('number-of-stocks').value)
  },
  isValidNumber: function() {
    return (this.number_of_stocks > 0 && Number.isInteger(this.number_of_stocks))
  },
  getTotalPrice: function() {
    this.total_price = this.number_of_stocks * stock.price
  },
  buyStock: function() {
    this.getNumberOfStocks()
    if (!this.isValidNumber()) return message.incorrectInput()
    this.getTotalPrice()

    if (this.total_price < wallet.money_available) {
      wallet.owned_stocks += this.number_of_stocks
      wallet.money_available -= this.total_price
      message.removeErrorMessage()
    }
    else {
      message.notEnoughFunds()
    }
    wallet.update()
  },
  sellStock: function() {
    this.getNumberOfStocks()
    if (!this.isValidNumber()) return message.incorrectInput()
    this.getTotalPrice()

    if (wallet.owned_stocks >= this.number_of_stocks) {
      wallet.owned_stocks -= this.number_of_stocks
      wallet.money_available += this.total_price
      message.removeErrorMessage()
    }
    else {
      message.notEnoughStocks()
    }
    wallet.update()
  }
}
```

modules/result.js

```
import { wallet } from './wallet.js'
import { level } from './level.js'

export function stopSimulation() {
  let isPassed = wallet.profit >= level.required_profit


window.open(`result.html?profit=${wallet.profit}&required_profit=${level.required_profit}&isPassed=${isPassed
}`, '_self')
}
```

modules/charts/average.js

```
import { stock } from '../stock.js'

export function getSimpleMovingAverage(day) {
  let sum = 0
  let average = 0

  for (let i = stock.count - day + 1; i <= stock.count; i++) {
    sum = sum + stock.data[i].close
  }

  average = sum / day
  return average
}

export function getWeightedMovingAverage(day) {
  let sum = 0
  let average = 0
  let weight = 0
  let total_weight = 0

  for (let i = stock.count - day + 1; i <= stock.count; i++) {
    weight++
    sum = sum + weight * stock.data[i].close
    total_weight = total_weight + weight
  }

  average = sum / total_weight
  return average
}
```

modules/charts/candlestick.js

```
import { stock } from "../stock.js"
import { pattern } from '../pattern-recognition/pattern.js'
import { newShape } from '../new-layout/new-shape.js'
import { newAnnotation } from '../new-layout/new-annotation.js'
```

```
export let candlestick = {
  container: 'candlestick-chart',
  data: [{
    x: [],
    open: [],
    high: [],
    low: [],
    close: [],
    type: 'candlestick'
  }],
  layout: {
    dragmode: 'zoom',
    title: 'Candlestick Chart',
    shapes: [],
    annotations: [],
    xaxis: {
      autorange: true,
      type: 'date',
      rangeslider: { visible: false }
    },
    yaxis: {
      autorange: true,
      type: 'linear',
      title: 'Price($)'
    }
  },
  extend: function() {
    if (this.data[0].x.length >= 14) {
      this.data[0].x.shift()
      this.data[0].open.shift()
      this.data[0].high.shift()
      this.data[0].low.shift()
      this.data[0].close.shift()
    }
    this.data[0].x.push(stock.data[stock.count].date)
    this.data[0].open.push(stock.data[stock.count].open)
    this.data[0].high.push(stock.data[stock.count].high)
    this.data[0].low.push(stock.data[stock.count].low)
    this.data[0].close.push(stock.data[stock.count].close)
  },
  updateShapes: function() {
    if (this.layout.shapes.length > 12) this.layout.shapes.shift()
    let new_shape = pattern.name ? newShape(pattern.prev.date, pattern.curr.date) : null
    this.layout.shapes.push(new_shape)
  },
  updateAnnotations: function() {
    if (this.layout.annotations.length > 12) this.layout.annotations.shift()
    let new_annotation = pattern.name ? newAnnotation(pattern.prev.date, pattern.name) : null
    this.layout.annotations.push(new_annotation)
  }
}
```

modules/charts/moving-average.js

```
import { stock } from "../stock.js"
import { getSimpleMovingAverage, getWeightedMovingAverage } from './average.js'
import { newShape } from '../new-layout/new-shape.js'
import { newAnnotation } from '../new-layout/new-annotation.js'
import { point } from '../intersection/point.js'
import { getSlope } from '../intersection/get-slope.js'
import { isGoldenCross } from '../intersection/is-golden-cross.js'
import  { updatePlot } from './plot.js'

export let moving_average = {
  day_short: 5,
  day_long: 15,
  type: 'sma',
  container: 'moving-average-chart',
  data: [
    {
      x: [],
      y: [],
      type: 'scatter',
      mode: 'lines',
      name: 'Short-term SMA'
    },
    {
      x: [],
      y: [],
      type: 'scatter',
      mode: 'lines',
      name: 'Long-term SMA'
    }
  ],
  layout: {
    dragmode: 'zoom',
    title: 'Moving Average Chart',
    shapes: [],
    annotations: [],
    xaxis: {
      autorange: true,
      type: 'date',
      rangeslider: { visible: false },
    },
    yaxis: {
      autorange: true,
      type: 'linear',
      title: 'Price($)'
    }
  },
  extendShortMovingAverage: function() {
    if (this.data[0].x.length >= 14) {
      this.data[0].x.shift()
      this.data[0].y.shift()
    }

