Dokumentacja Projektu

Stock Market Simulator to aplikacja odpowiedzialna za wierne zasymulowanie operowania aktywami takimi jak waluty, złoto oraz kryptowaluty. Umożliwia ona kupowanie i sprzedawanie aktyw po sprawdzanych na bieżąco cenach.

Część Graficzna (GUI)

startmenu.h :: 'StartMenu'

Klasa StartMenu odpowiada za główne okno aplikacji, zarządzanie danymi finansowymi oraz interakcję z użytkownikiem. Jest to centralny punkt, z którego użytkownik może przejść do różnych funkcji aplikacji, takich jak historia transakcji, portfolio, szczegóły instrumentów, depozyty i wypłaty.

Pola Prywatne:

- `Ui::StartMenu* ui` - wskaźnik na interfejs użytkownika generowany przez Qt Designer.
- `QTimer* timer` - timer odpowiedzialny za aktualizację daty i czasu.
- `QTimer* midnightTimer` - timer odpowiedzialny za sprawdzanie statusu rynku w północy.
- `double balance` - obecny stan konta użytkownika.
- `double allocatedFunds` - ilość zaalokowanych środków.
- `double freeFunds` - ilość wolnych środków.
- `int currentWeekday` - obecny dzień tygodnia.
- `QString marketStatus` - status rynku ("open" lub "closed").

Metody:

- `StartMenu(QWidget *parent = nullptr)` - konstruktor klasy, inicjalizuje interfejs użytkownika oraz ustawia początkowe wartości pól.
- `~StartMenu()` - destruktor klasy, zapisuje ustawienia do pliku CSV oraz usuwa dynamicznie alokowane obiekty.
- `void showDetailsWidget(const QString &instrumentName)` - wyświetla widget szczegółów dla wybranego instrumentu.
- `void showDetails(const QString &instrumentName)` - wyświetla szczegóły wybranego instrumentu.
- `void updateDateTime()` - aktualizuje datę i czas na interfejsie użytkownika.
- `void updateRefreshButton()` - aktualizuje tekst na przycisku odświeżania.
- `void updateMarketStatus()` - aktualizuje status rynku w zależności od dnia tygodnia.
- `void addDepositAmount(double amount)` - dodaje kwotę depozytu do stanu konta.
- `void subtractWithdrawalAmount(double amount)` - odejmuje kwotę wypłaty od stanu konta.
- `void updateBalanceDisplay()` - aktualizuje wyświetlaną wartość stanu konta.
- `void updateAllocatedFundsDisplay()` - aktualizuje wyświetlaną wartość zaalokowanych środków.
- `void updateFreeFundsDisplay()`- aktualizuje wyświetlaną wartość wolnych środków.
- `void updateBalanceDisplays()` - ładuje wartości z pliku CSV i aktualizuje wyświetlane wartości.
- `void showDepositDialog()` - wyświetla okno dialogowe do dodania depozytu.
- `void showWithdrawalDialog()` - wyświetla okno dialogowe do wypłaty środków.

- `void saveSettingstoCSV()` - zapisuje ustawienia do pliku CSV.
- `void loadSettingsfromCSV()` - ładuje ustawienia z pliku CSV.
- `void loadBalancefromCSV()` - ładuje wartości stanu konta z pliku CSV.
- `void saveBalancetoCSV()` - zapisuje wartości stanu konta do pliku CSV.
- `void fetchDataAndWriteToCSV()` - pobiera dane z API i zapisuje je do pliku CSV.
- `void refreshDataAndReloadTable()` - odświeża dane i przeładowuje tabelę instrumentów.

Zdarzenia:

- `void balanceChanged(double newBalance)` - sygnał emitowany po zmianie stanu konta.
- `void freeFundsChanged(double newFreeFunds)` - sygnał emitowany po zmianie wolnych środków.

tradinginstruments.h :: 'TradingInstruments'

Plik 'tradinginstruments' zawiera implementację klasy TradingInstruments, która dziedziczy po QDialog i służy do wyświetlania tabeli instrumentów handlowych w interfejsie użytkownika. Klasa obsługuje również ładowanie danych z pliku, wyświetlanie ich w tabeli oraz reagowanie na kliknięcia użytkownika.

