Trading Platform for Stock and Cryptocurrency Exchange

Project Report for MINOR PROJECT II

submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science & Engineering

by

Name	Enrollment Number
Supragya Gandotra	R2142220674
Mohammad Zaid	R2142220744
Priyanshu Butola	R2142220346

Under the supervision of

Dr. Deepak Kumar Sharma
Assistant Professor
Department of Computer Science

January - May 2025
School of Computer Science
University Of Petroleum and Energy Studies
Dehradun-248007

CANDIDATE'S DECLARATION

We hereby certify that the project work entitled **"Trading Platform for Stock and Cryptocurrency Exchange"** in partial fulfillment of the requirements for the award of the Degree of **BACHELOR OF TECHNOLOGY** in **COMPUTER SCIENCE AND ENGINEERING** with specialization in **Artificial Intelligence and Machine learning**, submitted to the AIML Cluster, School of Computer Science, UPES, Dehradun, is an authentic record of our work carried out during a period from **January, 2025** to **May, 2025** under the supervision of **Dr. Deepak Kumar Sharma, Assistant Professor, Department of Computer Science**.

The matter presented in this project has not been submitted by us for the award of any other degree of this or any other University.

Name	**Enrollment Number**
Supragya Gandotra	R2142220674
Mohammad Zaid	R2142220744
Priyanshu Butola	R2142220346

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Dr. Deepak Kumar Sharma
Project Guide

Date: ________________ 2025

Acknowledgement

We wish to express our deep gratitude to our guide **Dr. Deepak Kumar Sharma**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.We sincerely thank our respected **Dr. Anil Kumar**, Head, Department of **Artificial Intelligence and Machine Learning**, for his great support in doing our project on **Trading Platform for Stock and Cryptocurrency Exchange**.We are also grateful to the Dean, SoCS UPES, for giving us the necessary facilities to carry out our project work successfully. We also thank our Course Coordinator, **Dr. Amit Panwar**, and our Activity Coordinator, **Dr. Amit Panwar**, for providing timely support and information during the completion of this project. We would like to thank all our friends for their help and constructive criticism during our project work. Finally, we express our sincere gratitude to our parents who have shown us this world and for every support they have given us.

Name	**Enrollment Number**
Supragya Gandotra	R2142220674
Mohammad Zaid	R2142220744
Priyanshu Butola	R2142220346

Contents

Trading Platform for Stock and Cryptocurrency Exchange

1 Abstract

The trade finance industry has seen a major digital transformation, mainly with the rise of internet-based stock and cryptocurrency exchanges. Hence in response to this transformation to the financial industry, the objective of this project is to design and develop a comprehensive live trading platform that allows users to buy and sell both stocks and cryptocurrencies. The platform utilizes Next.js for the frontend interface, offering high performance and server-side rendering for fast, responsive user interactions. The backend is powered by Node.js and Express, providing a scalable and event-driven infrastructure capable of handling high-frequency trading operations.

At the heart of this platform is an advanced order book system, a core feature responsible for maintaining buy and sell orders across various trading pairs. Unlike traditional systems that periodically load and save order data from databases, this system keeps the entire order book in-memory as a single component. This design significantly enhances execution speed and reduces latency, offering users a seamless real-time trading experience. Orders are matched based on price-time priority, ensuring fairness and market efficiency. The platform supports various types of orders, including market orders, limit orders, and stop-loss orders, allowing for flexible trading strategies and robust liquidity management.

To ensure the system's resilience, especially in the event of a server crash or data loss, a Redis queue-based backup system is implemented. This system replicates and stores order books individually for each trading asset, enabling quick recovery and ensuring data integrity. Real-time data flow is achieved using WebSocket technology, which allows instantaneous communication between the server and clients. This enables live streaming of market data, order book updates, and trade confirmations, all essential for an interactive trading environment.

The overall architecture follows API routing standards inspired by existing exchange platforms, ensuring compatibility with third-party tools and facilitating future integration with external APIs or trading bots. The backend is designed to handle a large volume of concurrent connections while maintaining transaction security and speed. This project not only aims to offer a cutting-edge trading experience but also serves as a robust, scalable foundation for future enhancements in the fintech and crypto exchange space.

2 Introduction

As digital financial assets become more widely used, there is a growing demand for a high-performance trading platform that lets users execute real-time trades, analyze market trends, and manage their portfolios effectively. The goal of this project is to create a real-time trading system that ensures smooth and effective trading by having an order book architecture that is optimized, has low-latency execution, and strong security measures. The system is built to offer scalable architecture that supports high-frequency trading while ensuring security and dependability, as well as real-time price updates and effective order execution. The order book will be built to efficiently manage high trading activity volumes, guaranteeing real-time updates and low-latency access to bid-ask spreads.

