Matching Engine

This is a high-performance cryptocurrency matching engine. This engine implements core trading functionalities based on REG NMS-inspired principles of price-time priority and internal order protection. Additionally, the engine generates its own stream of trade execution data.

- FastAPI (HTTP + WebSocket)
- SortedDict for orderbook price levels
- asyncio-friendly WebSocket broadcast manager

Run:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
uvicorn app.main:app
```

Run tests:

```
python -m pytest -v
```

1. System Architecture and Design Choices

The Matching Engine is developed using FastAPI to provide a high-performance asynchronous web API. It employs a modular architecture, separating API endpoints, and order matching logic. The architecture prioritizes low latency, scalability, and maintainability, ensuring smooth integration with other trading services.

WebSocket is used for real-time data transmission

2. Data Structures Used for the Order Book

The order book utilizes `SortedDict` from the `sortedcontainers` Python module to maintain sorted order of price levels.

Bids (buy orders): Stored in descending order (highest price first).
Asks (sell orders): Stored in ascending order (lowest price first).

Each price level contains a `PriceLevel` object that holds a deque of `Order` instances and their total quantity. This structure ensures O(log n) access for price lookup and O(1) queue operations for order matching.

```
@dataclass
class PriceLevel:
    price: Decimal
    total_qty: Decimal
    orders: Deque[Order]

class OrderBook:
    def __init__(self, symbol: str):
        self.symbol = symbol
        # Ascending order of prices
        # bids: use SortedDict where highest key is best bid
        self.bids: SortedDict[Decimal, PriceLevel] = SortedDict()
        # asks: lowest key is best ask
        self.asks: SortedDict[Decimal, PriceLevel] = SortedDict()
```

3. Matching Algorithm Implementation Details

The matching process follows price-time priority rules:
1. For a BUY order: Match with the lowest-priced ASK that is ≤ order price.
2. For a SELL order: Match with the highest-priced BID that is ≥ order price.
3. Orders at each price level are processed in FIFO (first-in, first-out) order.
4. Partial matches are allowed — if an order is partially filled, the remaining quantity stays on the book.
5. If no matches occur and the order type is `LIMIT`, the remaining portion is added to the book.
The algorithm ensures fairness and deterministic execution using consistent iteration over sorted price levels.

```
def match_order(order: Order, ob: OrderBook) -> List[Trade]:
    logger.info(f"Matching order: {order}")
    trades: List[Trade] = []
    remaining = order.quantity

    if order.side == Side.BUY:
        opposing = ob.asks
        # iterate ascending
        price_items = list(opposing.items())
    else:
        opposing = ob.bids
        price_items = list(opposing.items())[::-1]

    def price_ok(price: Decimal) -> bool:
        if order.order_type == OrderType.MARKET:
            return True
        if order.side == Side.BUY:
            return price <= order.price
        else:
            return price >= order.price

    for price, level in price_items:
        if remaining <= 0:
            break
        if not price_ok(price):
            break
        while level.orders and remaining > 0:
            maker = level.orders[0]
            match_qty = min(remaining, maker.quantity)
            trades.append(_emit_trade(order.symbol, price, match_qty,
order.side, maker.id, order.id))
            maker.quantity -= match_qty
            level.total_qty -= match_qty
            remaining -= match_qty
            if maker.quantity == 0:
                level.orders.popleft()
        if level.total_qty == 0:
            try:
                del opposing[price]
            except KeyError:
                pass

    # rest on book if limit and remaining
    if remaining > 0 and order.order_type == OrderType.LIMIT:
        rest_order = Order(
            symbol=order.symbol,
            side=order.side,
            order_type=order.order_type,
            quantity=remaining,
            price=order.price,
            timestamp=order.timestamp,
            id=order.id
        )
        ob.add_limit_order(rest_order)

    if len(trades) > 0:
        logger.info(f"Trade successful: {trades}")

    return trades
```

4. API Specifications

Endpoints:

- POST `/orders` -> Submit orders. The data should go in the following JSON format.

```
{
 "symbol":"BTC-USDT",
 "side":"sell" | "buy",
 "order_type":"market" | "limit" | "ioc" | "fok",
 "quantity":10,
 "price": 900
}
```

- WebSocket `/ws/trades` -> successful trade messages. Would emit an event upon successful trade.

- WebSocket `/ws/market` -> Market Data Dissemination. Would emit events after there is a change in order book. This feed includes:
    - Current BBO
    - Order book depth (e.g., top 10 levels of bids and asks)

Sample orders

You can post orders to the server by running `client.js` file. The code generates a series of request with random order types, prices and quantities.

Run

```
node client.js
```

Sample outputs

- POST `/orders`

```
{
  "status": "ok",
  "trades": [
    {
      "trade_id": "98168ce0-3aef-4eab-b82b-63302e3ded06",
      "timestamp": "2025-10-26T10:28:34.973435Z",
      "symbol": "BTC-USDT",
      "price": "900",
      "quantity": "10",
      "aggressor_side": "sell",
      "maker_order_id": "78b97b91-5018-43c3-a6cd-9e5fa1335118",
      "taker_order_id": "8ad252d2-bd0b-4d2b-819b-89d87404eace"
    }
  ]
}
```

- `/ws/trades`

```
{
  "trade_id": "1c61dd7c-faf6-461e-8675-ef0d55fd2699",
  "timestamp": "2025-10-26T12:58:02.578019Z",
  "symbol": "BTC-USDT",
  "price": "35034.45",
  "quantity": "181.74",
  "aggressor_side": "sell",
  "maker_order_id": "fd35fd5a-57fd-4e50-8c21-359e500053da",
  "taker_order_id": "7fc7135e-b459-49f3-aa05-0e18b219da81"
}
```

- `/ws/market`

```
{
    "type":"bbo",
    "symbol":"ETH-USDT",
    "best_bid":"784.26",
    "best_ask":"16020.53"
}
```

```
{
  "type": "depth",
  "timestamp": "2025-10-26T13:00:08.481831Z",
  "symbol": "XRP-USDT",
  "bids": [],
  "asks": [
    ["40989.7", "691.67"],
    ["96130.76", "665.03"]
  ]
}
```

Logging

The logs are consoled and persisted into a file as well `logs/app.log`. We can extend it to store log in multiple files in order to make files easy to handle.

5. Trade-off Decisions Made During Development

SortedDict vs Heap: SortedDict provides better visibility across all price levels, whereas a heap only gives O(1) best-price access but lacks efficient iteration for multiple levels.

Single-threaded Matching: Ensures atomic execution for correctness, trading off some parallel performance.

FastAPI vs Flask: FastAPI was chosen for asynchronous I/O performance and better developer experience via type hints and OpenAPI support.