

Projekt Indywidualny - Aplikacja do obsługi giełd kryptowalutowych

Adam Kasperowicz – 279046

Repozytorium projektu: <https://github.com/Kadek/EasyCrypto>

Spis treści

| | |
|------------------------------|---|
| 1. Założenia Projektu | 2 |
| 2. Struktura aplikacji | 2 |
| a. Moduł GUI (gui) | 3 |
| b. Moduł Parser(macro)..... | 4 |
| c. Moduł Wizualizacji | 4 |
| d. Moduł Bazy Danych..... | 6 |
| e. Moduł komunikacji | 7 |
| 3. Fazy implementacji | 9 |

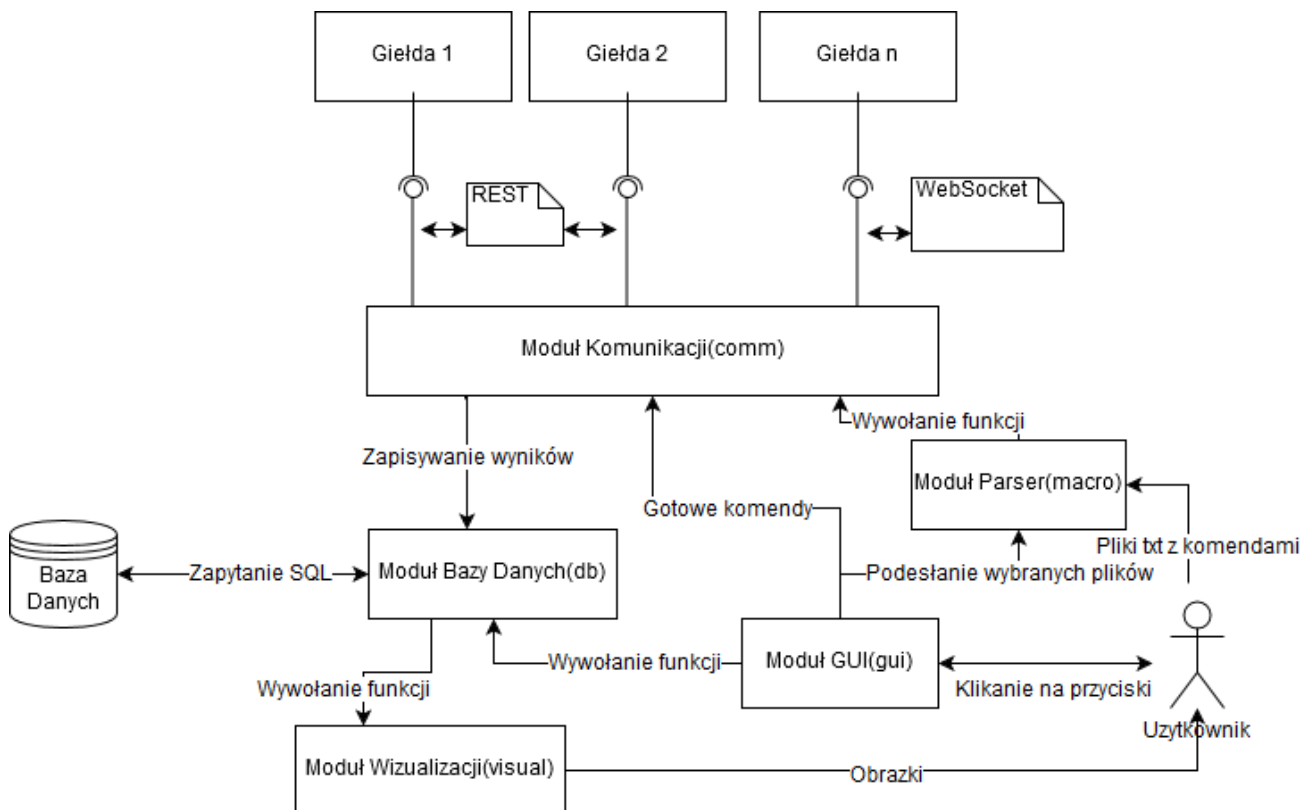
1. Założenia Projektu

Celem projektu było stworzenie aplikacji pozwalającej na wykonywanie operacji na wielu giełdach kryptowalut jednocześnie. Aplikacja powinna pozwalać nam na:

- uzyskiwanie danych z giełd bez potrzeby wchodzenia na stronę giełdy;
- zbieranie danych z giełdy w tle (np. zapisywanie w bazie danych obecnych wartości danej pary walut co minutę przez nieokreślony czas);
- jak najprostsze podłączanie nowych giełd do aplikacji;
- możliwość wyświetlania danych z baz danych w postaci wykresów itp.;
- wykonywanie transakcji bez potrzeby wchodzenia na giełdę;
- przyjemność i prostotę użytkowania.

2. Struktura aplikacji

Oprogramowanie zostało skonstruowane z modułów. Każdy z modułów odpowiedzialny jest wyłącznie za pewną część działania programu i tym samym zmniejsza złożoność ogólnego projektu. Poniżej przedstawiona została architektura całej aplikacji w postaci diagramu. Każdy z modułów zostaje dokładnie opisany w następnych podrozdziałach.



a. Moduł GUI (gui)

Element pozwala na prostą i intuicyjną obsługę aplikacji. Obsługa polega na wybieraniu jednej lub wielu giełd, następnie wyborze możliwych akcji, podaniu specyfikacji, jeśli takie są możliwe/wymagane i wreszcie zatwierdzeniu akcji.

Podstawowe funkcje interfejsu graficznego zostały poprawnie zaimplementowane. Kod znajduje się w pakiecie „src/gui”. Szata graficzna została utworzona z pomocą języka QML zaś funkcje kontrolne zaimplementowane zostały w Pythonie przy pomocy PyQt5.

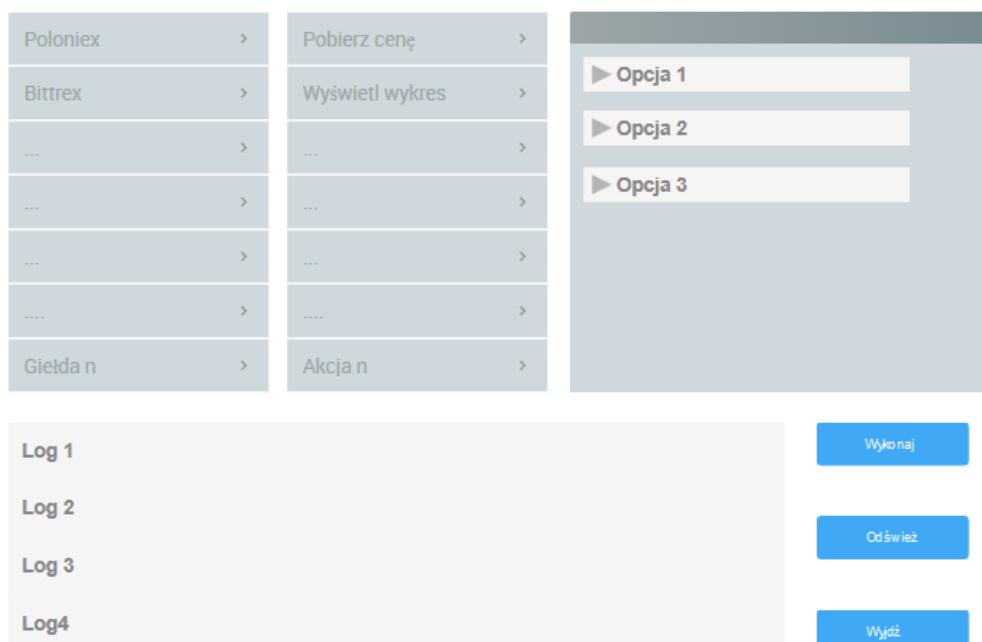
Niestety, biblioteka okazała się znacznie mniej przyjazna niż można było sądzić po pierwszym wrażeniu. Duże nieścisłości w dokumentacji, nieaktualne informacje oraz duże różnice pomiędzy wersjami sprawiły, że nawet najmniejsze elementy wymagały dużo pracy i ogromnego nakładu czasu.