2.1 Problem Statement

The growing demand for real-time trading platforms presents challenges in ensuring low-latency execution, efficient order book management, and robust security. Traditional systems often struggle with handling high-frequency transactions, leading to issues such as delayed trade execution, order mismatches, and data inconsistencies. Ensuring seamless market data updates, trade accuracy, and liquidity provision requires an optimized and scalable solution.

The platform must also address data integrity, system reliability, and market stability to ensure seamless trading operations. Implementing scalable microservices, in-memory order book processing, and Redis-backed replication is crucial for handling high trade volumes without performance degradation. Additionally, integrating market makers and liquidity providers will enhance order execution efficiency and price stability, ensuring a seamless trading experience for users.

2.2 Objective

To build a high-performance, secure, and scalable real-time trading platform

Sub Objectives

a. To develop an efficient order-matching engine and optimize API/WebSocket communication to ensure minimal latency in trade execution.

b. To implement robust security measures (e.g., 2FA, OAuth, API rate limiting) and scalable infrastructure (e.g., microservices, containerization) to ensure secure and dynamic handling of high trade volumes.

2.3 Purpose of the Project

The purpose of this project is to develop a real-time, secure, and scalable trading platform for stock and cryptocurrency exchanges. The platform is designed to handle high-frequency trading (HFT) with low-latency execution, in-memory order book processing, and real-time market updates. By leveraging microservices architecture, cloud deployment (Azure), and WebSocket communication, the system will ensure seamless trade execution, efficient liquidity management, and enhanced security for traders and institutions. The project also focuses on integrating market makers and liquidity providers to ensure price stability and trade efficiency while maintaining a secure and compliant environment through OAuth authentication, Two-Factor Authentication (2FA), and API rate limiting.

2.4 Project Scope

The scope of this project encompasses the design, development, deployment, and maintenance of a scalable trading platform with the following key components:

2.4.1 User Access Security

a. Secure user authentication authorization (OAuth, JWT, 2FA).

b. Role-based access for traders, market makers, and admins.

2.4.2 Trading Features

a. Order placement, cancellation, and trade execution.

b. Support for multiple order types (Market, Limit, Stop-Loss, Iceberg Orders).

c. Real-time market data streaming via WebSockets.

2.4.3 Order Book Liquidity Management

a. In-memory order book processing for ultra-fast execution.

b. Redis-backed replication to prevent data loss.

c. Market maker integration to ensure price stability.

2.4.4 Trading Features

a. Microservices architecture for scalability and fault tolerance.

b. Cloud-based deployment on Azure with auto-scaling (VMs, Load Balancer).

c. Docker & Kubernetes for seamless containerized deployment.

2.4.5 Analytics & Monitoring

a. Trade history, transaction logs, and market insights.

b. System performance monitoring and load balancing.

2.5 Motivation

The motivation behind developing this trading platform arises from the increasing demand for high-speed, secure, and scalable trading solutions. Traditional trading platforms often face latency issues, lack of real-time updates, inefficient order book management, and security vulnerabilities.

This project is driven by the need to:

a. Optimize trade execution speed using in-memory order book processing.

b. Enhance security through API rate limiting, OAuth authentication, and 2FA.

c. Ensure high availability & scalability with cloud deployment and microservices.

d. Provide real-time market data using WebSockets for a seamless trading experience.

2.6 Literature Review

Several existing trading platforms, such as Binance, Coinbase, and Robinhood, offer trading services for various asset classes. Low-latency architectures and real-time data streaming techniques are necessary for high-frequency trading, according to studies on financial trading systems[1, 2, 3].

2.7 Key Research Areas in Trading Platforms

2.7.1 Real-Time Data Processing:

a. WebSocket-based streaming is a widely adopted approach for providing real-time data updates[4].

b. For traders, it is essential to handle market data feeds—such as order book updates, trade execution reports, and price movements—efficiently.

c. Event-driven architectures enhance responsiveness in dynamic trading environments[5].

2.7.2 Order Book Management:

a. The order book is a critical component in trading platforms, maintaining a real-time list of buy and sell orders.

b. In-memory processing for extremely fast execution and database persistence for historical analysis are two methods for implementing order books.

c. Important elements that affect trading efficiency include spread computation, market depth, and liquidity availability[6].

d. Centralized exchanges (CEXs) make use of high-speed databases and caching methods, whilst decentralized exchanges (DEXs) frequently use smart contracts for on-chain order book implementation[7].