    if (stock.count >= this.day_short) {
```

```
    this.data[0].x.push(stock.data[stock.count].date)

    let y = (this.type === 'sma')
      ? getSimpleMovingAverage(this.day_short)
      : getWeightedMovingAverage(this.day_short)

    this.data[0].y.push(y)
  }
},
extendLongMovingAverage: function() {
  if (this.data[1].x.length >= 14) {
    this.data[1].x.shift()
    this.data[1].y.shift()
  }

  if (stock.count >= this.day_long) {
    this.data[1].x.push(stock.data[stock.count].date)

    let y = (this.type === 'sma')
      ? getSimpleMovingAverage(this.day_long)
      : getWeightedMovingAverage(this.day_long)

    this.data[1].y.push(y)
  }
},
updateShapes: function() {
  if (this.layout.shapes.length > 12) this.layout.shapes.shift()

  let new_shape = point.intersect ? newShape(point.prev.short.x, point.curr.short.x) : null

  this.layout.shapes.push(new_shape)
},
updateAnnotations: function() {
  if (this.layout.annotations.length > 12) this.layout.annotations.shift()

  let slope_short = getSlope(point.prev.short.y, point.curr.short.y)
  let slope_long = getSlope(point.prev.long.y, point.curr.long.y)
  let text = isGoldenCross(slope_short, slope_long) ? 'Buy' : 'Sell'

  let new_annotation = point.intersect ? newAnnotation(point.prev.short.x, text) : null

  this.layout.annotations.push(new_annotation)
},
changeSettings: function() {
  let short_term_input = Number(document.getElementById('short-term-input').value)
  let long_term_input = Number(document.getElementById('long-term-input').value)
  let type_input = document.getElementById('options').value

  if (short_term_input > 15 || short_term_input < 1 || !Number.isInteger(short_term_input)) {
    alert('Short-term input must be an integer between 1 and 14')
    return this.resetInputs()
  }
  else if (long_term_input > 30 || long_term_input < 15 || !Number.isInteger(long_term_input)) {
    alert('Long-term input must be an integer between 15 and 30')
```

```
      return this.resetInputs()
    }

    this.type = type_input

    this.data[0].x = []
    this.data[0].y = []
    this.data[0].name = 'Short-term ' + this.type.toUpperCase()

    this.data[1].x = []
    this.data[1].y = []
    this.data[1].name = 'Long-term ' + this.type.toUpperCase()

    this.layout.shapes = []
    this.layout.annotations = []

    updatePlot(this)
  },
  resetInputs: function() {
    document.getElementById('short-term-input').value = this.day_short
    document.getElementById('long-term-input').value = this.day_long
    document.getElementById('options').value = this.type
  }
}
```

modules/charts/plot.js

```
export function drawNewPlot(plot) {
  Plotly.newPlot(plot.container, plot.data, plot.layout);
}

export function updatePlot(plot) {
  Plotly.update(plot.container, plot.data, plot.layout)
}
```

modules/intersection/do-intersect.js

```
import { isSameSign } from './is-same-sign.js'
import { stock } from '../stock.js'

export function doIntersect(prev, curr) {
  let x1 = stock.count - 1, y1 = prev.short.y
  let x2 = stock.count, y2 = curr.short.y
  let x3 = stock.count - 1, y3 = prev.long.y
  let x4 = stock.count, y4 = curr.long.y
  let a1, b1, c1, a2, b2, c2
  let r1, r2, r3, r4
  let denom

  a1 = y2 - y1
  b1 = x1 - x2
  c1 = x2 * y1 - x1 * y2
```

```
  r3 = a1 * x3 + b1 * y3 + c1
  r4 = a1 * x4 + b1 * y4 + c1

  if (r3 !== 0 && r4 !== 0 && isSameSign(r3, r4)) return false

  a2 = y4 - y3
  b2 = x3 - x4
  c2 = x4 * y3 - x3 * y4

  r1 = a2 * x1 + b2 * y1 + c2
  r2 = a2 * x2 + b2 * y2 + c2

  if (r1 !== 0 && r2 !== 0 && isSameSign(r1, r2)) return false

  denom = a1 * b2 - a2 * b1
  if (denom === 0) return true

  return true
}
```

modules/intersection/get-slope.js

```
export function getSlope(y1, y2) {
  return (y2 - y1)
}
```

modules/intersection/is-golden-cross.js

```
export function isGoldenCross(slope_short, slope_long) {
  return slope_short > slope_long
}
```

modules/intersection/is-same-sign.js

```
export function isSameSign(a, b) {
  return Math.sign(a) == Math.sign(b)
}
```

modules/intersection/point.js

```
import { doIntersect } from './do-intersect.js'
import { moving_average } from '../charts/moving-average.js'

export let point = {
  prev: {
    short: {},
    long: {}
  },
  curr: {
    short: {},
    long: {}
  },
```

