Order API

To buy or sell an instrument/stock, a buyer places a bid order in the exchange with the desired quantity(to buy) and a maximum price(bid price) at which he/she is willing to buy, whereas a seller places an ask order with desired quantity(to sell) and a minimum price(ask price) at which he/she is willing to sell.

When a bid offer/order matches an ask offer/order, a trade takes place. This is the core working of an exchange matching engine which accepts Requests[to place(create), to modify(update), to cancel(delete)] an order, and matches them to create a transaction. It uses an order book data structure.

What is a limit order book?

For this task, we'll simulate an order api using which we can place, modify, or cancel an order. We'll be able to get snapshots of currently placed orders, as well as all the trades that have taken place so far. You'll implement a data structure that maintains a limit order book and takes actions as and when a request arrives.

Order matching means that a bid and ask are compatible for a trade to take place. The traded price is the one that's already present in the book, meaning if a bid comes that matches the best ask then the ask price is taken for trade price. Similarly if an ask comes that matches the best bid then the bid price is the traded price.

📘Functionalities of the API

1. Provide a CRUD interface to Place, Modify, Cancel, and Fetch an order.
    a. Place[POST]: Takes a `quantity(int, >0)` and a `price(float, >0, multiples of 0.01)` and `side[1(buy) or -1(sell)]`, inserts the order on either bid or ask side of the book depending upon the `side`, and returns back an `order_id`.
    b. Modify[PUT]: Takes `order_id`, `updated_price` and updates the order book. returns `success: bool`.
    c. Cancel[Delete]: Takes `order_id`, cancels the order if not yet traded. In case of partially traded, cancels the remaining quantity. returns `success: bool`
    d. Fetch[Get]: Takes `order_id`, and returns the `order_price`, `order_quantity`, `average_traded_price`, `traded_quantity`, `order_alive: bool`
    e. All orders[Get]: Returns all the orders placed. Each order information contains fields described in `1d`
    f. All trades[Get]: Returns all the trades taken place. Each trade object contains a `unique_id`, `execution_timestamp`, `price`, `qty`, `bid_order_id`, `ask_order_id`.
2. Provide a websocket that sends an update whenever a trade takes place, with information specified in `1f`.
3. Provide a websocket that sends a snapshot of the order book with 5 levels of both bid and ask depth, every second. Each depth level should contain `price`, `quantity`.

📝Deliverables

1. You are required to write the implementation in a python based framework. The description is open ended as we'd like to judge your design decisions as well. Though, you're free to ask as many questions as you want. The focus of the task is to judge your programming abilities and your decision-making framework.
2. You're expected to follow standard coding practices, i.e., writing neat, modular, and well-commented code. Try to include documentation highlighting your design decision and overall architecture. How you

have broken the application in microservices, and what's the functionality of each service necessary for the OrderAPI. Data flow across different components with labels.
3. Provide either a postman collection or a simple webpage to test out the application.

Nice to have (Bonus Points)

1. You break down the application into a couple of microservices.
2. Can containerize the application and its services and provide a `docker-compose` yaml file with instructions on how to run it, and interact with the services.
3. As in any production level application, in the event of a crash you should be able to replicate the state of the application with a restore mechanism.

☁️Thoughts

Every operation that you need to do on an order book can be done in amortized O(1) time. And if you can remove disk/network access from the critical path of those operations, that'd give maximum throughput. The objective is to match orders as quickly as possible. Everything else can be done with multiple processes and horizontal scaling, but order matching has to be sequential.