2.7.3 Microservices-Based Architecture:

a. Modern trading platforms use microservices architecture to handle different functionalities like order matching, account management, and market data services[8, 9].

b. This modular approach improves scalability, fault tolerance, and system flexibility.

c. Kubernetes and Docker are widely used for containerized deployments, ensuring seamless scaling and fault isolation[10].

2.7.4 Security and Compliance:

a. Security measures such as OAuth authentication, two-factor authentication (2FA), and API rate limiting help in protecting user data and preventing unauthorized access[11].

b. Techniques for mitigating Distributed Denial-of-Service (DDoS) assaults, such as rate limitation and CAPTCHA verification, protect against automated attacks[12, 17].

2.7.5 Market Making and Liquidity Providers:

a. Market makers play a key role in ensuring liquidity by constantly providing buy and sell orders.

b. To maximize market-making activities, minimize price slippage, and guarantee equitable order execution, a variety of algorithms and tactics are employed[13].

c. Compared to traditional exchanges, liquidity pools, which are frequently employed in decentralized finance (DeFi), offer an alternative method of market creation[14].

2.7.6 Trading Strategies and Algorithmic Trading:

a. Algorithmic trading relies on automated strategies that execute orders based on predefined rules.

b. High-frequency trading (HFT) requires low-latency execution and access to high-speed data feeds to make quick trading decisions[1].

c. Traders can use historical market data to test their methods via backtesting tools.

2.7.7 Cloud-Based Deployment and Scalability:

a. Cloud platforms such as Azure enable auto-scaling, ensuring the system adapts to fluctuations in trading volumes.

b. Load balancing ensures fair distribution of system requests, improving overall system reliability and uptime[15].

c. Serverless computing options, such as AWS Lambda and Azure Functions, are explored for event-driven trade execution and order processing[16].

Studies on financial trading systems indicate that high-frequency trading requires low-latency architectures and real-time data streaming mechanisms.

a. WebSocket-based streaming is widely adopted for real-time financial data.

b. Microservices-based backend architectures ensure scalability.

c. Security measures like OAuth authentication, two-factor authentication (2FA), and API rate limiting are crucial to protect user data.

d. Order book systems play a critical role in financial trading by matching buy and sell orders efficiently, optimizing liquidity, and reducing market volatility.

This project incorporates best practices from existing platforms and enhances them with modern technology stacks and an improved user experience.

3 Project Description

3.1 Architecture Overview

The trading platform follows a modular microservices architecture, that guarantees flexibility, scalability, and fault tolerance. The key components of the system include:

3.1.1 Frontend (Next.js + WebSockets)

a. The frontend is built using Next.js to provide a fast and interactive user experience.

b. WebSockets are integrated to handle real-time price updates, market depth visualization, and instant trade execution.

c. Users can place various types of orders (market, limit, stop-loss) through an intuitive UI.

3.1.2 Backend (Node.js + Express + WebSockets)

a. The backend is structured using a microservices model, with each service handling a specific task (such as user authentication, market data retrieval, order execution, etc.).

b. WebSocket-based market data streaming ensures that users receive up-to-date price information and trade executions instantly.

c. Order validation and execution logic are handled asynchronously for high-speed processing.

3.1.3 Order Matching Engine

a. The core of the trading platform is the order-matching engine, which processes buy and sell orders in real time.

b. Orders are stored in an in-memory order book component, ensuring ultra-fast trade execution.

c. Redis is used as a backup queue, preventing data loss in case of system failures.

3.1.4 Database Layer (PostgreSQL + Redis)

a. User portfolios, account information, and historical trade data are stored in PostgreSQL.

b. Redis queues act as a temporary cache for real-time order book snapshots, ensuring data persistence and quick access.

3.1.5 Security & Authentication

a. User authentication is managed through JWT-based authentication and OAuth integration for secure access.

b. Two-Factor Authentication (2FA) is used for additional security.

c. API rate limiting is implemented to prevent abuse and DDoS attacks.

3.1.6 Deployment & Scalability

a. The platform is deployed using containerized services (Docker + Kubernetes) for scalability and easy management.

b. Hosted on Azure (VMs, Load Balancer) to provide high availability and load balancing.

c. Auto-scaling mechanisms adjust the system resources dynamically based on trading volume.