Poniżej widać wersję interfejsu obecną prowizoryczną i planowaną ostateczną.

Obecny interfejs graficzny



Przewidywana ostateczna wersja interfejsu graficznego



b. Moduł Parser(macro)

Ten moduł ma za zadanie obsługę makr zapisywanych w specjalnym języku. Moduł parsuje otrzymane makra i przerabia je na wywołania odpowiednich funkcji z modułu komunikacyjnego. Z początku moduł będzie musiał zająć się tylko prostymi formułami typu.

„Poloniex: getTicker” – co zostałyby przetłumaczone na Poloniex.do(„getTicker”)

Ostateczna wersja modułu przewiduje funkcje typu:

- Pętle wywołujące komendy co dany interwał czasu
- Warunki pozwalające na programowanie prostych botów
- Wielowątkowość dająca możliwość działania wielu botom jednocześnie

Celem modułu jest pozwolenia użytkownikowi na wyjście poza ograniczenia narzucane przez Moduł GUI.

Jako, że większość zasobów czasowych została poświęcona modułowi GUI, moduł Parser pozostał w fazie planowania. W folderze „src/macro” możemy znaleźć przykładowe makro, które zawierałoby wszystkie funkcje potrzebne użytkownikowi.

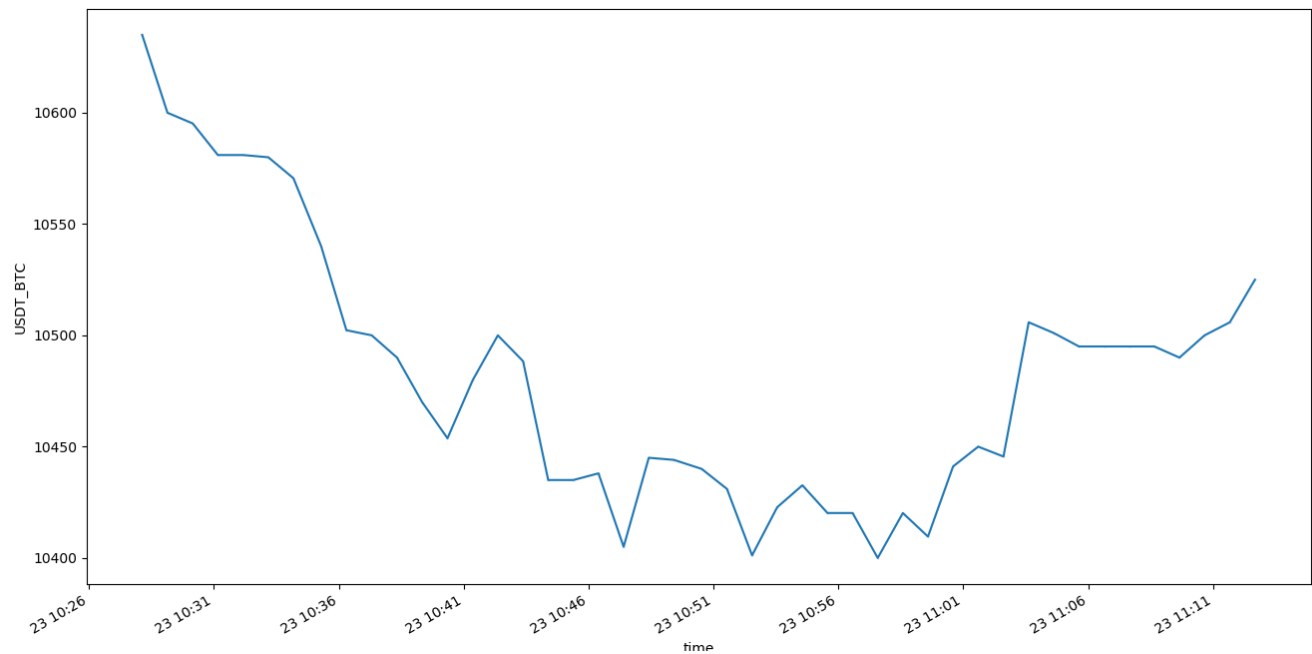
c. Moduł Wizualizacji

Moduł Wizualizacji to część programu zajmująca się tworzeniem wykresów dla danych. Na wejściu przyjmuje obiekty DTO (Data Transfer Object, opisane w module Bazy Danych), odczytuje z nich informacje o typie przechowywanych danych, a następnie tworzy odpowiednie wykresy dla użytkownika.

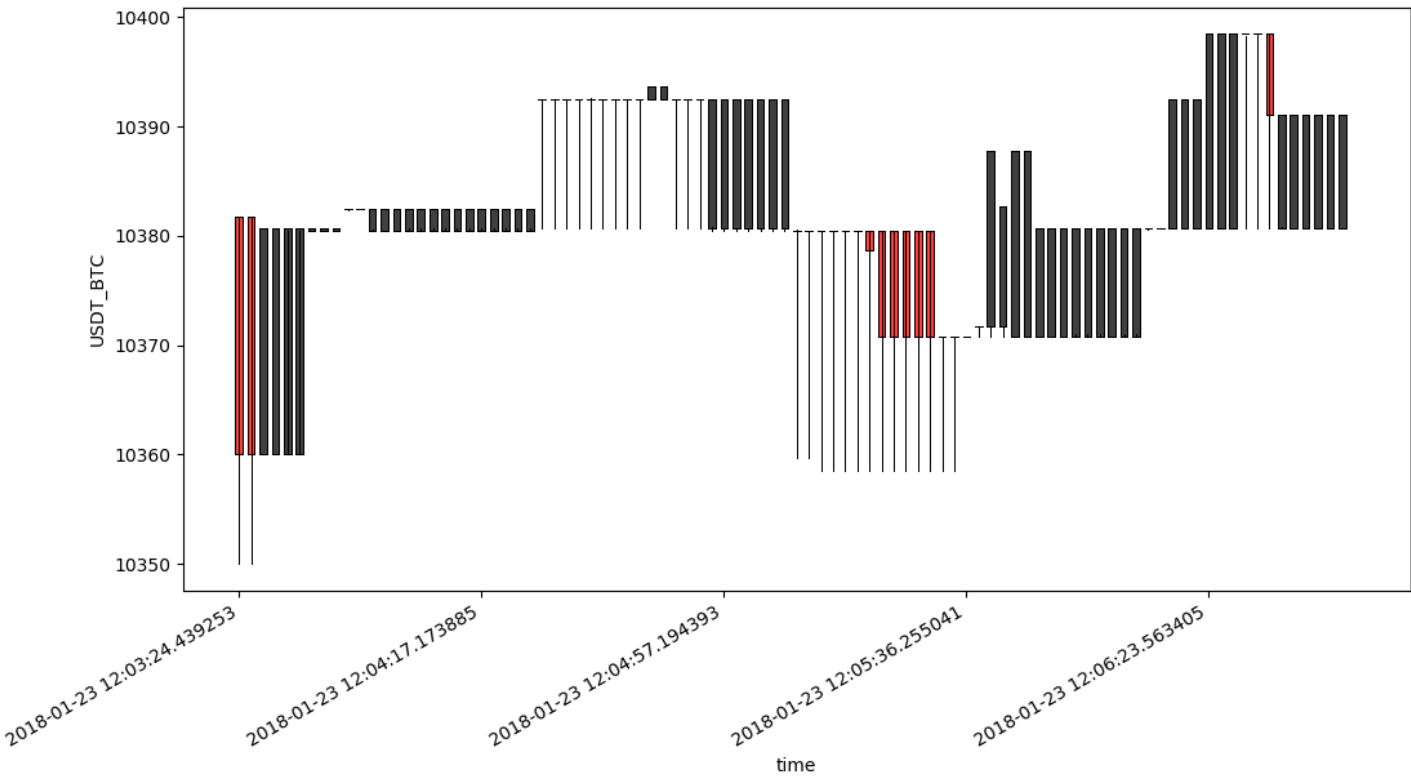
Moduł, który został zaimplementowany w bardzo prostej postaci pozwala nam wyświetlić wykres utworzony z cen zapisanych w naszej bazie danych. Poniżej widzimy dostępne diagramy cen:

- Wykres liniowy (Rysunek 1) – Tego rodzaju diagram pozwala nam na szybkie i ogólnikowe zobrazowanie zmienności kursu.
- Wykres słupkowy (Rysunek 2) – Wykorzystywany w analizie technicznej, przekazuje dokładne informacje o obecnych pozycjach w księdze zamówień (powtarzające się słupki przekazują tak naprawdę inną informację, lecz zmiana kursu pomiędzy dwoma słupkami jest tak mała, że może być ona niezauważalna na wykresie).

Rysunek 1 (Wykres liniowy)



Rysunek 2 (Wykres słupkowy)



d. Moduł Bazy Danych

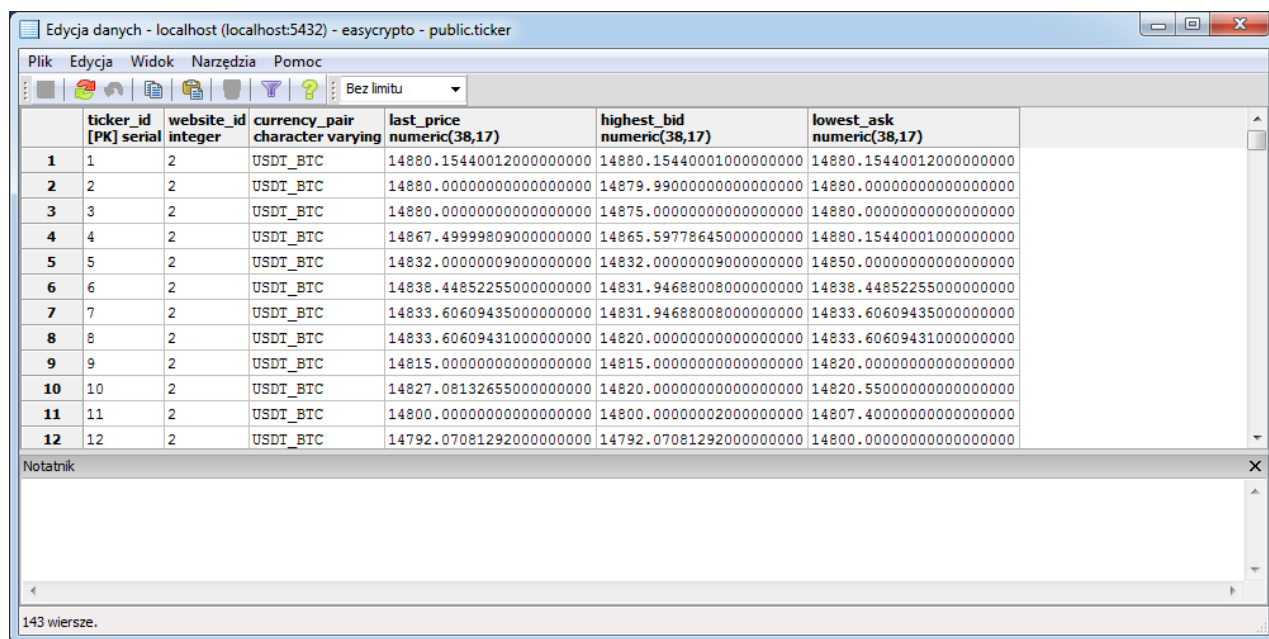
Moduł Bazy Danych ma za zadanie przetłumaczenie komend wysyłanych przez Moduł GUI oraz Moduł Komunikacji na odpowiednie wywołania funkcji. Moduł sięga do Bazy Danych poprzez zapytania SQL oraz do Modułu Wizualizacji poprzez obiekt DTO.