Metody:

- `TradingInstruments(QWidget *parent)` - Konstruktor inicjuje interfejs użytkownika, ustawia tabelę, motyw kolorystyczny oraz połączenie sygnału kliknięcia komórki w tabeli z odpowiednim slotem.
- `~TradingInstruments()` - Destruktor zwalnia zasoby zajmowane przez interfejs użytkownika.
- `setupTable()` - Metoda inicjalizuje tabelę, ustawiając nagłówki kolumn oraz ich rozmiar.
- `setDarkTheme()` - Metoda ustawia ciemny motyw dla tabeli, scrollbarów oraz sekcji nagłówków.
- `populateTable(const QStringList &data)` - Metoda wypełnia tabelę danymi, ustawiając poszczególne wiersze i kolumny jako nieedytowalne oraz wyśrodkowane.
- `loadInstruments(const QString &filePath)` - Metoda ładuje dane z pliku podanego jako argument, odczytuje jego zawartość linia po linii i przekazuje do metody populateTable.

Sloty:

- `onCellClicked(int row, int column)` - Slot reagujący na kliknięcie komórki w tabeli. Jeśli kliknięta komórka znajduje się w pierwszej kolumnie, tworzy obiekt odpowiedniego instrumentu, odświeża jego dane, zapisuje je do pliku i otwiera nowe okno details z aktualizowanymi szczegółami.

details.h :: 'Details'

Klasa details jest odpowiedzialna za wyświetlanie szczegółowych informacji o aktywie, takich jak cena, wskaźniki oraz wykresy, a także umożliwia kupno i sprzedaż aktywów.

Metody:

- `details(QWidget *parent)` - Konstruktor klasy, inicjalizuje elementy interfejsu użytkownika oraz łączy sygnały przycisków kupna i sprzedaży z odpowiednimi slotami.
- `~details()` - Destruktor klasy, zwalnia pamięć zarezerwowaną dla ui oraz instrumentObject, jeśli jest zainicjalizowany.
- `void updateDetails(const QString &instrumentName)` - Metoda odpowiedzialna za aktualizację szczegółowych informacji o aktywie na podstawie jego nazwy. Odczytuje dane z pliku CSV, ustawia

odpowiednie etykiety w interfejsie oraz tworzy obiekt odpowiedniego typu aktywa (np. Gold, Currency, CryptoApi).
- `void loadChartData(const QString &chartFilePath)` - Metoda odpowiedzialna za ładowanie danych wykresu z pliku CSV oraz rysowanie wykresu cenowego. Ustawia odpowiednie osie i dodaje wykres do widżetu chartWidget.
- `void loadIndicators(const QString &chartFilePath)` - Metoda odpowiedzialna za ładowanie wartości wskaźników (MA, Volatility, MACD) z pliku CSV i wyświetlanie ich w interfejsie użytkownika.
- `void onBuyButtonClicked()` - Slot odpowiedzialny za obsługę kliknięcia przycisku kupna. Sprawdza poprawność wprowadzonej ilości, wykonuje akcję kupna za pomocą metody Serializer::performAction, aktualizuje dane oraz wyświetla komunikaty informacyjne.
- `void onSellButtonClicked()` - Slot odpowiedzialny za obsługę kliknięcia przycisku sprzedaży. Sprawdza poprawność wprowadzonej ilości, wykonuje akcję sprzedaży za pomocą metody Serializer::performAction, aktualizuje dane oraz wyświetla komunikaty informacyjne.

Sloty:

- `void onBuyButtonClicked()` - Slot onBuyButtonClicked jest odpowiedzialny za obsługę zdarzenia kliknięcia przycisku kupna. Sprawdza poprawność wprowadzonej ilości do zakupu. Wykonuje akcję kupna aktywa przy użyciu metody Serializer::performAction. Aktualizuje dane oraz wyświetla komunikaty informacyjne.
- `void onSellButtonClicked()` - Slot onSellButtonClicked jest odpowiedzialny za obsługę zdarzenia kliknięcia przycisku sprzedaży. Sprawdza poprawność wprowadzonej ilości do sprzedaży. Wykonuje akcję sprzedaży aktywa przy użyciu metody Serializer::performAction. Aktualizuje dane oraz wyświetla komunikaty informacyjne.

deposit.h :: 'Deposit'

Klasa Deposit reprezentuje okno dialogowe służące do realizacji symulowania wpłat.

Metody:

- `Deposit(QWidget *parent = nullptr)` - Konstruktor, inicjalizuje interfejs użytkownika dla okna dialogowego wpłaty. Łączy sygnały kliknięcia przycisków cancelButton i depositButton z odpowiednimi slotami reject oraz handleDeposit.
- `~Deposit()` - Destruktor, zwalnia zasoby przydzielone dla interfejsu użytkownika.

Sloty:

- `void handleDeposit()` - Slot handleDeposit jest odpowiedzialny za obsługę zdarzenia kliknięcia przycisku wpłaty. Pobiera wartość wpłaty z pola tekstowego, emituje sygnał depositMade i zamyka okno dialogowe.

withdrawal.h :: 'Withdrawal'

Klasa Withdrawal reprezentuje okno dialogowe służące do symulacji realizacji wypłat.