3.2 Advantages of this Architecture

a. **Low Latency:** Transaction delays are minimized by keeping order book operations in memory.

b. **Fault Tolerance:** Order book data loss is prevented by Redis queue backup.

c. **Scalability:** Microservices enable horizontal scaling as demand increases.

d. **Security-First Approach:** Enforced authentication, encryption, and API protection mechanisms.

e. **Real-Time Data Processing:** WebSocket connection enables real-time market updates and trade execution.

This trading platform offers customers a safe, effective, and scalable trading environment that supports a variety of asset classes, including equities and cryptocurrencies, and is built to handle high-frequency trading (HFT) scenarios.

3.3 Technical Concepts

3.3.1 Tech Stack

a. **Frontend:** Next.js (React framework) for real-time user interactions.

b. **Backend:** Node.js with Express for handling API requests and WebSocket-based trade execution.

c. **Database:** PostgreSQL for persistent trade storage and Redis for high-speed order book caching.

d. **Messaging & Queues:** Redis Pub/Sub for event-driven architecture and message broadcasting.

e. **Security:** JWT authentication, OAuth, Two-Factor Authentication (2FA), and API rate limiting.

f. **Deployment:** Hosted on Azure using Virtual Machines (VMs) and Load Balancer for high availability.

3.3.2 Frontend Architecture

a. Built using Next.js, allowing server-side rendering (SSR) and optimized client performance.

b. Utilizes WebSockets for real-time order book updates and live market data.

c. UI designed with TailwindCSS for responsiveness and an intuitive trading experience.

3.3.3 Backend Architecture

a. Microservices-based backend using Node.js + Express to separate trade execution, user authentication, and data storage.

b. Order Matching Engine processes buy/sell orders and maintains the real-time order book in memory with Redis as a backup.

c. Market Data Service fetches live ticker prices, market depth, and trade history.

d. Event-driven architecture using Redis Pub/Sub ensures real-time trade execution updates.

3.3.4 Deployment and Scalability

a. Azure VMs & Load Balancer handle high trading volumes and prevent single points of failure.

b. Containerized deployment with Docker ensures consistency across environments.

c. Auto-scaling mechanisms adjust resources based on trading demand.

d. Monitoring tools like Prometheus and Grafana track system performance and trade latency.

