Matching Engine

Billy Sumners

August 24, 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

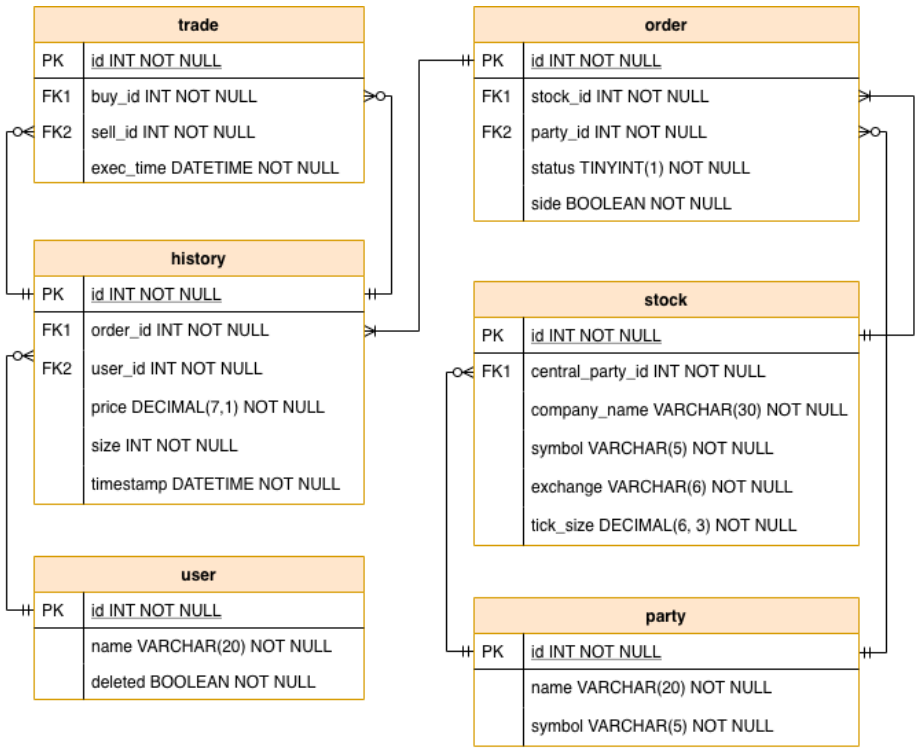

Figure 1: Database entity-ralationship diagram

Stock The stock entity corresponds to a stock being traded at the exchange. It has an `id` to act as the primary key, a `central_party_id` representing the party trading the stock (should be LCH), a `symbol`, the `exchange` the stock is traded at, and a `tick_size`.

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

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), and the order's `status`.

Each time an `order` is made, a new `history` must be made referencing the initial version of the order.

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 an `id` to act as the primary key, an `order_id` to reference the order, a `user_id` to reference the user who made the change to the order (if an order is matched by the system, this will be the user who made the matching order), a `price`, a `size`, and a `timestamp` set to when a `history` row is made. The idea is that when an order changes its price or size 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

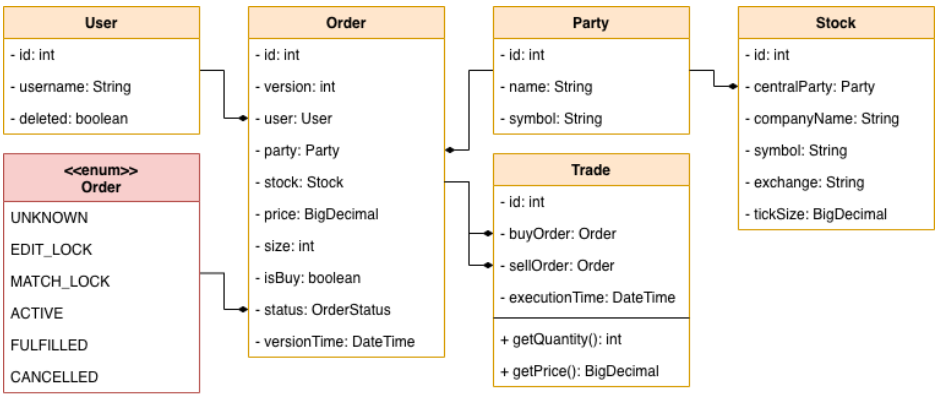

Figure 2: Java entities

Stock The `Stock` Java class corresponds exactly to the database. A `Stock` is valid if:

1. None of its fields are null.
2. The `companyName` has length at most 30.
3. The `symbol` has length at most 5.
4. The `exchange` has length at most 6.

5. The `tickSize` is positive, is at most 100, and has scale at most 3 (i.e. between 0.001 and 100.000).

User The `User` Java class corresponds exactly to the database, with `name` being called `username` in the class. A `User` is valid if

1. None of its fields are null.
2. The `username` has length at most 20.

Party The `Party` class corresponds exactly to the database. A `Party` is valid if

1. None of its fields are null.
2. The `name` has length at most 20.
3. The `symbol` has length at most 5.

Order The `Order` class is effectively a combination of the `order` and `history` tables. While the `id` will remain as it is in the database, the `version` will be calculated based on the position of the version in the `history` table. The `versionTime` will correspond to the `timestamp` in the `history` table. The `party_id` field will be replaced by the referenced `Party` entity, and the `stock_id` will be replaced by the referenced stock entity. The `price` and `size` will be as they are in the database. 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 design is that it will be easier to commit it to the database, while still remaining somewhat self-documented. An `Order`'s status will be an enum called `OrderStatus` which can take five values:

- **`UNKNOWN` when an order's status is unknown.**
- `EDIT_LOCK` when an order is being edited.
- `MATCH_LOCK` when the system finds a matching order and is updating this order in response.
- `ACTIVE` when an order is available for usage.
- `FULFILLED` when the order's size is 0.
- `CANCELLED` when the order has been manually cancelled.

A corresponding state diagram is found in figure 3.

An `Order` is valid if

1. None of its fields are null.
2. The `version` is at least 0.
3. The `price` is nonnegative, has scale at most 2 and at most 8 digits.
4. The `size` is at most 10 000 000.
5. The `versionTime` 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, **except for the `status`, which will be set to `UNKNOWN`**. There will be `getQuantity()` and `getPrice()` properties, which will be calculated based on the buy order and sell order.

A trade is valid if

- None of its fields are null
- The execution time is in the past.

