RO4

Stock Exchange Simulator

Presented by

Akshat Agarwal

Sukruti Rai

Supervisor :
Prof. Wentao Xie

RO4

table of contents

RO4

Rise of Electronic Trading

- The evolution of stock markets has transitioned from physical trading floors to automated, electronic trading systems.
- Benefits: Faster transactions, higher accuracy, scalability, and better compliance with regulatory requirements.
- Modern Trading: High-frequency trading, algorithmic trading, and the adoption of electronic trading platforms.

RO4

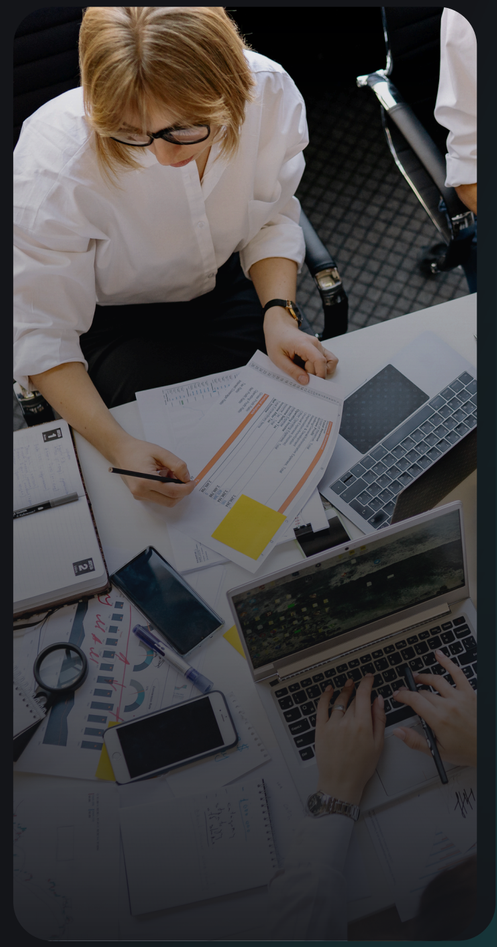
Roles of EMS and OMS in Electronic Trading

Execution Management Systems (EMS)

- Manage trade execution across brokers and exchanges.
- Ensure best execution by optimizing for liquidity, price, and speed.

Order Management Systems (OMS)

- Handle the entire order lifecycle, from creation to execution.
- Enable real-time monitoring, routing, and tracking of orders.



RO4

Terminology

- Bid Price: The price at which users are willing to **buy** a stock.
- Ask Price: The price at which users are willing to **sell** a stock.
- Limit Order: An order to buy or sell a stock at a **specified** price or better.
- Market Order: An order to buy or sell a stock immediately at the **current market price**.
- Orderbook: A real-time digital record of all buy and sell orders in a market, typically organized by price level showing shows bid prices (buy orders) on one side and ask prices (sell orders) on the other, and it helps traders visualize market depth and liquidity.
- Financial Information eXchange (FIX) protocol: an industry standard communications protocol for international real-time exchange of securities transactions and markets related information.

RO4

What is FIX Protocol?

- Standardized messaging format for trade-related information (e.g., order submission, amendments, cancellations).
- Widely adopted for seamless communication between exchanges, brokers, and institutional traders.

Key Benefits:

- Streamlined trading processes, reduced errors, and improved execution speed.
- Global standard for real-time and automated trading.

RO4

Motivation for the Project

High Cost and Limited Access to Live Testing:

- Testing on real exchanges is very expensive and restricted to narrow time windows.
- Firms must book special testing sessions or wait for limited exchange test slots, slowing development and increasing costs.

Need for Reliable EMS and OMS Testing:

- Modern trading relies heavily on EMS and OMS to place, route, and manage trades.
- These systems must be fast, accurate, and regulation-compliant across different markets.

What our project solves?

Eliminates High Testing Costs

Overcomes Time Limitations

Validates EMS and OMS Systems

Handles Multi-Exchange Complexity

RO4

Objectives

Develop a FIX-Based Matching Engine

- Build a custom engine that supports market and limit orders.
- Enforce price-time priority and tick-size rules accurately.

Importance: Forms the core of the simulator, enabling realistic trade execution and compliance with exchange protocols.