3.4 UML Diagrams

3.4.1 Use Case

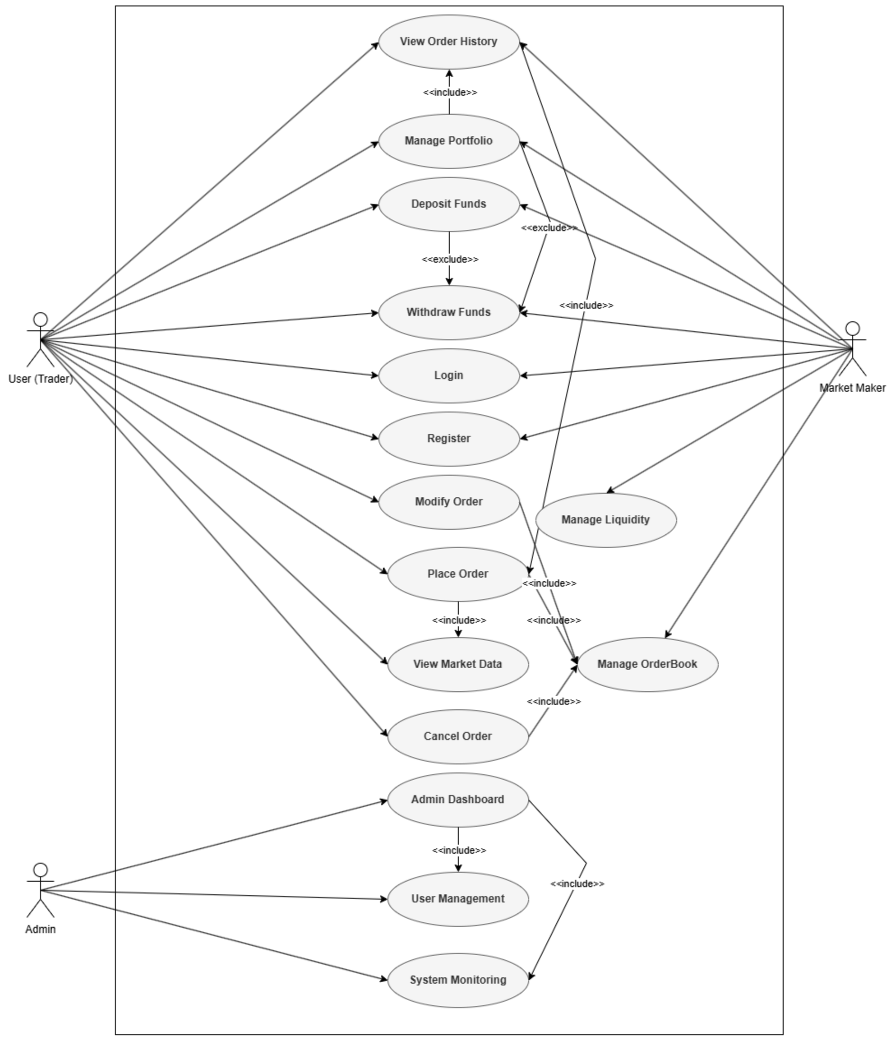


Figure 1: Use Case Diagram

3.4.2 Data Flow Diagram

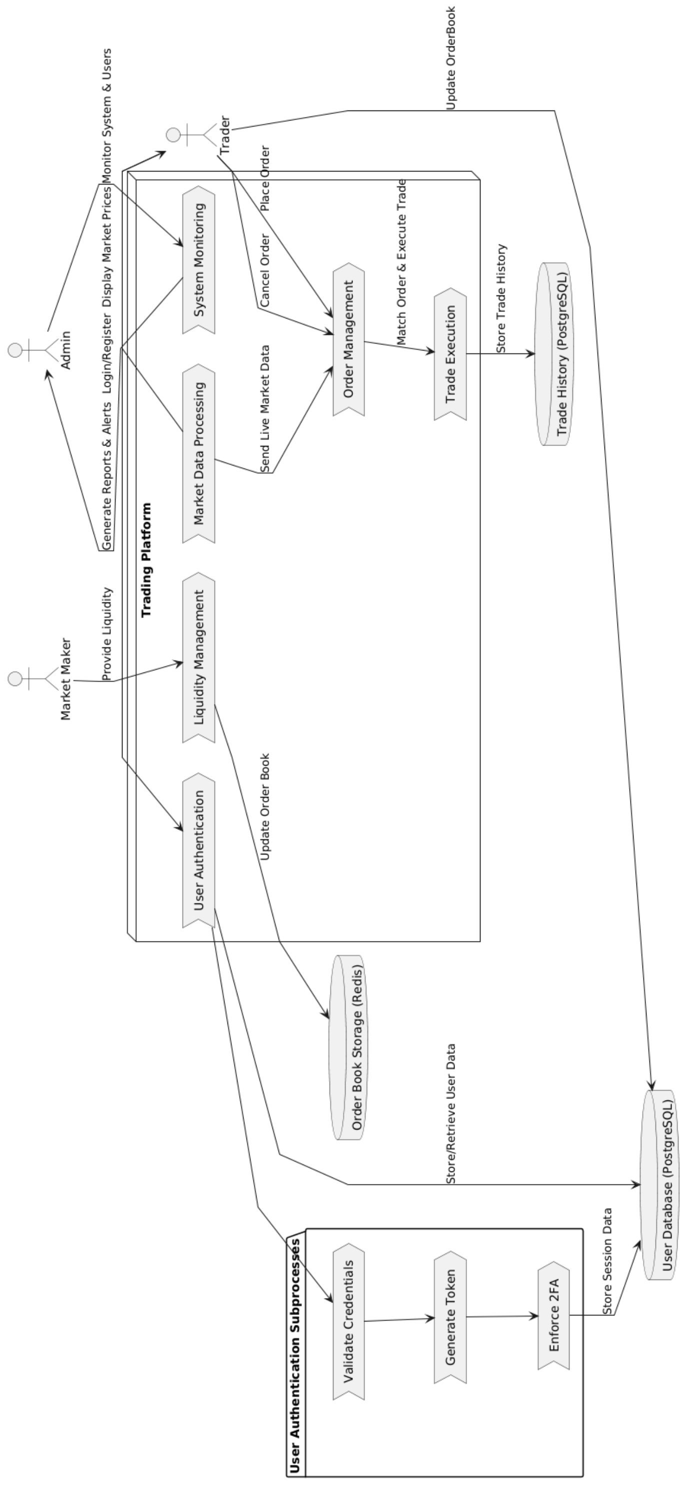


Figure 2: Data Flow Diagram

3.4.3 Class Diagram

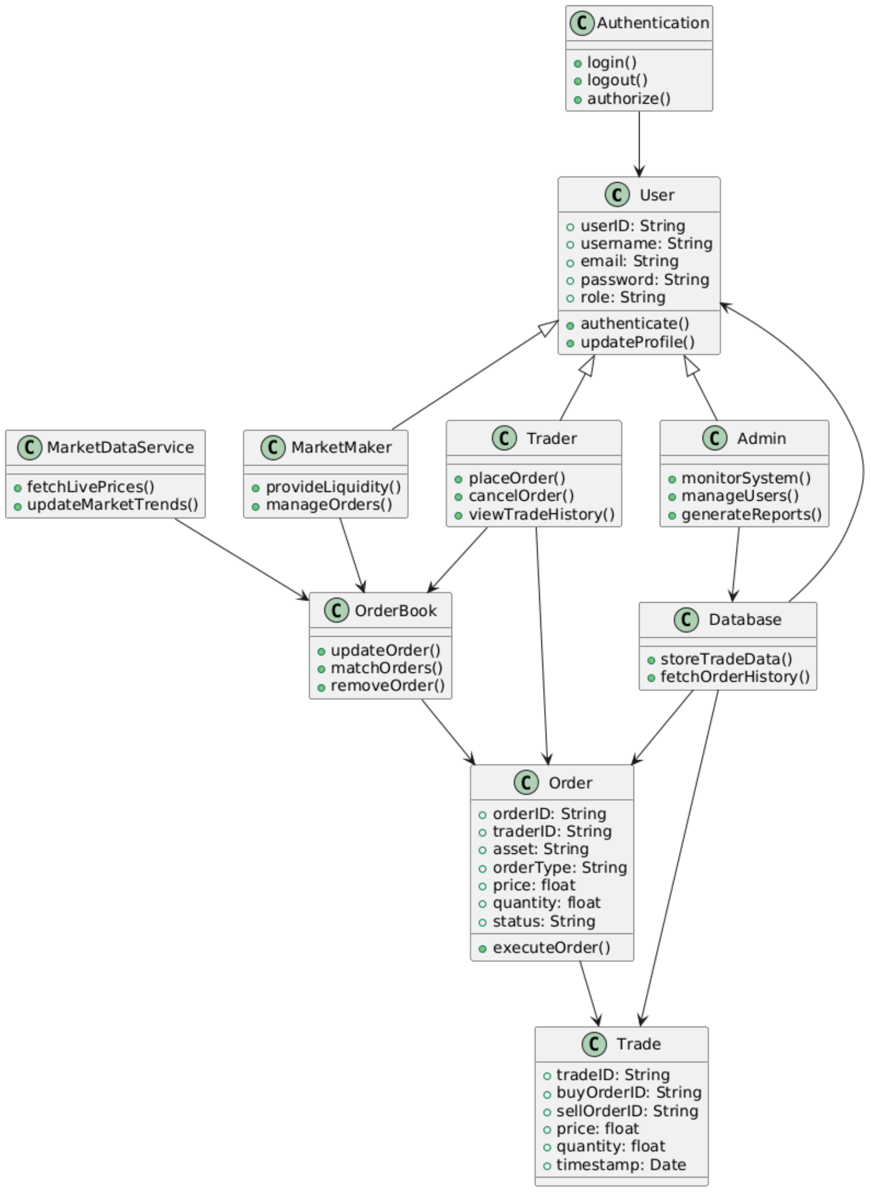