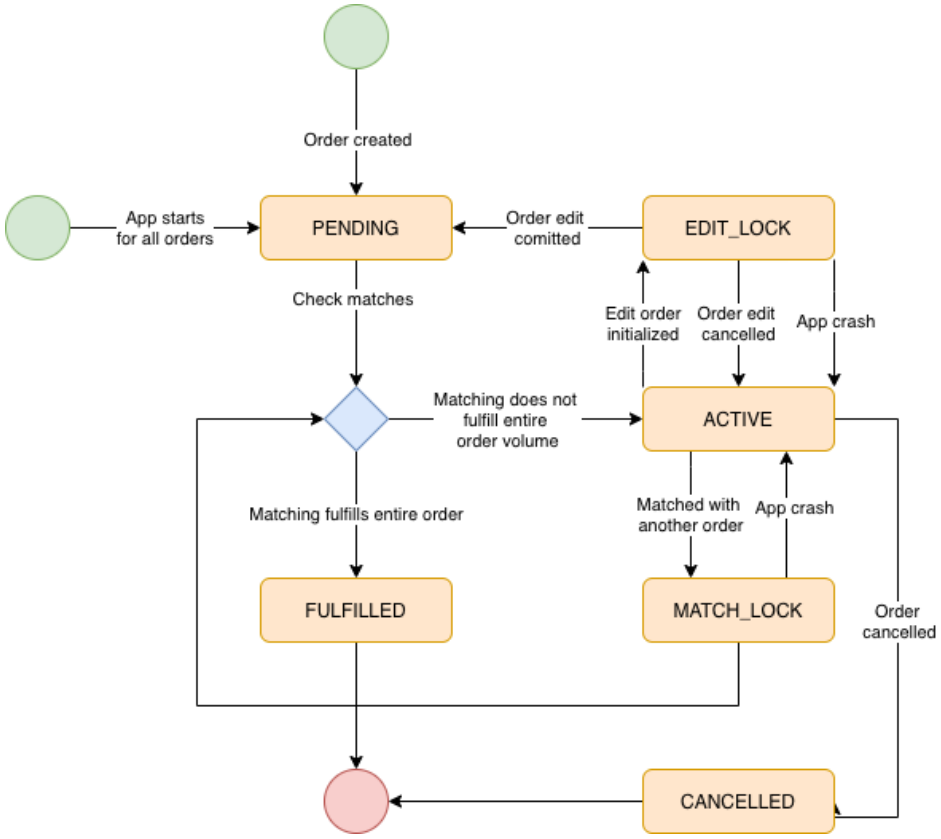

Figure 3: Order status state diagram

3 Java Backend

The backend is a skyscraper-like design, with classes from controllers down to repositories for each data entity.

Repositories There are six repositories, one for each data entity, and one for auditing purposes separated into interfaces and implementations. The data entity implementations are prefixed `Database` to indicate they read to/write from a relational database using `org.springframework.jdbc.core.JdbcTemplate`. The audit repository implementation is prefixed with `TextFile` to indicate it writes to a text file.

The `Stock`, `User`, and `Party` DAOs contain a small number of methods as a consequence of being small and mostly immutable entities. In particular, there are methods to get all entities of the relevant type in the system and get an entity by its ID, as well as methods to delete all such entities (primarily for testing purposes), and adding entities. The `UserDao` also contains a method to edit a user, since a user's `deleted` field is mutable.

The `AuditDao` contains only one method, `writeMessage(message: String)`, whose `message` parameter contains the message to write to the audit log, to which additional information may be appended by the method.

The `TradeDao` contains the same add, delete all, get all, and get by ID methods, as well as methods to get a trade by its buy order and its sell order.

The `OrderDao` contains the same add, delete all, get all, and get by ID methods, as well as including the ability to edit an order, getting all orders by the side they lie on (buy or sell), getting all orders by their status, and getting all orders made by a particular user.

Valdation checks are performed in the DAOs when adding or editing an entity. A specific `InvalidEntityException` is thrown with a list of validation errors when the checks fail. Editing an entity throws a `MissingEntityException` when an entity with the ID of the entity in the parameter doesn't already exist in the system.

Services There are five service classes, one for each entity. The service implementations are prefixed with `Data` to indicate they talk with a data layer to do their work. Each service implementation communicates with the `AuditDao`

to write down everything that happens in them, as well as the repository corresponding to the entity they manage. The `DataOrderService` also communicates with the `TradeDao`.

All services contain passthrough methods delegating to the repository for each method in their entity's repository, except for delete methods. In fact, `StockService`, `TradeService`, `UserService`, and `PartyService` contain only these methods

The `OrderService` also contains passthrough methods for its repository, but the `createOrder( .. )` and `editOrder( .. )` methods are slightly special. They take parameters which allow an `Order` to be constructed, and as it is committed to the repository, the repository checks for any orders of the opposite side matching the committed order, and creates trades/updates orders in response to this. There are also `beginEditOrder(orderId: int)` and `cancelEditOrder(orderId: int)`, which update

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 Party

GET /party Returns a JSON array of all parties in the system.