```
  intersect: false,
  updatePrev: function() {
    this.prev.short = {
        x: moving_average.data[0].x[moving_average.data[0].x.length - 2],
        y: moving_average.data[0].y[moving_average.data[0].y.length - 2]
    }
    this.prev.long = {
        x: moving_average.data[1].x[moving_average.data[1].x.length - 2],
        y: moving_average.data[1].y[moving_average.data[1].y.length - 2]
    }
  },
  updateCurr: function() {
    this.curr.short = {
        x: moving_average.data[0].x[moving_average.data[0].x.length - 1],
        y: moving_average.data[0].y[moving_average.data[0].y.length - 1]
    }
    this.curr.long = {
        x: moving_average.data[1].x[moving_average.data[1].x.length - 1],
        y: moving_average.data[1].y[moving_average.data[1].y.length - 1]
    }
  },
  updateIntersect: function() {
    this.intersect = doIntersect(this.prev, this.curr)
  }
}
```

modules/new-layout/new-annotation.js

```
export function newAnnotation(startDate, text) {
  return {
    x: startDate,
    y: 1,
    xref: 'x',
    yref: 'paper',
    text: text,
    font: { color: 'black', size: 8 },
    showarrow: false,
    xanchor: 'left',
    ax: 0,
    ay: 0
  }
}
```

modules/new-layout/new-shape.js

```
export function newShape(startDate, endDate) {
  return {
    type: 'rect',
    xref: 'x',
    yref: 'paper',
    x0: startDate,
    y0: 0,
    x1: endDate,
```

```
    y1: 1,
    fillcolor: '#ffff00',
    opacity: 0.4,
    line: { width: 0 }
  }
}
```

modules/pattern-recognition/candlestick-patterns.js

```
export function isBearishKicker(prev, curr) {
  return prev.open < prev.close &&
         curr.open > curr.close &&
         prev.open > curr.open
}

export function isBullishKicker(prev, curr) {
  return prev.open > prev.close &&
         curr.open < curr.close &&
         prev.open < curr.open
}

export function isShootingStar(prev, curr) {
  return prev.open < prev.close &&
         curr.open > curr.close &&
         prev.close < curr.close &&
         curr.high - curr.close > 2 * (curr.open - curr.close) &&
         curr.open - curr.close > curr.close - curr.low
}
```

modules/pattern-recognition/pattern.js

```
import { isBearishKicker, isBullishKicker, isShootingStar } from './candlestick-patterns.js'
import { stock } from '../stock.js'
import { candlestick } from '../charts/candlestick.js'

export let pattern = {
  show: true,
  prev: {},
  curr: {},
  name: '',
  toggleShow: function() {
    this.show = !this.show
    candlestick.layout.shapes = []
    candlestick.layout.annotations = []
  },
  updatePrev: function() {
    this.prev = stock.data[stock.count - 1]
  },
  updateCurr: function() {
    this.curr = stock.data[stock.count]
  },
  updateName: function() {
    if (isBearishKicker(this.prev, this.curr)) {
```

```
      this.name = 'Bearish Kicker'
    } else if (isBullishKicker(this.prev, this.curr)) {
      this.name = 'Bullish Kicker'
    } else if (isShootingStar(this.prev, this.curr)) {
      this.name = 'Shooting Star'
    } else {
      this.name = ''
    }
  }
}
```

Bibliography

Amadeo, K., 2021. *What is the ideal GDP growth rate?.* [Online]
Available at: https://www.thebalance.com/what-is-the-ideal-gdp-growth-rate-3306017
[Accessed 1 March 2021].
Bybit Learn, 2020. *16 Must-Know Candlestick Patterns for a Successful Trade.* [Online]
Available at: https://learn.bybit.com/trading/best-candlestick-patterns/
[Accessed 31 March 2021].
Chen, J., 2020. *Crossover.* [Online]
Available at: https://www.investopedia.com/terms/c/crossover.asp
[Accessed 8 April 2021].
Forex Trading 200, 2018. *Candlestick Charts.* [Online]
Available at: https://www.forextrading200.com/candlestick-chart/
[Accessed 31 March 2021].
Hayes, A., 2020. *Candlestick Definition.* [Online]
Available at: https://www.investopedia.com/terms/c/candlestick.asp
[Accessed 31 March 2021].
Knueven, L., 2020. *The average stock market return over the past 10 years.* [Online]
Available at: https://www.businessinsider.com/personal-finance/average-stock-market-return?op=1
[Accessed 1 March 2021].
Konchar, P., 2018. *What is the legal age to trade Forex?.* [Online]
Available at: https://mytradingskills.com/legal-age-to-trade
[Accessed 1 March 2021].
Mahony, J., 2019. *A trader's guide to moving averages.* [Online]
Available at: https://www.ig.com/uk/trading-strategies/moving-averages--how-to-calculate-them-and-use-them-in-your-trad-181008
[Accessed 8 April 2021].
Matange, S., 2014. *CandleStick Chart.* [Online]
Available at: https://blogs.sas.com/content/graphicallyspeaking/2014/09/27/candlestick-chart/
[Accessed 31 March 2021].