Support Real-Time Multi-Client Order Management via GUI and FIX

- Handle distinct FIX sessions for concurrent users and allows multiple clients to place, amend, and cancel orders in real-time.
- Provide a user-friendly GUI to test order flows without manual FIX coding.

Importance: Simulates a live, multi-user environment, making it ideal for team testing and real-world EMS/OMS integration.

RO4

Objectives

Maintain Real-Time Order Book Persistence

- Keep a live, dynamic order book showing bid/ask depth and active orders.
- Reflect instant updates when orders are placed, matched, or canceled.

Importance: Enable detailed tracking of order flow and market behavior and replicates true exchange dynamics.

Build a Web-Based User Interface

- Develop a secure frontend to display the order book in real time and visualize trading activity.
- Add authentication to ensure controlled access for testers

Importance: Makes the simulator accessible, promotes visual feedback, and supports remote QA testing.

RO4

System Design

Service-Oriented Architecture (SAO)

- System is built as a set of independent, loosely coupled services.
- Each service handles a single responsibility: receiving orders, validating, matching, persisting, etc.
- New features can be easily integrated by extending or adding services, making the system modular.

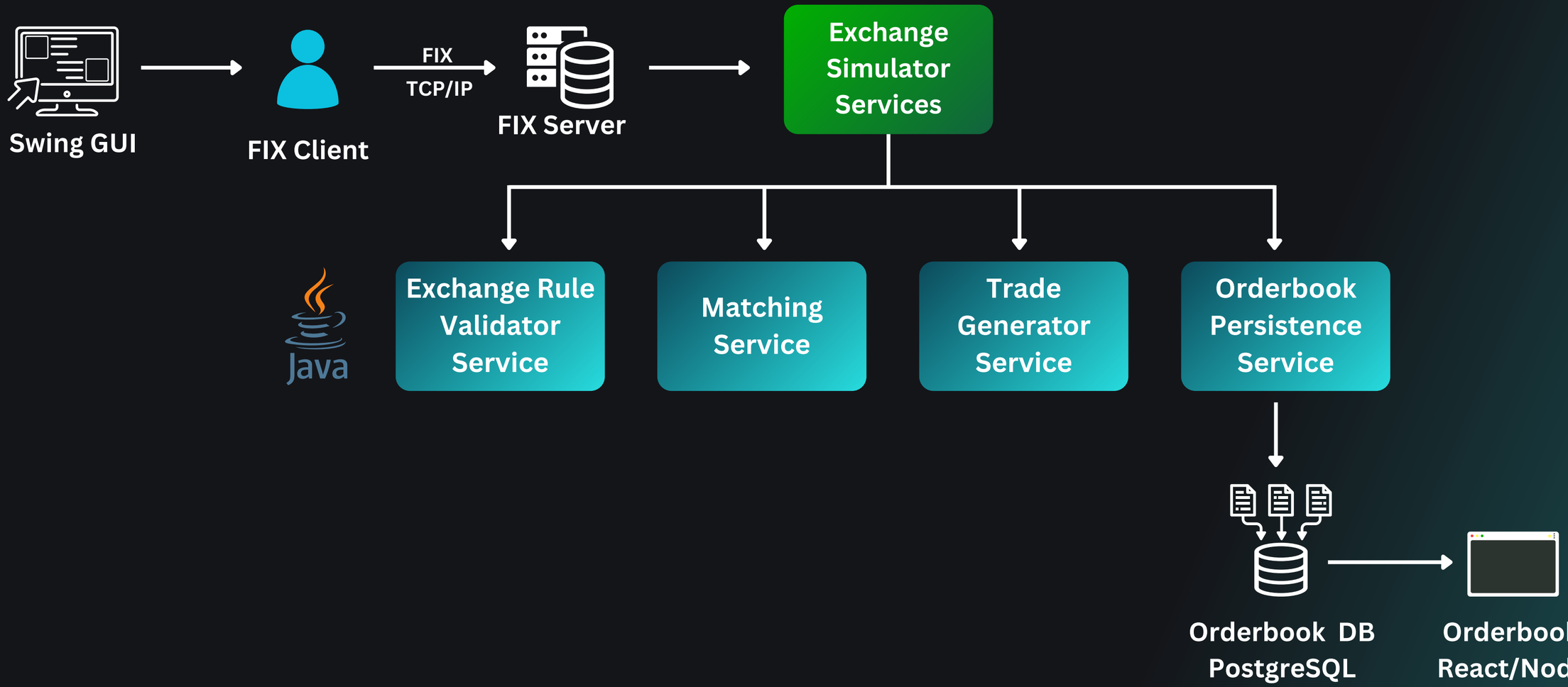
Domain Driven Design (DDD)

