Developer’s Documentation

Developer‘s Documentation of the program Cryptocurrency Algorithmic Trading Bot

Programming in C# (NPRG035), Advanced Programming in C# (NPRG038)

at MFF UK.

Author: Tomáš Pop

Date of issue: 08/07/2022

Version: 1.0

Introduction

In the developer’s documentation, you can read about algorithms, decisions made during the development process, and some implementation details included in the Cryptocurrency Algorithmic Trading Bot implemented in C#.

Used Tools

For the development, the author used Visual Studio 2022 v17.2 with .NET 6.

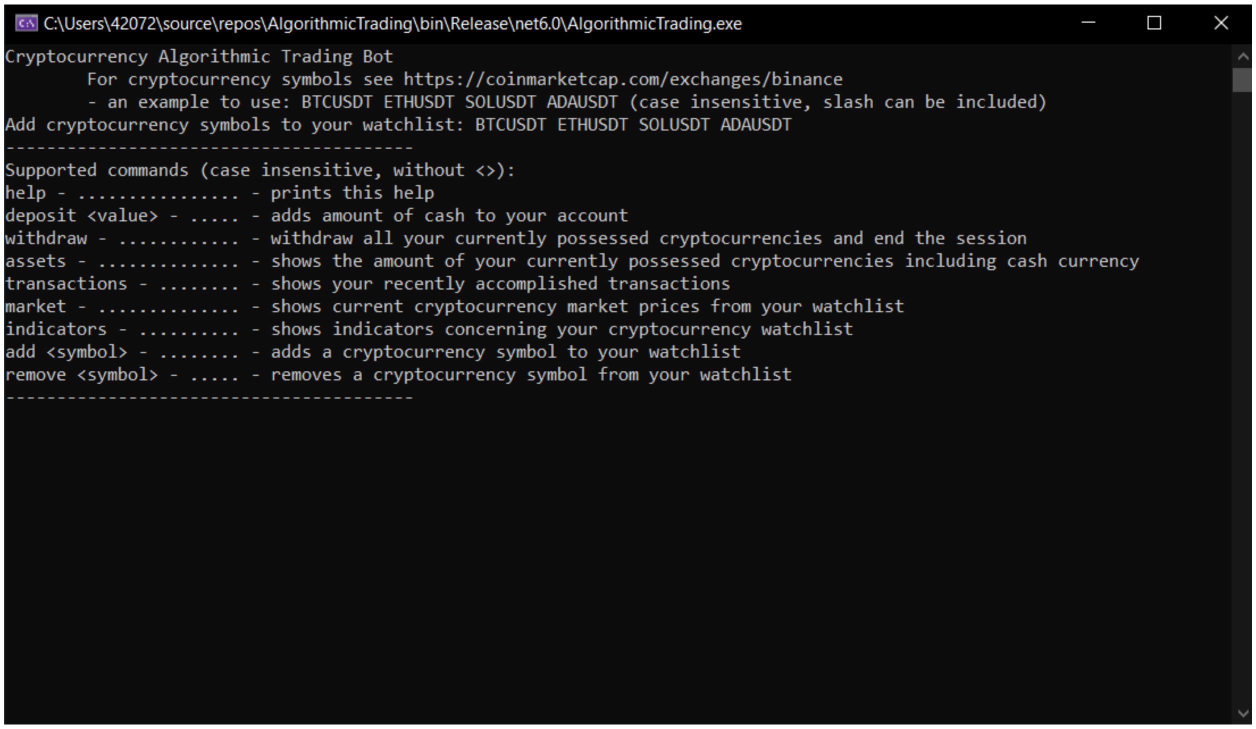
Project Structure

All necessary source files are contained in the root directory. The entry point source file is *Entrypoint.cs*. The most crucial source files are 1) Analyzer.cs which takes care of the analysis process using technical indicators, cash management, and transaction handling. 2) Connection.cs where the connection to the cryptocurrency exchange service is taken care of, and 3) InputProcessor.cs takes care of the dialog with the user. The rest of the sources mostly plays a support role for the sources mentioned earlier.

Common Workflow

At the start of the application, the user is prompted to fill in cryptocurrency symbols that he wants to have on the watchlist. Afterward, the watchlist is checked to whether it contains valid records, otherwise, the user is warned about it. For instance, if the user decides to skip filling the watchlist, he is warned about such action.

