Assignment for Backend position

Introduction

In this assignment, you will build a straightforward matching engine. A matching engine is a technology that lies at the core of any exchange.
From a high-level perspective, a matching engine matches people (or organizations) who want to buy an asset with people who want to sell an asset.
We want to support only one type of order for this exercise, namely limit order. A limit order is an order to buy an asset at no more than a specific price or sell an asset at no less than a specific price.
In real life, different algorithms can be used to match orders. Here we expect you to implement a continuous trading Price/Time algorithm (aka FIFO).
This algorithm ensures that all orders at the same price level are filled according to time priority; the first order at a price level is the first order matched.
For any new order, you prioritize the opposite order with the best price, and if you have multiple orders with the same price, the earliest takes precedence.

Usually, the term order book is used to list all active buy and sell orders. For example, consider such an order book:

ID	Direction	Time	Amount	Price	Amount	Time	Direction
3				10.05	40	07:03	SELL
1				10.05	20	07:00	SELL
2				10.04	20	07:02	SELL
5	BUY	07:06	40	10.02			
4	BUY	07:05	20	10.00			
6	BUY	07:10	40	10.00			

If a new limit order buy 55 shares at 10.06 price comes in, then it will be filled in this order:

1. 20 shares at 10.04 (order 2)
2. 20 shares at 10.05 (order 1)
3. 15 shares at 10.05 (order 3)

This leaves the order book in the following state:

ID	Direction	Time	Amount	Price	Amount	Time	Direction
3				10.05	25	07:03	SELL
5	BUY	07:06	40	10.02			
4	BUY	07:05	20	10.00			
6	BUY	07:10	40	10.00			

NB: order 3 is executed only partially.

Requirements

Implement the following HTTP API in Java 21 and Spring Boot 3.4.
Don't implement any persistence. All state will be lost after restart. That's okay.
Don't add support for different users.
Please keep your assignment in a private git repo and don't share it anywhere publicly.
Feel free to use any 3rd party libraries, but please add proper justification for the decision.

Place an order

`POST /orders`
It is used to place a new limit order. It responds with the current state of the just created order. It means that the order can be already filled, but it can still be pending. Every order can be filled by one or many trades.
Trade is a fact of finding matching counterparty order, order of the opposite direction with satisfying price and amount.

For example, taking our previous example, order buy 55 shares at 10.06 price was filled by three trades:

1. 20 shares at 10.04 (order 2)
2. 20 shares at 10.05 (order 1)
3. 15 shares at 10.05 (order 3)

Endpoint must accept the JSON with the following fields:

- asset - string, asset name, for simplicity this can be any text
- price - number, a price for limit order
- amount - number, amount of asset to fill by order
- direction - string, can be either "BUY" or "SELL"

And respond with the JSON containing the following fields:

- id - number, you need to generate a unique order ID
- timestamp - string, date and time when the system registered order
- price - number, the same as request body
- amount - number, the same as request body
- direction - string, the same as request body
- pendingAmount - number, amount still to be filled
- trades - an array of trade objects (see definition below), can be empty

Trade object:

- orderId - number, counterparty order ID
- amount - number, amount filled from counterparty order
- price - number, price used for this trade

Request body example:

```
{
   "asset": "TST",
   "price": 10.0,
   "amount": 100.0,
   "direction": "SELL"
}
```

Response body example:

```
{
   "id":2,
   "timestamp":"2021-12-08T10:22:00.460575730Z",
   "asset":"TST",
   "price":10.0,
   "amount":100.0,
   "direction":"SELL",
   "trades":[
      {
         "orderId":0,
         "amount":10.0,
         "price":10.0
      }
   ],
   "pendingAmount":90.0
}
```

Get current order state

`GET /orders/{orderId}`

Responds with the current state of the order with ID orderId (see previous endpoint response body).

