Stock Trades API

OVERVIEW

You are part of a team building a brokerage company's accounts and trading management platform. One requirement is for a REST API service to manage trades using the Nodejs (Typescript) Fastify framework. You will need to add functionality to add and delete transactions as well as to perform some queries. You'll be dealing with typical information for financial transactions like account ownership data, shares, price, transaction time, etc. The team has come up with a set of requirements including filtering and ordering requirements, response codes and error messages for the queries you must implement.

The definitions and a detailed requirements list follow. You will be evaluated on whether your application performs data retrieval and manipulation based on given use cases exactly as described in the requirements.

Each trade is a JSON entry with the following keys:

- `id`: The unique ID of the trade.
- `type`: The trade type, either *buy* or *sell*.
- `user`: The user responsible for the trade, a JSON entry:
    - `id`: The user's unique ID.
    - `name`: The user's name.
- `symbol`: The stock symbol.
- `shares`: The total number of shares traded. The traded shares value is between *10* and *30* shares, inclusive.
- `price`: The price of one share of stock at the time of the trade (up to two places of the decimal). The stock price is between *130.42* and *195.65* inclusive.
- `timestamp`: The timestamp for the trade creation given in the format `yyyy-MM-dd HH:mm:ss`. The timezone is `EST(UTC -4)`.

Sample JSON trade data object

```
{
   "id": 1,
   "type": "buy",
   "user": {
      "id": 1,
      "name": "David"
   },
   "symbol": "AC",
   "shares": 28,
   "price": 162.17,
   "timestamp": "2014-06-14 13:13:13"
}
```

You should complete the given incomplete project so that it passes all the test cases when running the provided unit tests. The project by default supports the use of SQLite3 database, but you can use MongoDB or Postgres and it would be graded additionally.

The *REST* service should implement the following functionalities:

1. *Erasing all the trades*: The service should be able to erase all the trades by the *DELETE* request at /erase. The *HTTP response* code should be *200*.
2. *Adding new trades*: The service should be able to add a new *trade* by the *POST* request at /trades. The event *JSON* is sent in the request body. If a trade with the same id already exists then the *HTTP* response code should be *400*, otherwise, the response code should be *201*.
3. *Returning all the trades*: The service should be able to return the JSON array of all the trades through a *GET* request at /trades. The *HTTP* response code should be *200*. The JSON array should be sorted in ascending order by trade ID.
4. *Returning the trade records filtered by the user ID*: The service should be able to return the *JSON* array of all the trades which are performed by the user ID through a *GET* request at /trades/users/{userID}. If the requested user does not exist then the *HTTP* response code should be *404*, otherwise, the response code should be *200*. The JSON array should be sorted in ascending order by trade ID.

5. *Returning the highest and lowest price for the stock symbol in the given date range*: The service should be able to return the *JSON* object describing the information of highest and lowest price in the given date range specified by start date and end date inclusive, through a *GET* request at `/stocks/{stockSymbol}/price?start={startDate}&end={endDate}`. If the requested stock symbol does not exist then the HTTP response code should be *404*, otherwise, the response code should be *200*. The response JSON should consist of the following three fields:
    - `symbol`: the symbol for the requested stock
    - `highest`: the highest price for the requested stock symbol in the given date range
    - `lowest`: the lowest price for the requested stock symbol in the given date range

    If there are no trades for the requested stock symbol, then the response JSON should be:

    ```
    {
      "message":"There are no trades in the given date range"
    }
    ```

6. *Returning the fluctuations count, maximum daily rise and maximum daily fall for each stock symbol for the period in the given date range:* The service should be able to return the fluctuations count, maximum daily rise and maximum daily fall for each stock symbol for the period in the given date range inclusive. This will be accomplished through a *GET* request at `/stocks/stats?start={startDate}&end={endDate}`. The response code should be *200*. The JSON array should be sorted in ascending order by the stock symbol. The response JSON should consist of the following four fields:
    - `symbol`: The stock symbol for the requested stock.
    - `fluctuations`: This field describes the number of fluctuations, or reversals, in stock price for the given date range. If there are not at least *3* data points, this value should be *0*. A fluctuation is defined as:
        - There is a daily rise in price followed by a daily fall in price.
        - There is a daily fall in price followed by a daily rise in price.
    - `max_rise`: This field is the maximum daily price increase for the period.
    - `max_fall`: This field is the maximum daily price decline for the period.