On one hand, the program periodically keeps renewing market data about the cryptocurrency symbols by running HTTP requests from the cryptocurrency exchange service provider to keep up with the latest information provided. On the other hand, the program keeps waiting for the user‘s input. The user can choose from currently supported commands, as shown in Figure 1.


C:\Users\42072\source\repos\AlgorithmicTrading\bin\Release\net6.0\AlgorithmicTrading.exe

```
Cryptocurrency Algorithmic Trading Bot
        For cryptocurrency symbols see https://coinmarketcap.com/exchanges/binance
        - an example to use: BTCUSDT ETHUSDT SOLUSDT ADAUSDT (case insensitive, slash can be included)
Add cryptocurrency symbols to your watchlist: BTCUSDT ETHUSDT SOLUSDT ADAUSDT
----------------------------------------
Supported commands (case insensitive, without <>):
help - ................ - prints this help
deposit <value> - ..... - adds amount of cash to your account
withdraw - ............ - withdraw all your currently possessed cryptocurrencies and end the session
assets - .............. - shows the amount of your currently possessed cryptocurrencies including cash currency
transactions - ........ - shows your recently accomplished transactions
market - .............. - shows current cryptocurrency market prices from your watchlist
indicators - .......... - shows indicators concerning your cryptocurrency watchlist
add <symbol> - ........ - adds a cryptocurrency symbol to your watchlist
remove <symbol> - ..... - removes a cryptocurrency symbol from your watchlist
----------------------------------------
```

Figure 1: User’s console after filling in the watchlist shows currently supported commands (a help command does the same)

Note that the program should never end uncontrollably or throw any exceptions besides of the information to the user that he has done something wrong. For example, if the user fills in an unknown, not supported the action, it is expected that the user is only warned by the console that the behavior is not supported/that the action is not available (for example, even when the number of parameters is not right as shown in the *add <symbol>*, *remove <symbol>, or deposit <value>* commands).

Invalid or inconvertible parameter values (missing or not integral), and non-existing cryptocurrency symbol addition or removal should be handled properly as well. For instance, the user may have left the transactions.csv file open, for such an issue, the user is warned that the transactions will not be recorded until closing. Moreover, if the user loses connection to the API service, he is warned that the connection to the service can’t be established.

Considering that the user uses the library otherwise with only valid operations and the correct number of parameters, the common dialog should be that the user fills in a command and receives a response in the console. The only output which is done by the worker without a previous request is a BUY/SELL signal.
If the user has enough money to accomplish the transaction, it shows at what price it was bought/sold (all is done only hypothetically since we do not fill in our own personal details nor the API key is needed, at least for Binance API). Nevertheless, the transaction records can be tracked via transactions command (a few latest ones

in case there are many transactions accomplished – not complete records in order not to keep it all in the memory and not to spam the console).
The whole transactions history in a CSV file is created which can be checked how the bot was doing. For a new run, the file is cleared and prepared for a new session.
One and the only way to end the program (besides of forcing the end by quitting completely) from the user's end point of view is by calling withdraw command.

Interesting Classes and their Data Structures

In this section, there is a brief description of some interesting classes and their data structures utilization.

Processor Class

- Takes care of the fluent dialog between the user and the rest of the program
- Does all the parsing stuff
- Transfers (delegates) parsed data to the class which should be taken care of using connection to the API service
- Commands definitions
- Data Structures
  - Options that are currently supported are defined using an enum
  - Their mapping to the string literal entities is done using a dictionary that contains the key as an enum item
  - These keys map to their real string representation including their description for a help command (Command immutable struct which has only these two string properties)
  - These options are mapped as keys in another dictionary where they are mapped to particular function delegates

IConnection Interface with its *Connection sealed classes

- IConnection – interface which establishes necessary contract for specialized classes, and establishes communication between API classes and the analyzer which is the backbone of the cryptocurrency trading bot
- Connection class
  - A generic sealed child class that acts as an adapter to the concrete *Connection
  - It communicates with the analyzer
  - It obtains a reference to the concrete service via the constructor and shifts the responsibility to the concerned class which takes care of a concrete API service connection (in this example *BinanceConnection*, nevertheless, the same construction would be done for another service analogically
  - For this purpose, a *DummyConnection* sealed class was added to show a potential further contribution to what the class should contain in order to both fulfill the contract and be compliant with the rest of the code)

- Internal specialized sealed classes which implement the interface (here BinanceConnection) take care of HTTP requests, data retrieval