- The system is modeled around core trading domain concepts (e.g., Order, Trade, OrderBook, MatchingRule).
- Business logic is encapsulated inside domain-specific services, reducing code duplication and improving maintainability.
- Each domain entity (e.g., an Order object) contains only logic relevant to itself, promoting clarity.

Together, SOA and DDD make the system modular, maintainable, and realistic — enabling accurate simulation of exchange behavior and easy future expansion.

RO4

Workflow



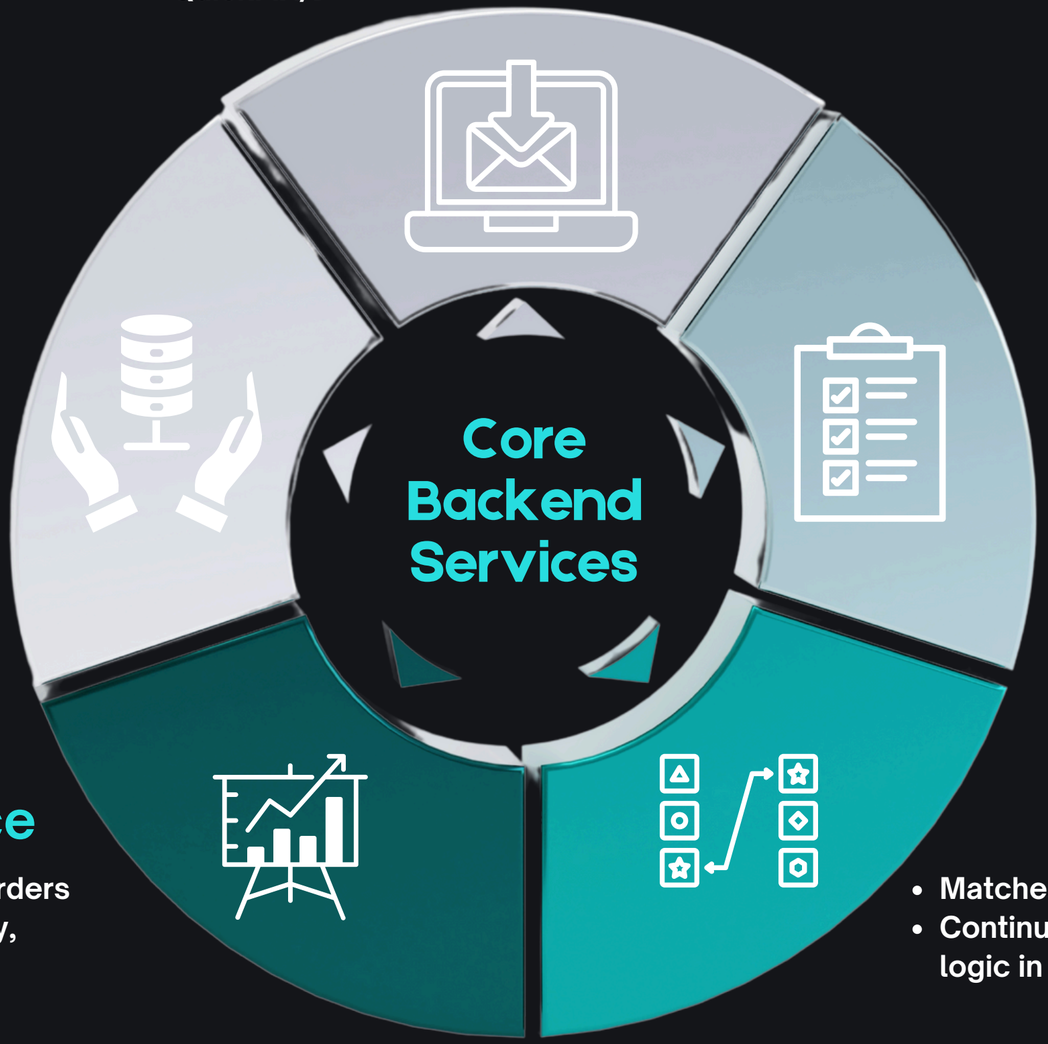
RO4

FIXOrderReceiverService

- Handles incoming FIX messages (New, Amend, Cancel)
- Establishes and maintains FIX sessions using QuickFIX/J