Obiekt DTO (Data Transfer Object) jest to wzorzec projektowy potrzebny nam do przesyłania dużych ilości skomplikowanych danych między modułami. Służy on nam do zserializowania danych otrzymywanych z giełd. Dzięki temu możemy pobrane informacje łatwo zapisywać w bazie danych a następnie wizualizować.

Wykorzystywaną technologią bazy danych jest PostgreSQL. Do obsługi zapytań wykorzystywana jest biblioteka SQLAlchemy. Bardzo miły w obsłudze zestaw narzędzi pozwolił na skuteczne utworzenie interfejsu pozwalającego innym modułom na dostęp do bazy danych.

W celu uruchomienia programu wymagana jest baza danych PostgreSQL obecna na komputerze. Baza powinna się nazywać „easycrypto”. Hasło do bazy umieszczamy w pliku „pass.txt”. Plik należy wcześniej utworzyć w folderze „src/db”.

Poniżej widzimy parę rekordów z tabeli „ticker”.



| | ticker_id [PK] serial | website_id integer | currency_pair character varying | last_price numeric(38,17) | highest_bid numeric(38,17) | lowest_ask numeric(38,17) |
|----|-----------------------|--------------------|---------------------------------|---------------------------|----------------------------|---------------------------|
| 1 | 1 | 2 | USDT_BTC | 14880.154400120000000000 | 14880.154400010000000000 | 14880.154400120000000000 |
| 2 | 2 | 2 | USDT_BTC | 14880.000000000000000000 | 14879.990000000000000000 | 14880.000000000000000000 |
| 3 | 3 | 2 | USDT_BTC | 14880.000000000000000000 | 14875.000000000000000000 | 14880.000000000000000000 |
| 4 | 4 | 2 | USDT_BTC | 14867.499998090000000000 | 14865.597786450000000000 | 14880.154400010000000000 |
| 5 | 5 | 2 | USDT_BTC | 14832.000000090000000000 | 14832.000000090000000000 | 14850.000000000000000000 |
| 6 | 6 | 2 | USDT_BTC | 14838.448522550000000000 | 14831.946880080000000000 | 14838.448522550000000000 |
| 7 | 7 | 2 | USDT_BTC | 14833.606094350000000000 | 14831.946880080000000000 | 14833.606094350000000000 |
| 8 | 8 | 2 | USDT_BTC | 14833.606094310000000000 | 14820.000000000000000000 | 14833.606094310000000000 |
| 9 | 9 | 2 | USDT_BTC | 14815.000000000000000000 | 14815.000000000000000000 | 14820.000000000000000000 |
| 10 | 10 | 2 | USDT_BTC | 14827.081326550000000000 | 14820.000000000000000000 | 14820.550000000000000000 |
| 11 | 11 | 2 | USDT_BTC | 14800.000000000000000000 | 14800.000000200000000000 | 14807.400000000000000000 |
| 12 | 12 | 2 | USDT_BTC | 14792.070812920000000000 | 14792.070812920000000000 | 14800.000000000000000000 |