Data Structures

- To store the current state of the cryptocurrency symbol, there is a dictionary that has keys as symbols, and values are Cryptocurrency structs. Cryptocurrency struct is an immutable data structure (parametric constructor, otherwise getters only) containing important information about a particular cryptocurrency

TechnicalIndicatorsAnalyzer

- A backbone of cryptocurrency trading bot
    - Buy/Sell/Hold signals handling
    - Contains money management strategy
    - Calculation of technical indicators: Bollinger bands, Relative strength index
    - Transactions management – writing to CSV file
- A backbone of cryptocurrency trading bot

Data Structures

- On some places of this source file, there can be seen utilization of Queue instead of a List, more precisely ConcurrentQueue
    - Since in this structure, there was a slight emphasis not to waste much memory since no one knows how long the user plans to keep the program running
    - Dataset – primarily required for the initial part of the program where we need to precompute initial values of technical indicators (and since the currently implemented indicators rely on the previous values, the initial values which do not have any predecessors are and need some records to get more precise)
        - It contains only a few records that are needed to compute the technical indicators – in order to keep the window rolling it was important to be able to *Enqueue* and *Dequeue* in *O(1)* for which a deque may be utilized in contrary to vector
        - Dataset is otherwise a dictionary that contains identifier symbols for keys and "matrices" as values that are ConcurrentQueue<Dictionary<string, double>>. Initially, the design was to directly store as Queue<List<double>>, nevertheless, key-value indexing seems more logical in case there were multiple technical indicators involved than remembering which indicator is located on which index
    - Transactions – to maintain whole history in memory may have been undesirable (although the program does not do that many trades) and therefore the *TechnicalIndicatorsAnalyzer* keeps only a few last records which are printed upon requested command – again

to keep a rolling window, a *Queue* proved to be definitely a better choice than *List*

    - The transactions storage is a queue that contains concrete Transaction structures (which is a data class containing important information about the transaction – cryptocurrency symbol, timestamp, exchange rate, cryptocurrency amount)

- To manage signals there is a dictionary as a counter per cryptocurrency symbol – it was important to prevent the program from mindless spamming that some cryptocurrency is supposed to be bought (if it occurs multiple times in a row there was an effort to reduce the risk that the signal might be wrong although mathematics says otherwise)
- Since the worker thread renews the data, and periodically sends update info to the analyzer while the user can launch its own command, thread-safety issues were avoided by introducing concurrent versions of data structures

Decisions Made

In this section, I would like to shift to the first-person view.

Dataset Preparation

As many technical indicators do not work exactly well at the beginning since they need to look back to compute its value, I came across Binance API which (not only) offers data with approx. 500 minutes lookback of cryptocurrency symbols in the minute interval on which I managed to better stabilize the values – therefore for each addition, an HTTP request is made, data is collected and the indicators are recomputed many times with a fixed window – afterward, they are used for the market situation assessment.

It is still kind of unpleasant since each symbol filled in by the user requires an HTTP request and JSON data parsing is required. Moreover, it did not look good from the user's point of view – if the user wanted to use a lot of cryptocurrency symbols from the very start of the program (consider that a single HTTP request with leading operations took approx. 400 ms per symbol)

- Since the only thing which is common for all the requests is that its result is supposed to be transferred to the analyzer where the dataset is built, otherwise the rest of the logic can run independently of each other, I tried to parallelize this task using PLINQ which lowered the delay drastically (especially in case the user wants to maintain a large watchlist from the beginning)

API Connection Design

Connection generic class is the entry point that is supposed to be used when using the connection to the cryptocurrency exchange service.
Connection class creates a concrete *Connection sealed class satisfying the interface contract by using this concrete sealed class, used as the adapter

- It is the communication endpoint which the Processor class (parsing user input and transferring the responsibilities further) calls for commands

To specify in a simple manner which service is supposed to be used, at the very top of the Entrypoint.cs source file, change global using Service to the service desired:

- global using Service = BinanceConnection to global using Service = CoinbaseConnection (in case this class existed and could fulfill the interface contract)

Technical Indicators

Initially, I wanted to use an external library for technical indicators, nevertheless, I decided to have no external dependencies at all and implemented my own.
I implemented two rather simple indicators, however, one of few which I know how they work (and use when I check the candlestick charts on Binance) and in combination, they are kind of useful in human cryptocurrency trading.
For demonstration purposes (since otherwise, the application is practically numb in case nothing is going on the cryptocurrency exchange)
I kept the condition of signal triggering that both indicators need to trigger the same signal multiple times in a row to ensure that the program does not spam fake signals with high intensity (which the very first one which overlaps the indicator’s boundaries probably is), on the other hand, it can suppress some not that significant signals and it can remain “numb‟ for a longer period of time (~order of minutes/tens of minutes).
To ensure that the indicators are doing the right thing, I have already implemented these and some more technical indicators in Python via pandas some time ago, this way I got gold data which I could check whether they make sense.