OrderbookPersistence Service

- Updates the live order book in PostgreSQL
- Exports current state to orderbook.json for frontend visualization



ExchangeRuleValidatorService

- Validates orders based on tick size, order type, and market rules
- Ensures only valid orders move forward to matching

TradeGeneratorService

- Generates trade objects from matched orders
- Includes execution details (price, quantity, timestamp)

MatchingService

- Matches orders using price-time priority logic
- Continuously scans queues and executes matching logic in real time

RO4

Domain Model- Core Classes & Structure

Key Domain Classes:

Order: Stores side, type, quantity, price, client ID, timestamp

Market: Holds order books, ticker symbol, and rules

PriceData: Price, quantity, and side

MarketData: Stores live bid/ask info, price, stats.

DAO Layer (Database Access):

To maintain a clear separation between application logic and data persistence, the simulator employs a DAO (Data Access Object) layer:

- **OrderBookDAO:** Performs all database operations for orders—adding, updating, removing, and querying—to rebuild and track the order book history.
- **MarketDataDAO:** Stores and retrieves aggregated market metrics (e.g., volumes, price movements) for analytics and frontend display.
- **DBConfigService:** A singleton utility responsible for loading and maintaining PostgreSQL database configuration settings and ensures consistent access to connection details.

RO4

Design Patterns Used

Singleton:
Ensure system-wide DB config consistency (DBConfigService)

Factory:
Dynamically creates orders from FIX fields (OrderFactory)

Observer:
Pushes trade updates to UI via JSON (PriceUpdateService)

Strategy:
Supports switchable matching logic (MatchingService)

Concurrent Structures:
Uses thread-safe maps and queues to handle multi-client access

RO4

Best Development Practices

Encapsulation

Each core function (validation, matching, persistence) is isolated in its own service for better maintainability.

Concurrency Handling

Thread-safe data structures (e.g., ConcurrentHashMap, synchronized queues) ensure stable state.

Polymorphism

Matching strategies and FIX message handlers use shared interfaces, enabling flexible behavior extensions.

Scalability

Modular, service-oriented architecture allows individual components to scale or evolve independently.

RO4

Matching Engine Implementation

- Core logic handled by **MatchingService**
- **Symbol-Based Isolation:** Each stock symbol maps to its own Market object
- **Order Queues:**
  Buy queue: sorted by highest price, earliest time
  Sell queue: sorted by lowest price, earliest time
- **Matching Logic:**
  Continuously compares top bid vs ask to execute matching trades
  Prioritizes best price and earliest arrival (price-time priority)
- **Thread-Based Execution:**
  Engine runs in a continuous background thread for real-time matching
  Ensures market updates are immediate and consistent

RO4

MatchingService

Code-Level View

- **Singleton Design:**
  Guarantees only one MatchingService instance across the app