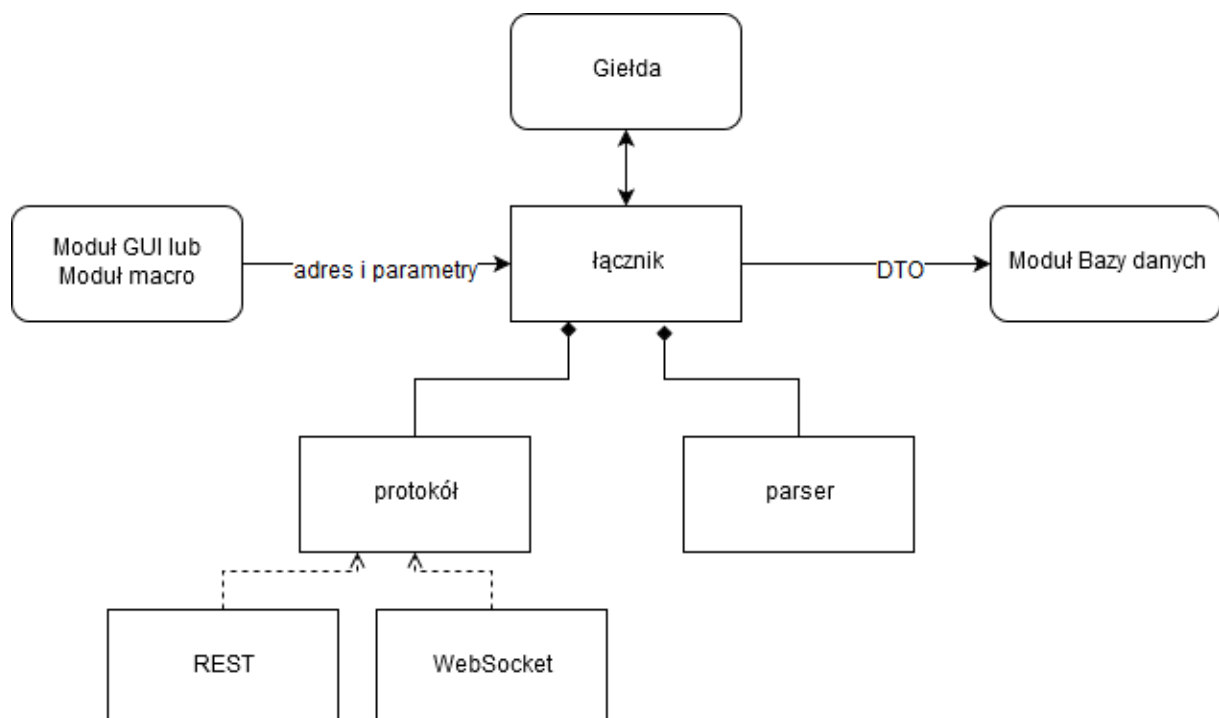
Struktura tabel definiowana jest formatem JSON. W folderze „src/db/data” znajdują się pliki JSON zawierające informacje o podstawowej strukturze tabel. Pliki te są czytane przez DBControl a następnie tworzone są tabele w bazie danych na ich wzór. Gdy chcemy dodać możliwość komunikacji z giełdą wystarczy dodać odpowiedni plik JSON oraz klasę Parser w module komunikacyjnym.

e. Moduł komunikacji

Zadaniem tego elementu jest hermetyzacja procesu komunikacji z giełdami. Bardzo ważne jest zaprojektowanie tego modułu w taki sposób, by dodawanie nowej giełdy wiązało się z jak najmniejszą ilością pracy. Dlatego wyodrębniamy części zmieniające się w każdej giełdzie i przedstawiamy metody najlepiej hermetyzujące daną część.

- Różne protokoły – API giełd wykorzystują często inne protokoły niż REST, dlatego dla każdego protokołu tworzymy klasę zajmującą się tylko danym protokołem. Klasa ta powinna przyjmować na wejście tylko adres i parametry zapytań dla danego protokołu. Dzięki temu dwie giełdy, wykorzystujące ten sam protokół, mogą wykorzystać dokładnie tą samą klasę.
- Różne formaty zwracanych danych – Każda giełda zwraca nam informację w trochę innym formacie. Niestety, tej charakterystyki nie da się zgeneralizować i klasa odpowiadająca za daną giełdę powinna posiadać własny parser, który otrzymane dane zapisuje do wystandaryzowanego DTO.
- Różne adresy i parametry – Istotną cechą, która różni się między giełdami, są także adresy zapytań do API. Ten problem rozwiązujemy poprzez ręczne wprowadzanie tych adresów do bazy danych. Następnie, podczas używania Modułu GUI adresy te są automatycznie pobierane i przesyłane do Modułu Komunikacji

Podsumowując, w celu skomunikowania się z daną giełdą tworzony jest obiekt, który w diagramie poniżej nazywa się „łącznik”. Dany łącznik tworzony jest przez klasę zarządzającą, która podejmuje pierwszy kontakt z Modułem BOT.