Figure 3: Class Diagram

3.4.4 Sequence Diagram

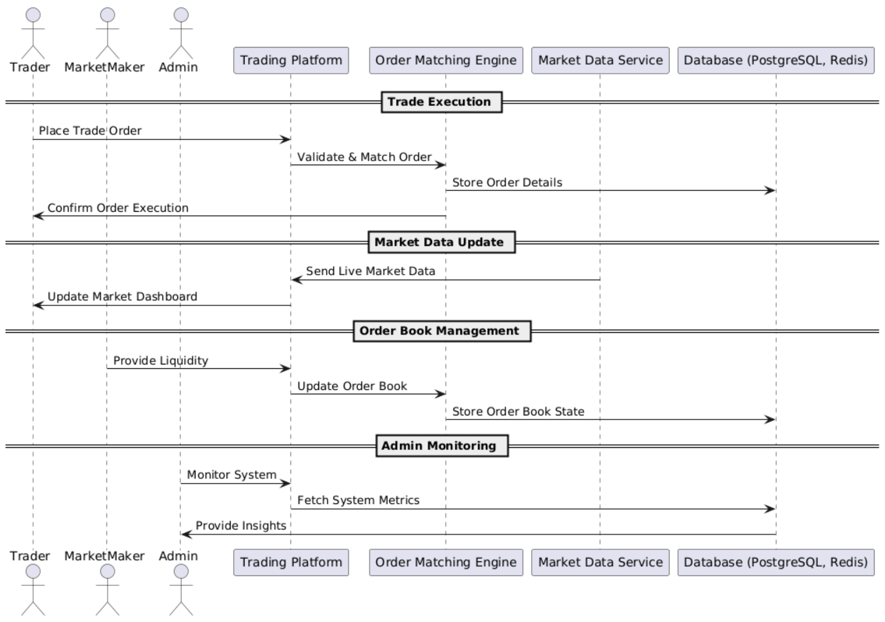
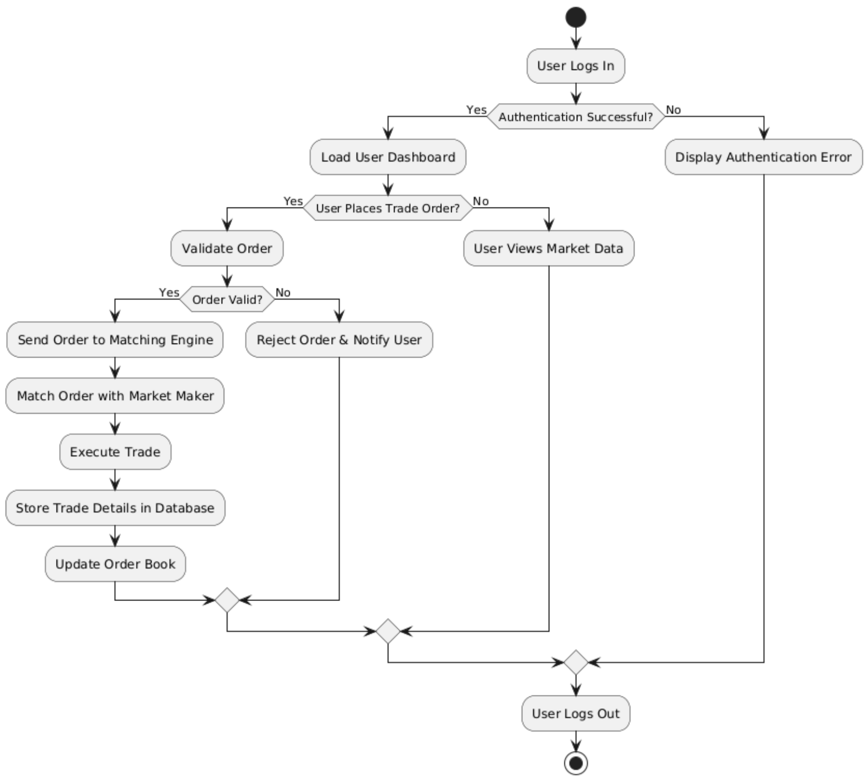


Figure 4: Sequence Diagram

3.4.5 Flow Chart



User Logs In
Authentication Successful?
Yes
No
Load User Dashboard
Display Authentication Error
User Places Trade Order?
Yes
No
Validate Order
User Views Market Data
Order Valid?
Yes
No
Send Order to Matching Engine
Reject Order & Notify User
Match Order with Market Maker
Execute Trade
Store Trade Details in Database
Update Order Book
User Logs Out

