Matching Engine

Billy Sumners

August 21, 2020

1 Introduction

The Matching Engine is a tool for creating, cancelling, and matching orders for a hypothetical stock at a hypothetical stock exchange. It provides a REST API to achieve these objectives, as well as querying the system for historial orders and trades.

The system will be based around a SQL database. A Spring Boot application will query this database using JDBC, providing a REST API for a web frontend based on React to talk to. Entities will be validated with Spring/Hibernate. Time permitting, there will be an admin frontend based on Thymeleaf.

2 Database and Entities

There are four entities considered in this application, represented as tables in the database and classes in the Java application.

2.1 Database Entities

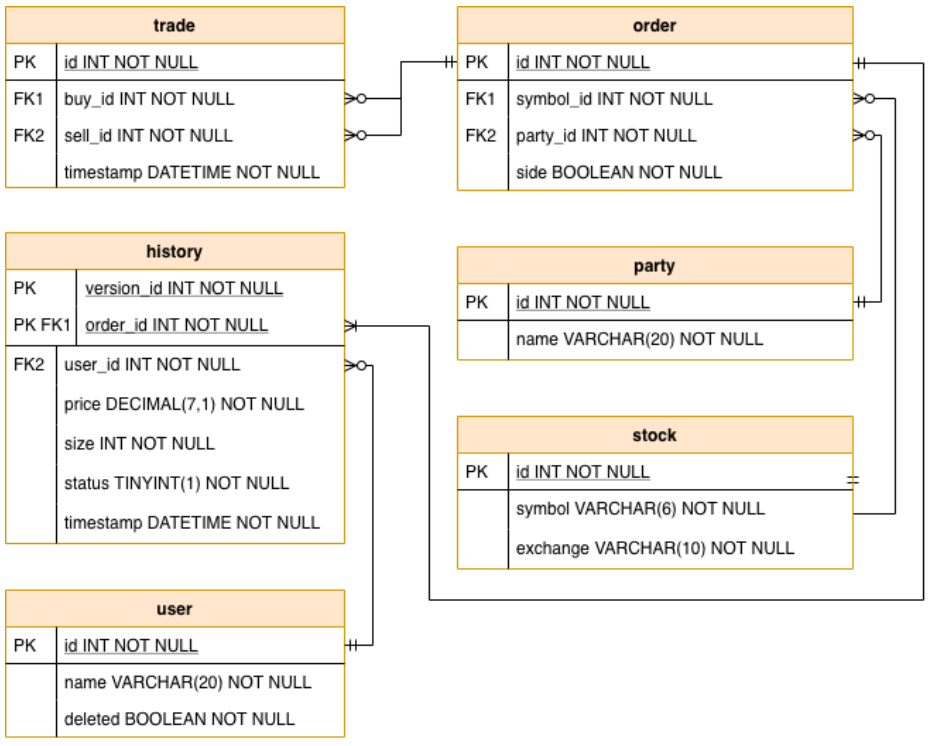

Figure 1: Database entity-ralationship diagram

Stock The stock entity corresponds to a stock being traded at the exchange. It has an `id` to act as the primary key, a `symbol`, and the `exchange` the stock is traded at.

Party The party entity corresponds to an actual party (e.g. a company) trading the stock. It has an `id` for the primary key, and a `name`.

User The user entity corresponds to a user working with the application. It has an `id` to act as the primary key, a `name` (the username), and a `deleted` boolean to indicate whether the user is suppressed from the system - deleting the user from the table entirely will break a foreign key constraint with order histories, which we must preserve.

Order The order entity corresponds to a buy or sell order made in the application. It has an `id` to act as the primary key, a foreign key `party_id` referencing the party owning the stock making the trade, a `stock_id` referencing the `stock` being bought/sold, the `side` of the order as a boolean (0 = sell, 1 = buy).

History The `history` entity corresponds to the history of an order's price and size from when it is first created to when the order is fulfilled or cancelled. It has a `version_id` and `order_id` to represent the version of a referenced order, acting as the primary key, a `price`, a `size`, a `status` of the order, and a `timestamp` set to when a `history` row is made. The idea is that when an order changes its price, size, or status in the system (through user modification or trades being made), a new version is made.

Trade The trade entity corresponds to a trade made by the application matching a buy order and a sell order. It has an `id` to act as the primary key, `buy_id` and `sell_id` referencing the buy and sell order respectively, and an `execution_time` when the trade is made.