- Market Management:
  HashMap<String, Market> stores per-symbol order books

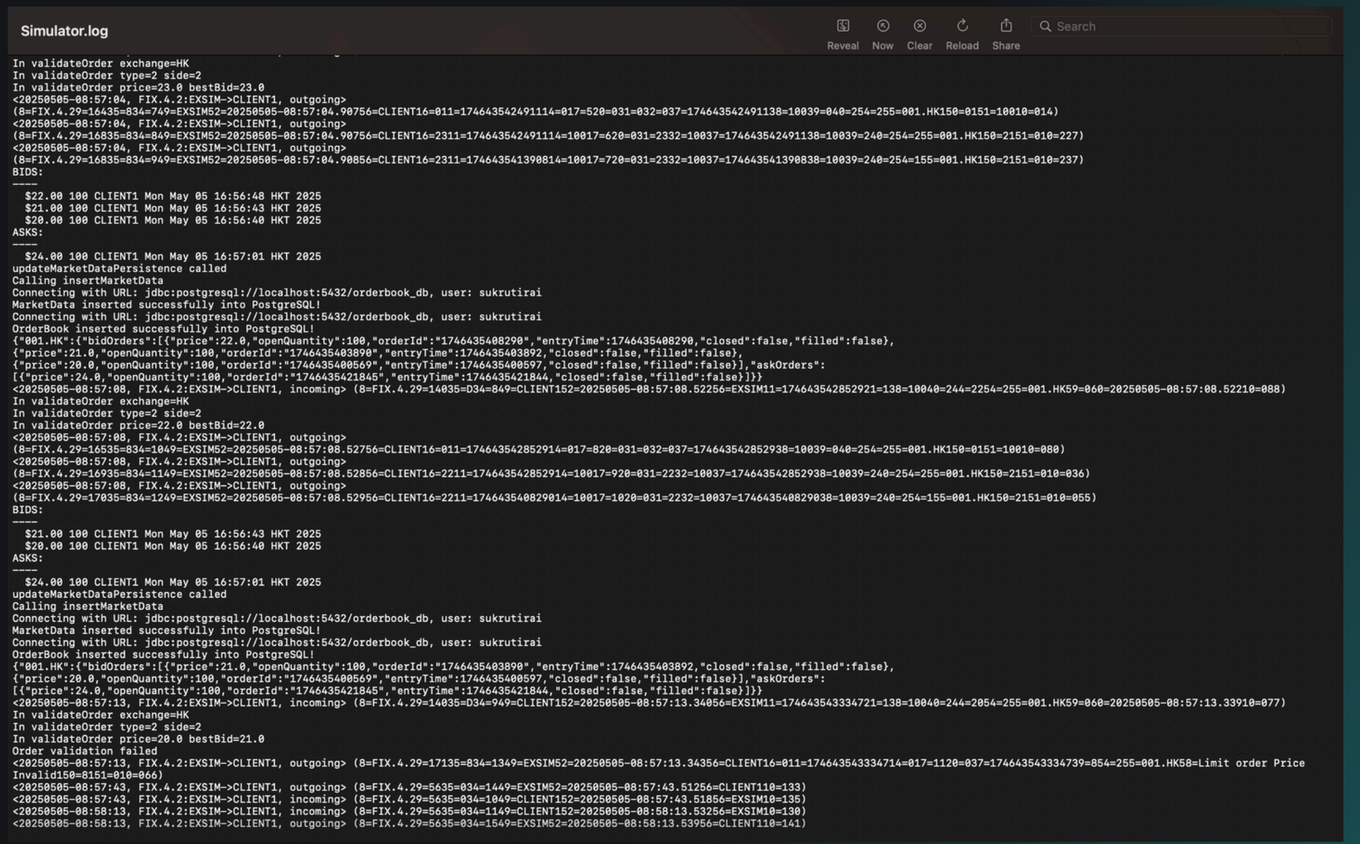
- **Key Methods:**
  Insert(Order) – Adds order to the appropriate Market
  match(symbol, orders) – Triggers price-time matching logic
  find(symbol, side, id) – Locates a specific order (used in amend/cancel)
  erase(Order) – Removes an order from the book
  displayOrderBook() – Calls PersistenceService to generate live JSON snapshot for frontend

- **Integration**: Works with FIX receiver and Persistence layers to handle the full trading cycle

RO4

FIX Protocol Integration

- FIX connectivity via QuickFIX/J
- Supports message types:
  D = New Order
  G = Amend Order
  F = Cancel Order
- ExSimApplication implements quickfix.Application
- Handles concurrent FIX sessions with sequencing & heartbeats
- Sends execution reports, fills, rejections over FIX
- Logs every execution + displays updated order book



RO4

Order Book & Persistence

In memory: Concurrent Hashmap stores real-time order states

Postgres SQL: Stores trades, orders, and market stats durably

DAO layer handles SQL abstraction (OrderBookDAO, MarketDataDAO)

Order book also serialized to orderbook.json for frontend sync and continuously updated

RO4

Rest API & Authentication

REST API Layer:

- Serves live order book snapshots, trade history, and market statistics
- Powers the React frontend using secure, real-time data endpoints

DAO Layer Integration:

- OrderBookDAO and MarketDataDAO handle all PostgreSQL read/write operations
- Abstracts SQL logic, enabling clean and efficient backend access for the API

JSON Snapshot Sync:

- Orderbook.json is auto-generated after every market update
- Used by the frontend for lightweight, real-time visualizations of bid/ask queues

JWT Authentication:

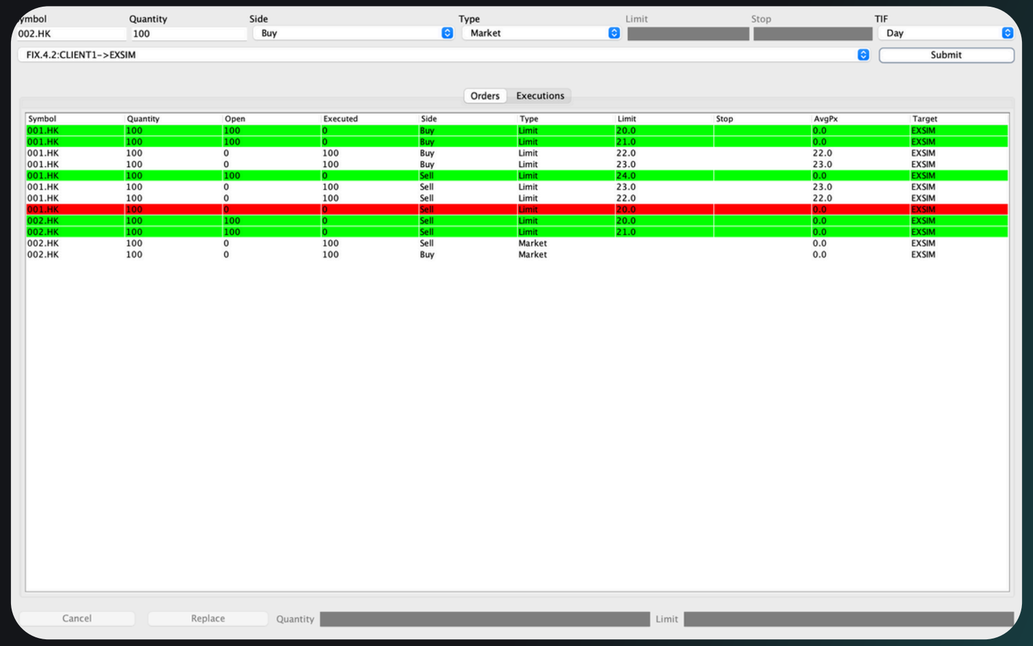
- Ensures secure login and access to frontend features
- All REST requests are verified using JWT tokens to maintain session integrity

RO4

User Interfaces

Graphical Test Tool - Swing GUI (Order Sender)

- Desktop client for FIX testing, built with Java Swing
- Users can:
- • Place new market/limit orders
- • Amend or cancel existing orders
- • Monitor real-time execution reports
- Simulates multiple clients via configurable FIX sessions
- Ideal for EMS/OMS system simulation