Strategy

I tried to keep things simple in terms of money management strategy for the bot and followed a simple rule:

- It is allowed to spend a twentieth of current USD assets and when the sell signal for a particular cryptocurrency is triggered, it flushes all these cryptocurrency assets and converts them to USD

- Since the cryptocurrencies are nontrivially correlated, I decided that if the bot triggers a BUY/SELL signal, it is supposed to occur many times in a row (for each API call which is done once a few seconds)
- This way some fake signals are filtered and it does not happen that often (it occurred almost all the time before the signal counter was introduced) that the bot would do the same decision for all cryptocurrencies on the watchlist

HTTP Requests

I used async/await approach in terms of API calls via HTTP requests – as I measured time per request, it sometimes showed rather unstable results, between 300-1300 ms (but almost always under 700ms) with a minor delay (i.e. 1 second, not to get suspended due to API calls overlap) which resulted in not catching up the data properly

- I tried stabilizing the variance by launching multiple tasks doing the HTTP requests which helped both the speed and lowered the variance significantly (by launching multiple tasks, and *Task.WhenAny(…)*)
- However, the multiplication of requests can cause issues with API calls limit, so I stuck with the single-threaded HTTP requests solution

File Writing

I did not want to let the transaction storage grow indefinitely (although the program proved to do approximately 200 transactions after approximately 8 hours I kept it running which should not be an issue) and decided to keep only recent records and full history would be found in a separated file.

If the user keeps the file open from the last run, reading and writing at the same is supposed to be suppressed and the application should not fall due to this issue

- One of the options would be to store all the runs and add a timestamp to the output file results_${time_stamp}.csv and the collisions will be gone
- I decided rather keep only a single file for writing and in case the user is currently checking previous run results (reading and writing at the same time threw exceptions), I only warn him that this run will not be recorded

Printing

I tried to completely separate printing parts from the logic of the program and during the development process, it helped with faster changes. Moreover, I tried to abstract the variables from the code and keep them together to simplify the development process.

Encountered Pitfalls

I would like to describe some of the issues which I encountered and hopefully managed to solve

- It is necessary to *deposit* before the bot starts doing something (besides shouting that when the signal is triggered that it either does not have the particular cryptocurrency symbol to sell or that it does not have any cash) – many times I left it running and then realized that it only spammed these messages
- Very slow behavior when the user wanted to *withdraw*
    - It took me a while to come up with something which would manage threads fast (initially, when the user sent a withdraw command, the user would have to wait until the worker thread woke up) – conditional variables solved this issue by *Monitor.ReleaseAll()*
    - In the earliest phases, I tried using a single thread for both (which was terrible) with a much smaller delay than it is in the release version which resulted in a temporary API service suspension for my IP address
- One of the unpleasant discoveries was that the bot programmed (in the earlier phases of development) like that either trigger signals for everything or for nothing (and if it did, it kept buying and buying until it had no money to spend)
    - I tried to regulate this issue by creating a counter mentioned earlier
- The signals do not occur that often and it takes some time before anything shows up (I first tested the functionality via generating random signals on purpose to see whether the processing works fine and for the real run I checked the signals whether they are triggered at right moments I had candlestick charts from Binance open for the comparison)

Potential Improvements

It is important to mention that the strategy picked is far from ideal and in case this software was used as a serious tool, it would be ideal to separate the strategy completely away from the analyzer class and implement something more sophisticated (which is a rather daunting task and not the goal of this project). At this point, it is a useful tool in case there were no deposit/withdraw/transaction fees involved. Nevertheless, the signals are either way too insignificant that it can’t weigh the fees, or that the program simply runs way too short a time to make up the fees.

Consequently, the project would be nicer (but also a lot more laborious) if it had a GUI and supported ideally graphs with the technical indicators. Nevertheless, I am not entirely fond of GUI applications, so I made a compromise to add colors to the console for buying, selling, and warnings and it seemed to be satisfactory too.