2.2 Java Entities

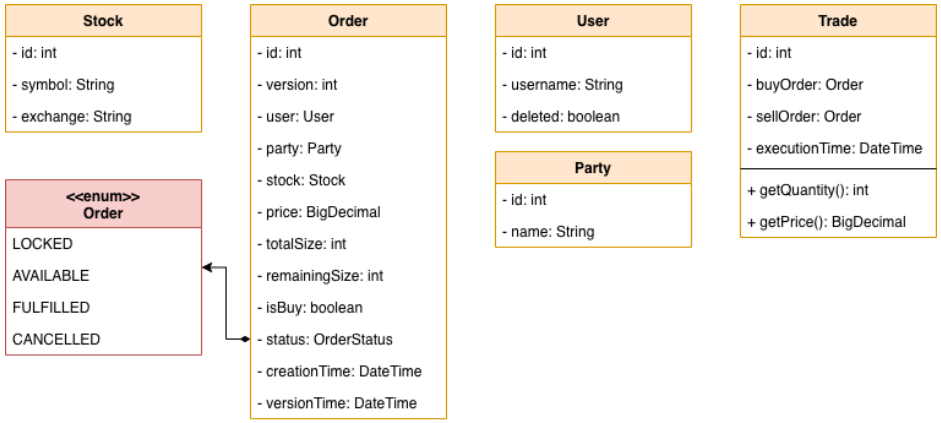

Figure 2: Java entities

Stock The `Stock` Java class corresponds exactly to the database. A `Stock` is valid if none of its fields are null, the symbol has length at most 6, and the exchange has length at most 10.

User The `User` Java class corresponds exactly to the database, with `name` being called `username` in the class A `User` is valid if none of its fields are null, and its name is at most 20 characters.

Order The `Order` class is effectively a combination of the `order` and `history` tables. Within the `Order` class, the `user_id` field will be replaced by the referenced user entity, and the `stock_id` will be replaced by the referenced stock entity. The boolean field `isBuy` will take the place of `side` in the table, being `true` if the order's side is buy, and `false` if the side is sell. The reason for this decision is that it will be easier to store in the database. The `status` field will be an enum matching the `status` column in the table.

The `Order` class will take values from the `history` table for the `price` and `remainingSize`. The fields `creationTime` and `lotSize` will be the relevant fields taken from the `history` table for when the order was first created.

An order is valid if none of its fields are null, the price has at most 7 digits with 1 digit after the decimal point, the `lotSize` and `remainingSize` is at most 100 000, the status is between 1 and 3, and the creation time is in the past.

Trade Within Java, the `buy_id` and `sell_id` will be replaced by the referenced order entities, with the orders set to be as they were at the time of the trade. There will be a `getQuantity()` property, which will be calculated based on the size of the buy and sell order at the time the trade is made, and a `getPrice()` property, which will be calculated based on the the price of the sell order at the time the trade is made.

A trade is valid if none of its fields are null, and the execution time is in the past.

3 Java Backend

Most things in the backend are CRUD. For each Java entity, there is a controller (`@RestController`), a service layer, and a DAO. The relevant classes for `Orders` are larger than the others, and most interesting are the `createOrder(..)` and `editOrder(..)` methods in the `OrderService`. These methods take in some preliminary values for a new/edited `Order`, construct it, check if it's valid (throwing an exception containing a list of validation errors if not), and adds it to the database. It then checks the `OrderDao` for any matching orders, creating a `Trade` entity and updating the relevant `Orders` if so.

4 REST Endpoints

4.1 Stock

GET /stock Returns a JSON array of all stocks in the system.

4.2 Trade

GET /trade Returns a JSON array of all trades in the system.

GET /trade/{id} Returns a JSON object of a trade in the system with the given ID if it can be found, and 404 not found otherwise.

4.3 User

GET /user Returns a JSON array of all users in the system.

4.4 Order

GET /order/buy Returns a JSON array of all buy orders in the system.

GET /order/sell Returns a JSON array of all sell orders in the system.

GET /order/status/{status} Returns a JSON array of all orders with the given status (`pending`, `fulfilled`, `cancelled`). Returns 404 Not Found if the given status does not exist.

`GET /order/user/{id}` Returns a JSON array of all orders made by the given user, returning 404 Not Found if a user with the given ID does not exist in the system.

`GET /order/{id}` Returns a JSON object of the order with the given ID, returning 404 Not Found if it cannot be found.

`POST /order?stock-id={stockId}&is-buy={isBuy}&price={price}&size={size}` Creates an order in the system with the given parameters, returning a JSON object of the order if it succeeded, and 422 Unprocessable Entity if it did not succeed due to a bad parameter.