Order Sender

RO4

User Interfaces

Web-Based Frontend for Real-Time Market View

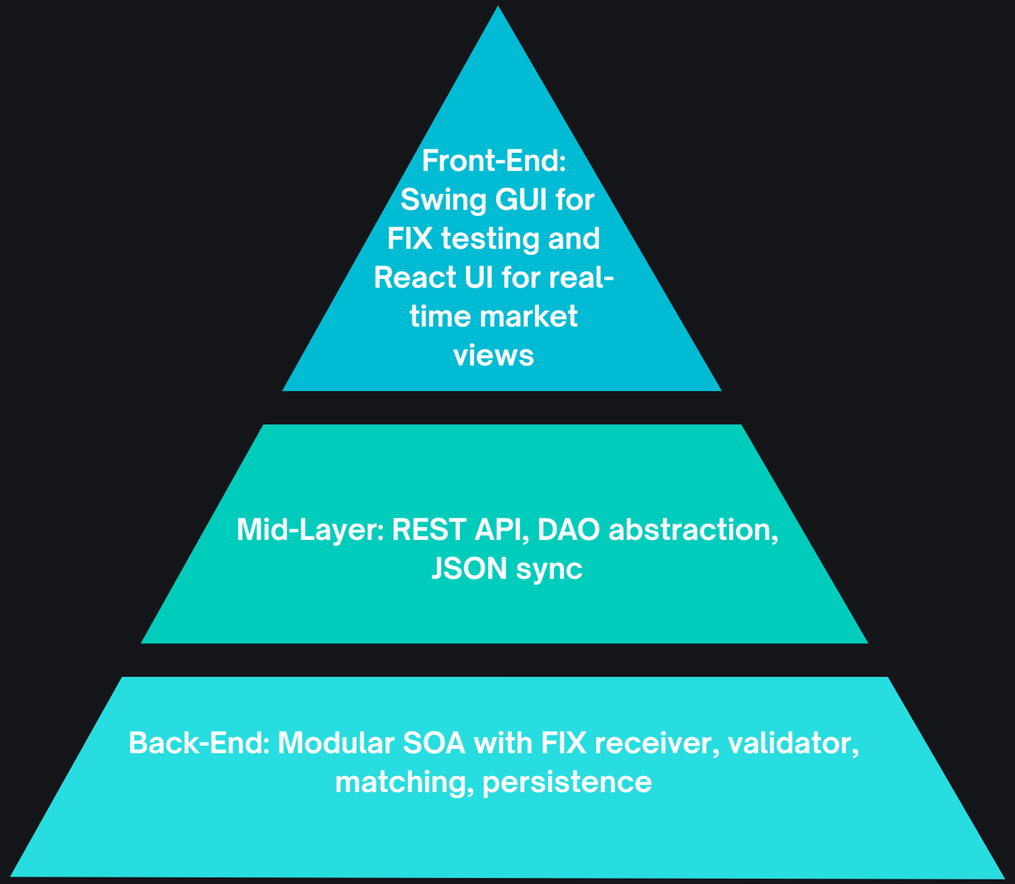
- Built with React + Node.js
- Authenticated using JWT
- Displays real-time data via REST API
- Tabs:

1. Login Dashboard – User auth, sign-up, forgot password
2. Home Tab – Top stock summaries (exchange, volume, avg. price)
3. Order Book Tab – Bid/ask queues with timestamp & quantity
4. Market Data Tab – Best bid/ask, high/low, last trade (auto-updated)

Home Tab View

RO4

Integration Recap



RO4

Testing

Unit Testing

- Tested backend services like: MatchingService, FIXOrderReceiverService, PersistenceService, ExchangeRuleValidatorService, PriceUpdateService
- Verified:
  Order matching by price-time priority
  Accurate trade execution logging
  Market data updates (best bid/ask, last price)
  PostgreSQL DAO operations (CRUD)

- Used custom test harnesses and logging for traceability

Integration Testing

- Simulated real workflows via Swing GUI + QuickFIX/J
- Tested Interactions Between:
  FIX ➜ Validation ➜ Matching ➜ Persistence ➜ Frontend
- Ensured:
  Correct FIX message lifecycle (D/F/G types)
  Real-time frontend sync via REST API
  Accurate orderbook.json generation for UI

RO4

Demo

RO4

Discussion & Technical Reflections

Persistence Layer Migration

- Started with H2 in-memory DB for prototyping speed.
- Faced issues with frontend integration and data consistency under concurrent loads.
- Migrated to PostgreSQL, enabled by clean DAO separation, minimizing system refactoring.

Market Data Replay - Deferred

- Initial plan to preload the order book via historical data replay.
- Feature was shelved due to complexity of dynamic reconstruction and timeline constraints.

Scalability & Deployment

- SOA enables horizontal scaling (multi-exchange support).
- Future-ready for:
  - Kubernetes-based failover
  - Persistent state restoration
  - Instrument-level sharding for performance

RO4

Technical Highlights

- Multi-threaded matching engine with price-time logic
- Domain-driven entities: Order, Trade, OrderBook
- FIX 4.2 integration with QuickFIX/J and message routing
- PostgreSQL + JSON dual persistence with DAO abstraction
- 

Exchange rule enforcement using market-specific tick-size constraints
- REST API with JWT-secured frontend

RO4

Conclusion

- Delivered a cost-effective, extensible simulator with modular backend services and a responsive frontend—ideal for research, development, and classroom demonstrations.
- Enables safe testing of algorithmic strategies and EMS/OMS systems through GUI- and FIX-based interfaces, eliminating risk from real capital markets.
- Prepared for real-world deployment with multi-exchange support, RESTful integration, and domain-driven service design that mirrors industry standards.

RO4

Q&A

RO4

Thank You!