Smart Order Routing (SOR) - Group Project

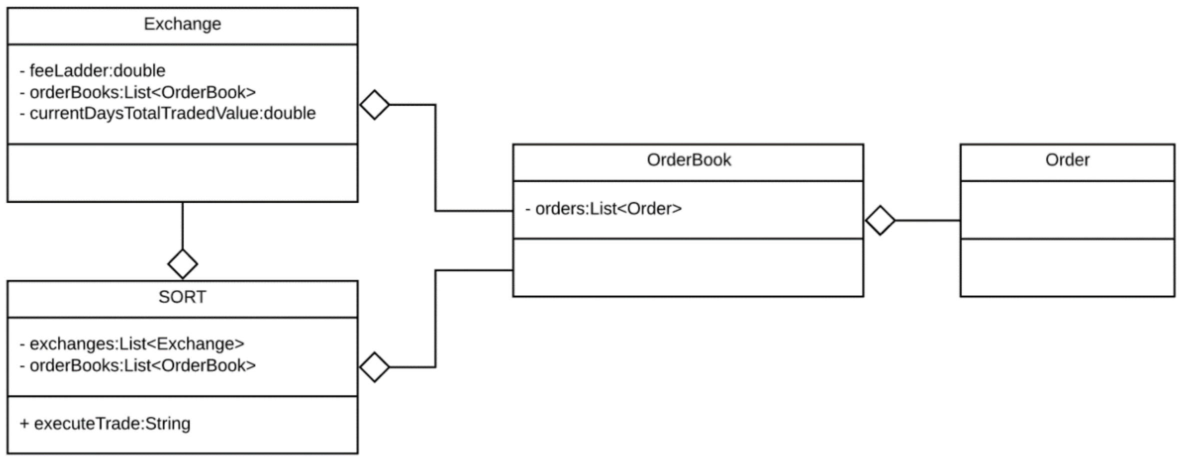
Objective

The objective of this group project is to build Full Stack Java Application using most of the learning outcomes from the course modules

Overview - Scenario and Design

You've been asked to develop a Full Stack Java Application for an electronic trading platform. The application will be available to online stock traders/clients who can place their orders (BUY & SELL) for trading, view their order status and the trade/order history.

Class Hierarchy & Sample Design Guidelines


Exchange
- feeLadder:double
- orderBooks:List<OrderBook>
- currentDaysTotalTradedValue:double
OrderBook
- orders:List<Order>
Order
SORT
- exchanges:List<Exchange>
- orderBooks:List<OrderBook>
+ executeTrade:String

SORT - trades on various exchanges based on best prices and fees. There would be at least one instance of SORT per region (eg: EMEA/APAC etc.). This class requires Exchange object to model exchange fees and how they change based on feeLadder (bulk=cheaper)

Exchange needs **orderbooks** so that we can store the exchanges orderbook and therefore decide whether an exchange has liquidity and a good price or not.

SORT requires **orderbooks** of the orders it's trading – this allows SORT to perform internal crossing. Generally, each instrument will have its own orderbook (with a collection of orders), BT has 1, VOD has 1 and so on

HINT: To start with you may consider generating sample orders 500 each (BUY and SELL) with some random prices and populate database tables

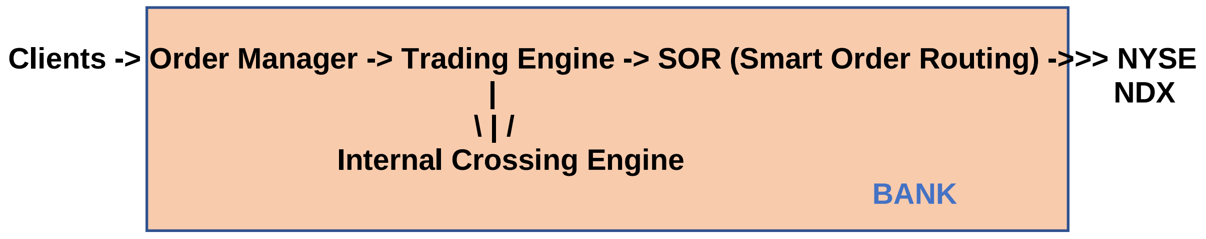
Functional Requirements

1. Ability to register and login
2. Ability to view BUY and SELL Orders
3. Ability to manage orders (Add new orders, cancel, or replace orders)
4. Add support for buy and selling of instruments
5. The status of all the orders (partially fill or fully filled)
6. Ability to Slice orders and send to SORT for now – avoid other ways to fill the orders other than sending them to SORT
7. Match Single orders (BUY -> SELL) to execute trades
8. Match multiple orders (BUY -> SELL) to execute trades
9. Ability to view trade history
10. Allow multiple traders to use the application (HINT: You may have to register some sample traders)
11. Use logging for output statements (avoid using System.out.println for any output statement)
12. Demonstrate Unit testing
13. Generate a nice Javadoc

Technical Requirements

- Use Agile with Scrum
- Use "git" for version control (Consider creating a "git" repo for your team to start with)
- Translate functional requirements into user stories
- Demonstrate Java SE and Spring Framework concepts/modules as part of backend development
- Demonstrate functional style code
- Consider using Java 8 Streams to deal with large data set (HINT: orderbooks with bulk orders)
- Use Spring Boot for Spring Web, Spring Data JPA and Spring Security modules
- Demonstrate Web Service and Microservices
- Demonstrate Angular or any relevant frameworks/libraries as part of front-end development
- Demonstrate JUnit 5 for unit testing
- Demonstrate Maven or Gradle for build automation
- Use SLF4J or Log4J for logging
- Use Lombok dependency to avoid boilerplate getters, setters and constructors
- Use JIRA and Confluence or any related Project workflow management and documentation tools
- Deploy your application on embedded Tomcat server provided by Spring Boot or any server on Cloud environment

Example - Electronic Trading System's Workflow


Clients -> Order Manager -> Trading Engine -> SOR (Smart Order Routing) ->>> NYSE
NDX
|
\ | /
Internal Crossing Engine
BANK