4.5 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

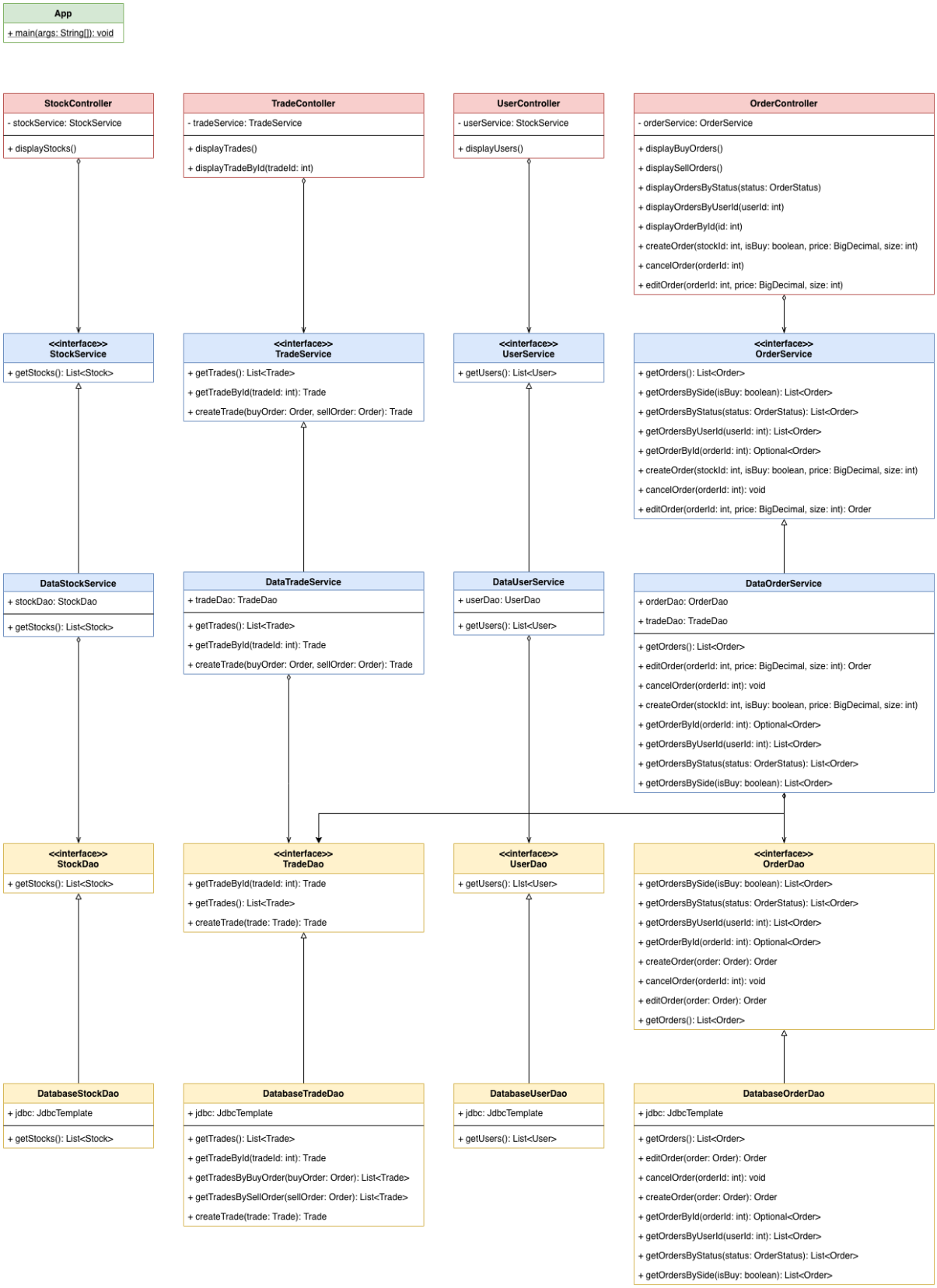

Figure 4: Class diagram