Do zaimplementowania tego modułu wykorzystana została biblioteka Twisted. Praca z tym oprogramowaniem okazała się zdecydowanie najprzyjemniejszą częścią zadania. Obecne działanie polega na przyjęciu komendy, którą należy wykonać. Następnie, komponujemy instancje klasy Bot wraz z komunikatorem odpowiadającym protokołowi oraz parserem odpowiadającym giełdzie, by ostatecznie pobierać dane z giełd i ewentualnie zapisywać je w bazie danych.

Zaimplementowane zostały 3 połączenia z giełdami:

- Poloniex REST – pozwala na okresowe pobieranie wartości danej pary walut z giełdy;
- Bittrex REST – pozwala pobrać dane na temat par walut dostępnych na giełdzie;
- Gemini WebSocket – wykorzystujący bibliotekę Autobahn, daje nam możliwość pobierania tych samych danych co REST-owy odpowiednik przy znacznie mniejszym opóźnieniu przesyłu danych.

Przy próbie implementacji protokołu WebSocket ujawnił się duży problem. Zarówno Twisted jak i PyQT korzystają z pętli zdarzeń jako rdzenia działania. Niestety, by obie biblioteki działały jednocześnie, wymagane jest obejście tego problemu. Trudność została rozwiązana przy pomocy biblioteki pyqt5reactor.

Jest ona wykorzystywana w następujący sposób:

1. Tworzymy instancję aplikacji Qt i tym samym jego reaktor.
2. Przed zaimportowaniem biblioteki Twisted wywołujemy funkcję `install()` z biblioteki `pyqt5reactor`. Funkcja ta tworzy własny reaktor Twisted, który może działać wspólnie z reaktorem Qt.
3. Wywołujemy funkcję `runReturn()` na reaktorze Twisted. Skutkuje to uruchomieniem prawdziwego reaktora Twisted w tle, przy dalszym poprawnym działaniu reaktora Qt.

Wspomniane wyżej rozwiązanie wymusiło zmianę sposobu wykorzystania biblioteki Autobahn. Niemożliwym okazało się użycie najprostszych funkcji `start()`, gdyż funkcja ta wyłącza reaktor po zamknięciu połączenia. Zamiast tego należało napisać, zgodnie z paradygmatami Twisted, własną klasę Factory tworzącą odpowiednie instancje klasy Protocol i następnie uruchamiać połączenie metodą `connectWS()`.

Problemem może w przyszłości być to, że Qt nie wspiera Pythonowej wersji swojego oprogramowania i przez to konflikty z reaktorem Twisted mogą się nasilać.

3. Fazy implementacji

1. Zapis obecnej wartości pary walut w bazie danych dla jednej giełdy.
2. Możliwość zapisywania obecnej wartości pary walut przez cały dzień i wizualizacja tej wartości w postaci wykresu.
3. Dołączenie dwóch kolejnych giełd.
4. Zaimplementowanie wszystkich ważniejszych danych możliwych do pobrania z giełdy
5. Możliwość zdalnego handlowania na giełdzie.
6. Prosty bot przyjmujący strategię średnich kroczących.

Powyżej zaznaczone zostały fazy zaimplementowane, częściowo zaimplementowane i nieruszone.

Obecny kurs działań przewiduje powstrzymanie się przed implementacją kolejnych funkcjonalności. Znacznie ważniejszą rzeczą jest naprawa wszelkich błędów oraz napisanie testów stabilizujących działanie programu.