Coding Challenge - Order Management

23 February 2024 17:54

This code challenge is an small showcase of your coding and architecture skills.
Your choices and potential enhancements will be discussed during the interview process.

The task is to build an Order Management system able to calculate instrument prices and trade with clients.

An Order, identified by an unique numerical identifier, represents a request to buy or sell a certain amount of a financial instrument, identified by a symbol, at a certain price.

orderId	symbol	side	amount	price
1	JPM	Buy	20	20
2	GOOG	Sell	12	25
3	AMZN	Sell	7	10
4	JPM	Buy	10	21

For example, orderId: 1 means that a trader wants to buy 20 JPM shares for a price of 20 each.

An "order book" is a collection of orders for a specific symbol which are sorted by its price.
It contains two sides: buy and sell. Orders with the same symbol can come from many different sources, with different prices and amounts. During the day, multiple orders can be added or removed from the order book as well as modified several times.

Part 1

Implement an order management application that exposes a REST API capable of **adding**, **removing** and **modifying** orders in the system.

The order management system should also be able to **calculate the best price** for a given symbol and amount. The best Buy price is the one of the order with smallest price in the book, multiplied by the requested amount. If the amount of the order is not enough, then the next smallest priced order of the book should be added to the calculation and the price calculated using the rest of the amount that could not be covered by the previous order:

```
calculatePrice("JPM", Buy, 20) = $20 * 20               = $400.00
calculatePrice("JPM", Buy, 10) = $20 * 10               = $200.00
calculatePrice("JPM", Buy, 22) = $20 * 20 + $21 * 2     = $442.00
```

The best Sell price is calculated in a similar way, but the first order to be considered should be the one with highest price instead of the lowest.

Finally, if a client is happy with the price, the system should **place** the trade. When a trade is placed, the application should subtract the amount bought or sold to the client from each order. Empty orders, where amount is equal to zero, should be removed from the order book.

Observations

- Assume that data provided to the system will always be valid: no checks for null or negative numbers is required.
- In order to sustain high performance, the application should maintain some state and avoid constantly sorting the orders.
- The interface of the application should be equivalent to the following Java interface definition:

```
enum Side {
  Buy,
  Sell
}

interface OrderManager {
  void addOrder(int orderId, String symbol, Side side, int amount, int price);
  void removeOrder(int orderId);
  void modifyOrder(int orderId, int amount, int price);
  int calculatePrice(String symbol, Side side, int amount);
  void placeTrade(String symbol, Side side, int amount);
}
```

- The application can be implemented in the programming language of your preference;

Part 2

Design the infrastructure required to run the Order Management application on a Public Cloud Service Provider (AWS, Azure, GCP) while covering the following requirements.

- A persistent data layer/database for Orders and Trades;
- Online load of Orders from a Blob/Object store;
- API must be exposed to internet;
- The application must be available 24/7 and updates done with zero downtime;
- The application must be able to scale in and out, depending on the load;
- API must return a response within 5ms;

An architecture diagram and supporting documentation should be prepared and made available via a public Source Code Management system (Github, Gitlab, Bitbucket) along with the application source code.