Response body example:

```
{
   "id":2,
   "timestamp":"2021-12-08T10:22:00.460575730Z",
   "asset":"TST",
   "price":10.0,
   "amount":100.0,
   "direction":"SELL",
   "trades":[
      {
         "orderId":0,
         "amount":10.0,
         "price":10.0
      }
   ],
   "pendingAmount":90.0
```

```
}
```

Example

Imagine that we are going to use this system to emulate cryptocurrency exchange.
For example, let's first place a limit order selling 1 Bitcoin at the price no less than 43,251.00 euro.
In order to do that we need to call the endpoint `POST /orders` with the following payload:

```
{
   "asset": "BTC",
   "price": 43251.00,
   "amount": 1.0,
   "direction": "SELL"
}
```

Assuming that the Bitcoin order book is currently empty we will receive the following response:

```
{
   "id":0,
   "timestamp":"2021-12-08T13:34:44.498775730Z",
   "asset":"BTC",
   "price":42251.00,
   "amount":1.0,
   "direction":"SELL",
   "trades":[],
   "pendingAmount":1.0
}
```

Now let's place some buy orders. Let's first start with an order to buy 0.25 Bitcoin at the price of no more than 43,250.00.

`POST /orders`

```
{
   "asset": "BTC",
   "price": 43250.00,
   "amount": 0.25,
   "direction": "BUY"
}
```

Since we currently don't have a sell order with price satisfying this buy order, it just ends up in our order book.

Now let's place an order to buy 0.35 Bitcoin at the price of no more than 43,253.00.

`POST /orders`

```
{
   "asset": "BTC",
   "price": 43253.00,
   "amount": 0.35,
   "direction": "BUY"
}
```

In this case we have a matching order in our book, we can sell 0.35 Bitcoin for 43,251.00, which is lower than 43,253.00.

Current state of order #0:

`GET /orders/0`

```
{
   "id":0,
   "timestamp":"2021-12-08T13:34:44.460575730Z",
   "asset":"BTC",
   "price":42251.00,
   "amount":1.0,
   "direction":"SELL",
   "trades":[
      {
         "orderId":2,
         "amount":0.35,
         "price":43251.00
      }
   ],
   "pendingAmount":0.65
}
```


Current state of order #2:

`GET /orders/2`


```
{
   "id":2,
   "timestamp":"2021-12-08T13:34:50.460575730Z",
   "asset":"BTC",
   "price":42253.00,
   "amount":0.35,
   "direction":"BUY",
   "trades":[
      {
         "orderId":0,
         "amount":0.35,
         "price":43251.00
      }
   ],
   "pendingAmount":0.00
}
```


Let's place one last buy order to fully fill our initial sell order.

`POST /orders`


```
{
   "asset": "BTC",
   "price": 43251.00,
   "amount": 0.65,
   "direction": "BUY"
}
```


Now both orders #0 and #3 are fully executed.

`GET /orders/0`


```
{
   "id":0,
   "timestamp":"2021-12-08T13:34:44.460575730Z",
   "asset":"BTC",
   "price":42251.00,
   "amount":1.0,
   "direction":"SELL",
```

```
   "trades":[
      {
         "orderId":0,
         "amount":0.35,
         "price":43251.00
      },
     {
         "orderId":3,
         "amount":0.65,
         "price":43251.00
      }
   ],
   "pendingAmount":0.00
}
```

`GET /orders/3`

```
{
   "id":3,
   "timestamp":"2021-12-08T13:40:07.460575730Z",
   "asset":"BTC",
   "price":42251.00,
   "amount":0.65,
   "direction":"BUY",
   "trades":[
      {
         "orderId":0,
         "amount":0.65,
         "price":43251.00
      }
   ],
   "pendingAmount":0.00
}
```

Evaluation

Most of all, your solution should be correct and clean.
Your code has to be thread-safe and shouldn't have any mistakes.
It must demonstrate the practical usage of available data structures and standard library.
Don't do premature optimization. Try to maintain a good balance between maintainability and performance.
Test coverage - at your discretion. Do as you usually do, be yourself.