Metody:

- `withdrawal(QWidget *parent = nullptr)` - Inicjalizuje interfejs użytkownika dla okna dialogowego wypłaty. Łączy sygnały kliknięcia przycisków cancelButton i withdrawalButton z odpowiednimi slotami

reject oraz handleWithdrawal.
- `~withdrawal()` - Zwalnia zasoby przydzielone dla interfejsu użytkownika.
- `double loadFreeFunds()` - Metoda loadFreeFunds odpowiada za wczytanie dostępnych środków z pliku balance.csv.

Sloty:

- `void handleWithdrawal()` - Slot handleWithdrawal jest odpowiedzialny za obsługę zdarzenia kliknięcia przycisku wypłaty. Sprawdza dostępność środków, pobiera wartość wypłaty z pola tekstowego, emituje sygnał withdrawalMade i zamyka okno dialogowe.

history.h :: 'History'

Klasa History odpowiada za wyświetlanie i zarządzanie historią transakcji użytkownika. Umożliwia wczytywanie danych z pliku oraz ich wyświetlanie w tabeli.

Metody:

- `History(QWidget *parent)` - Konstruktor klasy History tworzy nowy obiekt QVBoxLayout oraz tabelę QTableWidget. Następnie ustawia układ dla widżetu oraz aplikuje ciemny motyw.
- `void setupTable()` - Metoda setupTable ustawia nagłówki kolumn oraz konfigurację tabeli.
- `void setDarkTheme()` - Metoda setDarkTheme aplikuje ciemny motyw do tabeli.
- `void loadHistory()` - Metoda loadHistory wczytuje dane z pliku i przekazuje je do metody populateTable w celu wyświetlenia w tabeli.
- `void populateTable()` - Metoda populateTable wypełnia tabelę danymi wczytanymi z pliku.

portfolio.h :: 'Portfolio'

Klasa Portfolio odpowiada za wyświetlanie i zarządzanie portfelem użytkownika. Umożliwia wczytywanie danych z pliku oraz ich wyświetlanie w tabeli.

Metody:

- `Portfolio(QWidget *parent)` - Konstruktor klasy Portfolio tworzy nowy obiekt QVBoxLayout oraz tabelę QTableWidget. Następnie ustawia układ dla widżetu oraz aplikuje ciemny motyw.
- `void setupTable()` - Metoda setupTable ustawia nagłówki kolumn oraz konfigurację tabeli.
- `void setDarkTheme()` - Metoda setDarkTheme aplikuje ciemny motyw do tabeli.
- `void loadPortfolio(const QString &filePath)` - Metoda loadPortfolio wczytuje dane z pliku i przekazuje je do metody populateTable w celu wyświetlenia w tabeli portfela.
- `void populateTable(const QStringList &data)` - Metoda populateTable wypełnia tabelę portfela danymi wczytanymi z pliku.

---

main.cpp

Odpowiada za odpalenie startmenu

```
#include "startmenu.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    StartMenu w;
    w.show();
    return a.exec();
}
```

Praca z API

data.h :: 'Data'

Jest to abstrakcyjna klasa która ustala metody wymagane do pobierania danych z poszczególnych systemów Api.

Metody:

- `void ParseData(const QString& jsonData, double& rate, int type)` - odpowiada za parsowanie danych tekstowych na słownik w formacie json oraz wybranie z niego ceny, zapisywanej do zmiennej przekazywanej przez referencje 'rate'. Parametr type przydatny będzie w niektórych przypadkach, dla zapewnienia uniwersalnego działania metody
- `double getPastValue(int days)` - odpowiada za pozyskiwanie przeszłej ceny aktywa. Parametr 'days' ustala date pobieranej ceny w dniach wstecz
- `QString generateDate(int days)` - metoda odpowiedzialna za generowanie przeszłej daty w odpowiednim formacie, aby móc jej użyć do pobierania danych z przeszłości. Parametr 'days' ustala ilość dni wstecz do zwracanej daty
- `QString getName() const` - metoda zwracająca skrót nazwy poszczególnego aktywa, potrzebna do wyświetlania w gui
- `void refreshData()` - metoda odpowiedzialna za odświeżenie danych z api w czasie rzeczywistym
- `double getValue() const` - metoda zwracająca obecną cenę rynkową (w chwili ostatniego odswieżenia za pomocą metody 'refreshData')
- `double difference()` - metoda odpowiedzialna za uzyskanie procentowej zmiany ceny z ostatnich 24 godzin
- `QVector<double> monthly()` - metoda zwracająca dane cen aktywa z ostatnich 30 dni w postaci vectora, tak aby można było z nich korzystać do tworzenia wykresu
- `double getMA()` - metoda odpowiadająca za zwrócenie wartości wskaźnika Moving Average (korzysta z klasy 'MovingAverage')
- `double getVolatility()` - metoda odpowiadająca za zwrócenie wartości wskaźnika Volatility (korzysta z klasy 'Volatility')
- `double getMACD()` - metoda odpowiadająca za zwrócenie wartości wskaźnika MACD (korzysta z klasy 'MACD')