`POST /order/cancel/{id}` Cancels the order with the given ID, returning 404 Not Found if the given order doesn't exist.

`POST /order/edit/{id}?stock-id={stockId}&is-buy={isBuy}&price={price}&size={size}` Edits an order in the system with the given ID with the given parameters, returning a JSON object of the order if it succeeded, 404 Not Found if the order does not exist, and 422 Unprocessable Entity if it did not succeed due to a bad parameter.

5 User Frontend

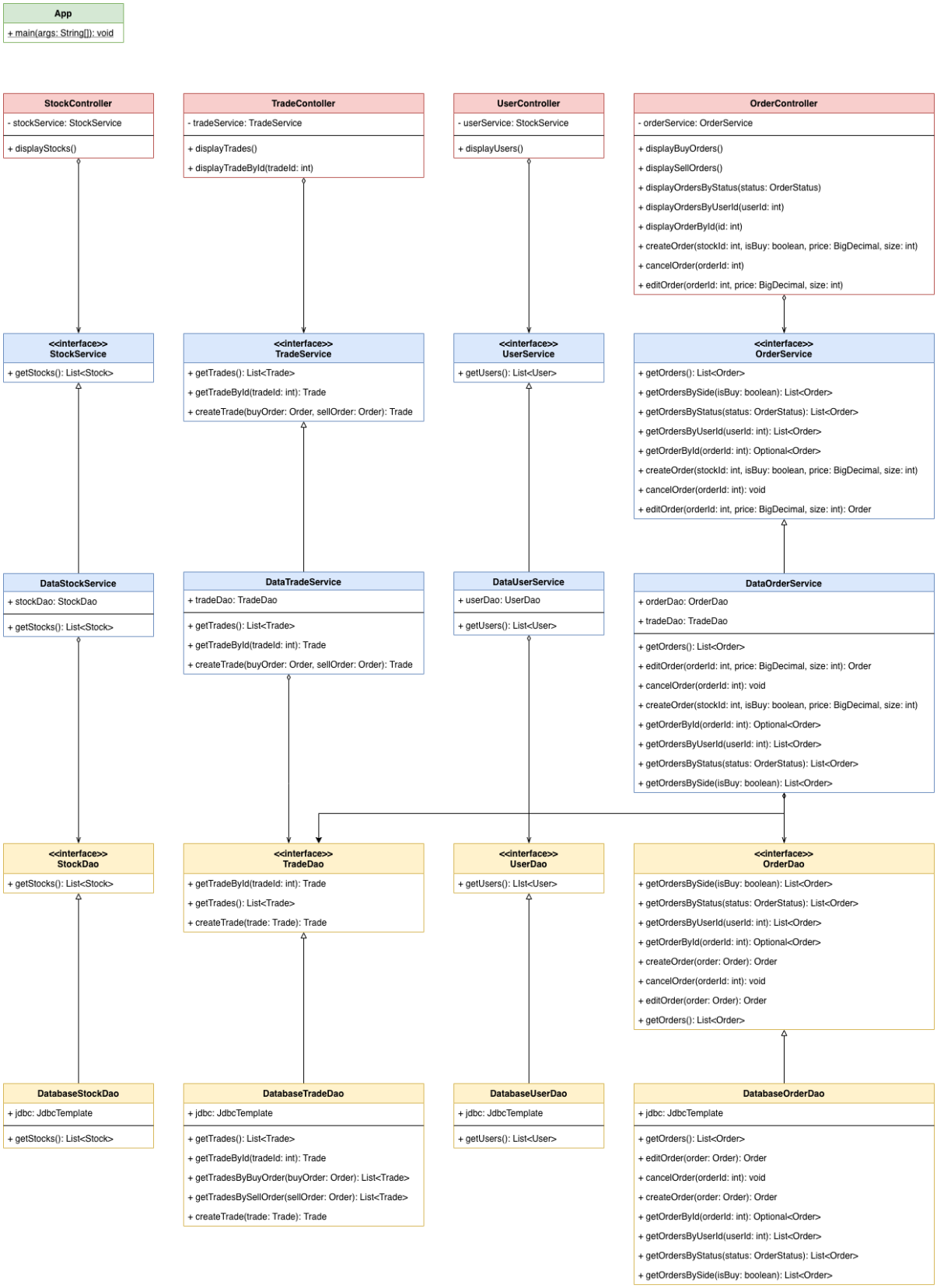

Figure 3: Class diagram