    If there are no trades for the requested stock symbol, then the response JSON should consist of the following two fields:

    - `symbol`: This is the requested stock symbol.
    - `message`: The message should be "There are no trades in the given date range".

You should complete the given incomplete project so that it passes all the test cases when running the provided unit tests. The project by default supports the use of SQLite3 database, but you can use MongoDB and it would be graded additionally.

Sample Series of Requests

Requests are received in the following order:

- **POST** `/trades`

Examples of request could be found in the examples folder inside the project template.

- **GET** `/trades/users/1`

Examples of request could be found in the examples folder inside the project template.

- **GET** `/stocks/ACC/price?start=2014-06-25&end=2014-06-26`

The response of the GET request is the following JSON with the HTTP response code 200.

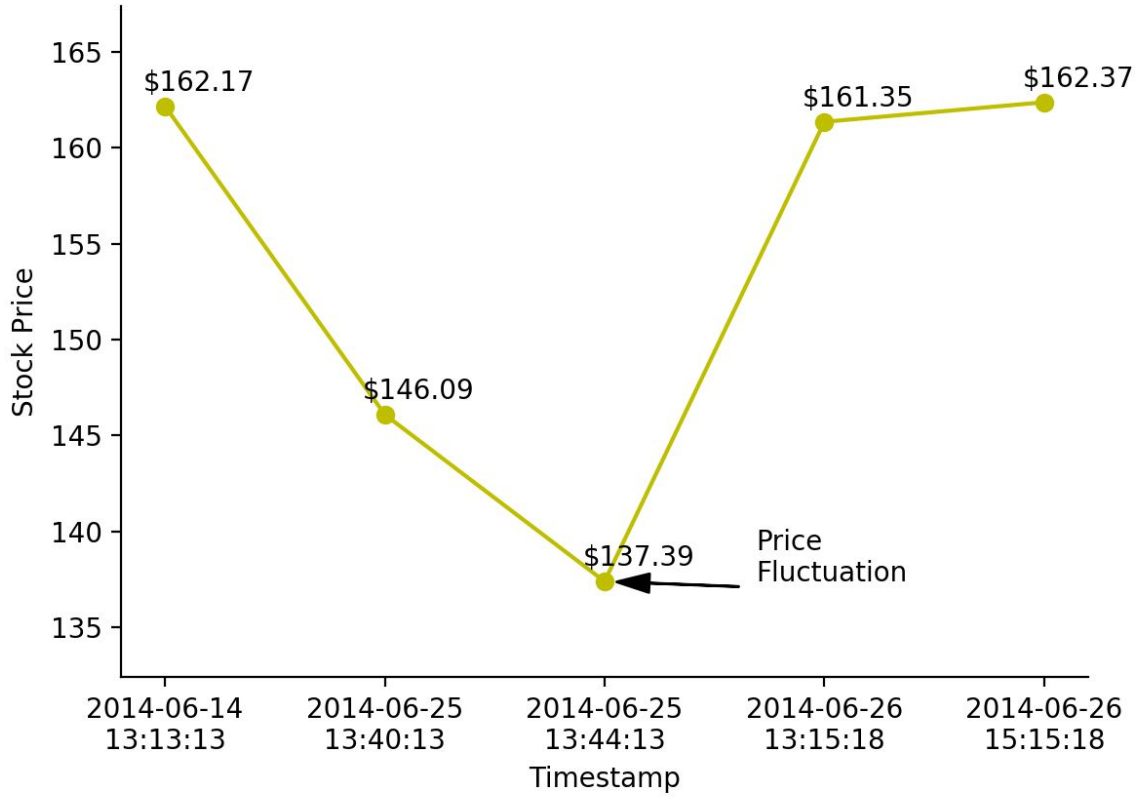
```
{
    "symbol": "ACC",
    "highest": 146.09,
    "lowest": 146.09
}
```

- **GET** `/stocks/stats?start=2014-06-14&end=2014-06-26`

The response of the GET request is the following JSON array with the HTTP response code 200:

```
[
    {
        "stock": "ABR",
        "message": "There are no trades in the given date range"
    },
    {
        "stock": "AC",
        "fluctuations": 1,
        "max_rise": 23.96,
        "max_fall": 16.08
    },
    {
        "stock": "ACC",
        "fluctuations": 0,
        "max_rise": 0.0,
        "max_fall": 0.0
    }
]
```

