# Golang Backend Developer Task: Order Matching Engine

## Problem Statement

You are tasked with designing and implementing a simplified order matching system, similar to a stock exchange's core matching engine. This system must accept buy and sell orders over a RESTful API, match incoming orders against existing ones based on price and time priority, execute trades, and maintain the resulting state in a MySQL database. The system should support market orders (execute immediately at the best available price) and limit orders (specify a price and wait to be matched). Orders that are not immediately matched should be kept pending in the order book. For example, a new buy limit order is matched against the lowest-price sell orders; a new sell limit order is matched against the highest-price buy orders. Partial fills are allowed: if an order is only partially filled, its remaining quantity should stay on the book (for limit orders) or be canceled (for market orders) once no further match is possible. You have 48 hours to complete this task. Focus on correctness and clear design rather than high-performance optimizations. No prior codebase is provided; you should start the project from scratch.

## Functional Requirements

- **Order Types:** Support both Limit Orders and Market Orders for Buy and Sell sides.
- **Order Matching:** Implement a matching algorithm that follows price-time priority: ○ Match orders at the best price first (highest bid vs. lowest ask).
  - ○ Among orders at the same price level, match the oldest order first (FIFO). ○ When a match occurs, record a trade (with matched quantity and price). Use the resting (existing) order's price for limit/limit matches, and the limit price for market/limit matches.
  - ○ Update or remove orders from the order book as they are filled. Partially filled limit orders should remain with the remaining quantity; remaining market order quantity should be canceled once no more matches are possible.
- **Operations (via REST API):** Implement endpoints for the following operations (suggested routes are examples):
  - ○ **Place Order:** POST /orders – Submit a new order. Request body (JSON) might include fields like symbol, side ("buy" or "sell"), type ("limit" or "market"), price (for limit), and quantity.
  - ○ **Cancel Order:** DELETE /orders/{orderId} – Cancel an existing pending order by its ID. Only pending (unfilled or partially filled) orders can be canceled.
  - ○ **Query Order Book:** GET /orderbook?symbol={symbol} – Retrieve the current state of the order book for a symbol, including top bids and asks or full lists of active orders.
  - ○ **List Trades:** GET /trades?symbol={symbol} – (Optional) List all executed trades.

    - **Get Order Status:** GET /orders/{orderId} – (Optional) Retrieve the status/details of a specific order (filled, pending, canceled, remaining quantity, etc.).
    - You may design the exact request/response schema, but ensure consistency (use JSON, proper HTTP status codes, etc.). Provide examples in your documentation.
- **Persistence:** Use a MySQL-compatible database to store all persistent data. At minimum, you should have tables for orders and trades (or executions). Each order record should track fields like order ID, timestamp, symbol, side, type, price (if limit), initial quantity, remaining quantity, and status (e.g., “open”, “filled”, “canceled”). Each trade record should log details of matched orders (e.g., buy order ID, sell order ID, executed price, quantity, timestamp). Ensure that all relevant state (open orders and completed trades) is saved in the database. (Bonus: You may use TiDB in place of MySQL if you are familiar with it, as it is MySQL-compatible.)
- **Error Handling & Edge Cases:** Handle cases such as orders with no available match (limit orders go to book, market orders fill partially then cancel), invalid inputs (e.g., negative quantity), and attempts to cancel non-existent or already-filled orders. Responses should use appropriate HTTP status codes (e.g., 400 for bad requests, 404 for not found).

**Technical Expectations**

- **Language:** Golang (latest stable version). Do not use other programming languages. Use Go modules for dependency management.
- **REST API:** You may use any Go HTTP framework (e.g. net/http, Gin, Echo) or write plain handlers. All APIs should consume and produce JSON. Document your endpoints and data formats clearly (you may include examples of JSON request/response bodies in your README).
- **Data Model:** Design appropriate tables in MySQL for orders and trades. Use primary keys, indexes, and foreign keys where logical. Provide SQL schema or migration scripts in your submission. **Important note: DO NOT USE ANY ORM. ONLY USE RAW SQL**
- **Matching Logic:** Implement the matching engine in Go. You can maintain in-memory order books and synchronize them with the database state. Ensure that when a new order arrives, matching and database updates happen atomically (consider transactions if needed). Correctness of matching (price/time priority) is critical.
- **Clean Code:** Write clear, maintainable Go code. Organize packages/modules sensibly (e.g. api, models, service packages, etc.). Include comments/docstrings for non-obvious logic. Use idiomatic error handling.
- **Documentation:** Provide a README with instructions to build and run your application. This should cover:
    - Dependencies and setup (e.g., how to install Go modules, set up/configure MySQL or TiDB).
    - Database initialization (schema creation).
    - How to start the server.
    - Examples of how to call the API (e.g., using curl or http with sample JSON).
    - Any assumptions or design decisions you made.

**Submission Guidelines**

- **Deliverables:** Submit your solution as a public Git repository URL (preferred). Git Repository name should be: "**GOLANG ORDER MATCHING SYSTEM**". Ensure all source code and documentation are included.
- **Running the Code:** We should be able to clone or extract your submission, follow your README, set up the database, and run the service to test it.
- **Time Frame:** Remember this is a take-home assignment with a 48-hour deadline. Plan your time to implement core functionality first (placing and matching orders), then add enhancements.

Focus on building a correct and maintainable order-matching API in Go within the given time. Good luck!