'CryptoApi'

Klasa odpowiedzialna za pracę z systemem CryptoCompare Api. Implementuje ona w odpowiedni sposób metody klasy 'Data'. Konstruułąc za jej pomocą obiekt, robimy to np. w sposób następujący:

```
auto btc = CryptoApi("BTC");
```

'NBPApi'

Klasa odpowiedzialna za pracę z systemem NBP Api. Implementuje część metod klasy 'Data', tak aby umożliwić dziedziczenie po niej kolejnych klas. Jej zadaniem jest uogólnić kod jej podklas Currency i Gold, tak aby uniknąć powtarzania tego samego kodu

'Currency'

Dziedziczy po klasie częściowo abstrakcyjnej 'NBPApi' oraz implementując metody które odpowiadają za pobieranie ceny walut. Konstruułąc za jej pomocą obiekt, robimy to np. w sposób następujący:

```
auto usd = Currency("USD");
```

'Gold'

Dziedziczy po klasie częściowo abstrakcyjnej 'NBPApi' oraz implementując metody które odpowiadają za pobieranie cene złota. Konstruułąc za jej pomocą obiekt, robimy to np. w sposób następujący:

```
auto gold = Gold();
```

'MovingAverage'

Klasa odpowiadająca za obliczanie wartości wskaźnika Moving Average, za pomocą wewnętrznej metody `static double calculate(const double* prices, int days)`, gdzie 'prices' oczekuje tablicy cen z ostatnich dni, a 'days' liczby dni uwzględnianych do obliczania wskaźnika

'Volatility'

Klasa odpowiadająca za obliczanie wartości wskaźnika Volatility, za pomocą wewnętrznej metody `static double calculate(const double* prices, int days)`, gdzie 'prices' oczekuje tablicy cen z ostatnich dni, a 'days' liczby dni uwzględnianych do obliczania wskaźnika

'MACD'

Klasa odpowiadająca za obliczanie wartości wskaźnika Moving Average, za pomocą wewnętrznej metody `static double calculate(const double* prices, int daysShort, int daysLong)`, gdzie 'prices' oczekuje tablicy cen z ostatnich dni, a 'daysShort' oraz 'daysLong' liczby dni uwzględnianych do obliczania wskaźnika

---

Praca z plikami lokalnymi

'Serializer'

To klasa odpowiadająca za całą pracę plikami lokalnymi. Jej celem jest zapisywaniu wykonywanych transakcji, stanu konta i posiadanych aktyw plikach 'history.csv', 'portfolio.csv' oraz 'balance.csv'. Dzięki tej klasie użytkownik nie traci danych po zamknięciu aplikacji

Metody:

- `static void loadPortfolio()` - ładuje dane z pliku 'portfolio.csv' do słownika wewnątrz klasy
- `static void savePortfolio()` - zapisuje dane ze słownika do pliku 'portfolio.csv'
- `static void updatePortfolio(const QString& assetName, double quantity)` - aktualizuje słownik wewnątrz klasy po zakupie/sprzedaży aktywa o nazwie 'assetName' oraz ilości 'quantity'. Warto dodać że sprzedaż lub kupno jest rozróżniane tutaj poprzez znak paramatru quantity
- `static void performAction(const Data& asset, double quantity)` - główna metoda odpowiedzialna za kupno/sprzedaż aktywa. Aktualizuje ona zarówno historie zakupów (history.csv) jak i posiadane obecnie aktywa (portfolio.csv) oraz posiadane i zaalokowane środki (balance.csv)
- `static void deleteHistory()` - czyści zawartość pliku history.csv
- `static void deletePortfolio()` - czyści zawartość pliku portfolio.csv
- `static void changeFreeFunds(double amount)` - zmienia ilość posiadanych wolnych środków zapisanych w pliku 'balance.csv'
- `static void changeAllocatedFunds(double amount)` - zmienia ilość zaalokowanych środków zapisanych w pliku 'balance.csv'
- `static void loadBalance()` - ładuje wartości posiadanych i zaalokowanych środków z pliku 'balance.csv' do klasy
- `static void saveBalance()` - zapisuje wartości posiadanych i zaalokowanych środków do pliku 'balance.csv' z pól prywatnych klasy