The following stock price plot describes the fluctuations for the stock symbol AC in the given date range:



Stock Price for symbol AC

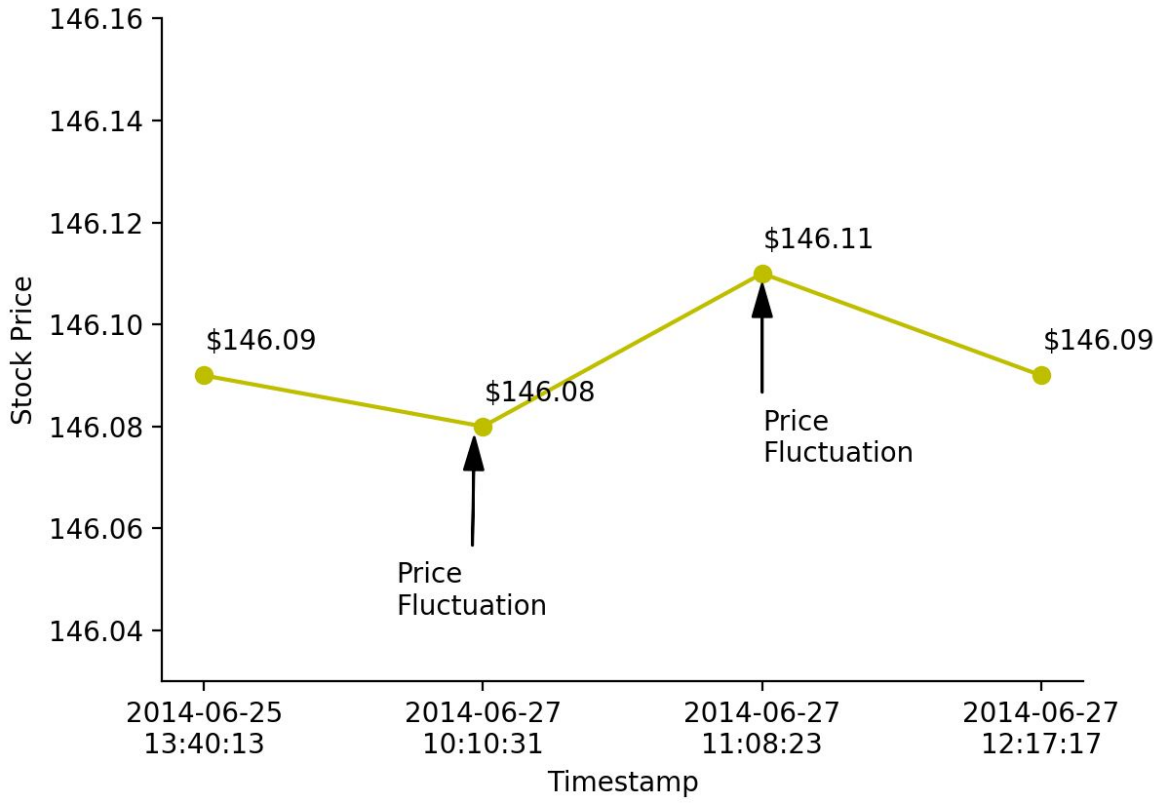
- **GET** `/stocks/stats?start=2014-06-14&end=2014-06-27`

The response of the GET request is the following JSON array with the HTTP response code 200:

```
[
    {
        "stock": "ABR",
        "message": "There are no trades in the given date range"
    },
    {
        "stock": "AC",
        "fluctuations": 1,
        "max_rise": 23.96,
        "max_fall": 16.08
    },
```

```
    {
        "stock": "ACC",
        "fluctuations": 2,
        "max_rise": 0.03,
        "max_fall": 0.02
    }
]
```

The following stock price plot describes the fluctuations for the stock symbol ACC in the given date range:



Stock Price for symbol ACC

Examples of the data could be found inside tests folder.

NICE TO HAVE

- Dockerfile to bring the application up and running with MongoDB or Postgres (if used so);
- All necessary or relevant information are in README.MD file;
- REST API should handle error cases and provide a consistent response (format, error structure, headers, etc.)
- Included tests are passing;

Implementation of the nice to have points would be not decisive during evaluation.

WHAT WOULD BE CHECKED

- Usage of framework-specific techniques
- Clean implementations, performance, and maintainability;
- Proper separation of concerns
- Production-ready code;
- Proper usage of Typescript

Feel free to ask questions!