Figure 5: Flow Chart

3.4.6 Component Diagram

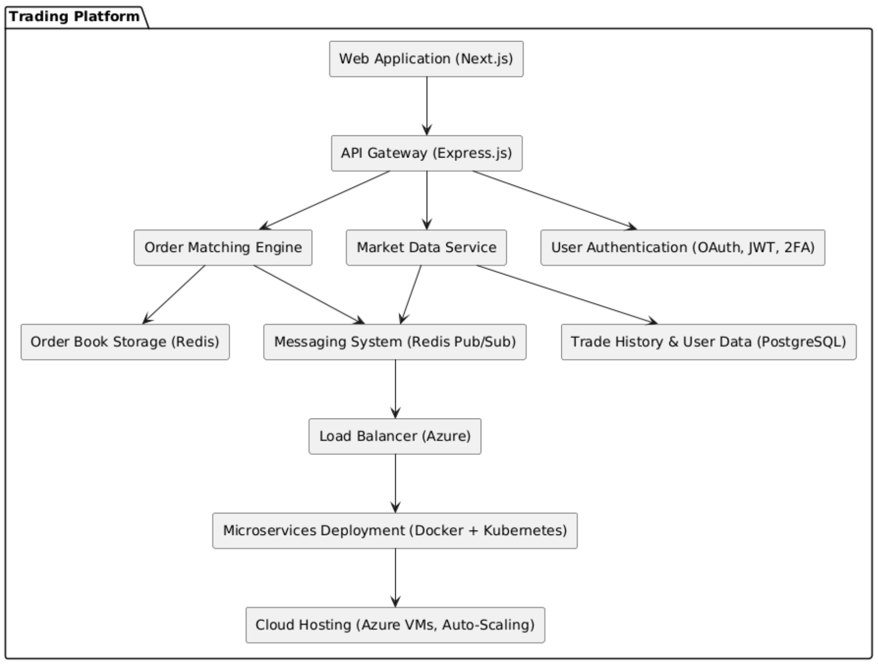

Trading Platform
Web Application (Next.js)
API Gateway (Express.js)
Order Matching Engine
Market Data Service
User Authentication (OAuth, JWT, 2FA)
Order Book Storage (Redis)
Messaging System (Redis Pub/Sub)
Trade History & User Data (PostgreSQL)
Load Balancer (Azure)
Microservices Deployment (Docker + Kubernetes)
Cloud Hosting (Azure VMs, Auto-Scaling)

Figure 6:

3.5 Swot Analysis

Figure 7 illustrates the SWOT analysis for the project. It highlights the project's Strengths, Weaknesses, Opportunities, and Threats providing an evaluation of internal and external factors. This analysis helps in strategic planning and risk management, ensuring the platform's long-term success.

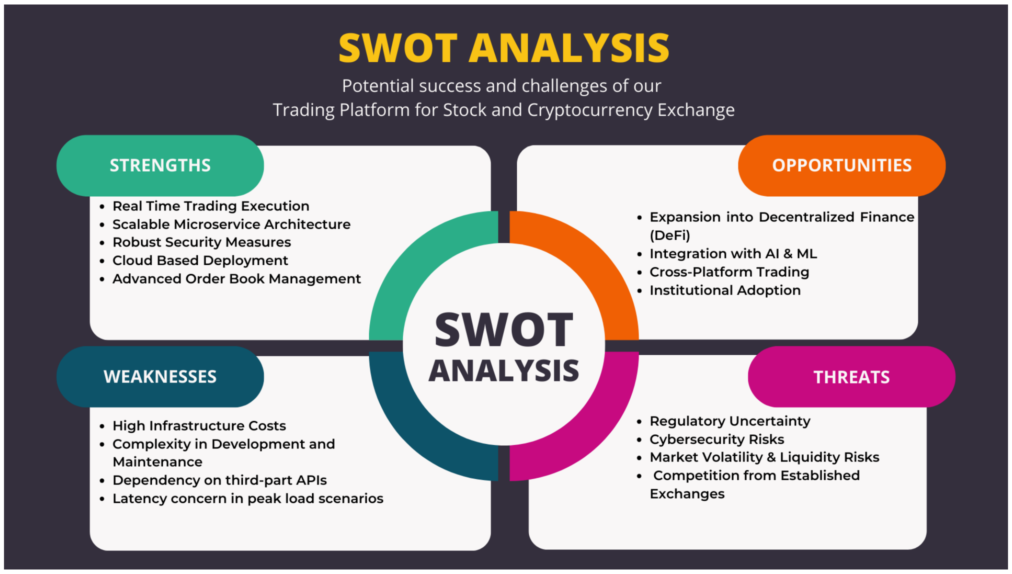

SWOT ANALYSIS
Potential success and challenges of our Trading Platform for Stock and Cryptocurrency Exchange
STRENGTHS
• Real Time Trading Execution
• Scalable Microservice Architecture
• Robust Security Measures
• Cloud Based Deployment
• Advanced Order Book Management
OPPORTUNITIES
• Expansion into Decentralized Finance (DeFi)
• Integration with AI & ML
• Cross-Platform Trading
• Institutional Adoption
SWOT ANALYSIS
WEAKNESSES
• High Infrastructure Costs
• Complexity in Development and Maintenance
• Dependency on third-part APIs
• Latency concern in peak load scenarios
THREATS
• Regulatory Uncertainty
• Cybersecurity Risks
• Market Volatility & Liquidity Risks
• Competition from Established Exchanges

Figure 7: SWOT Analysis

3.6 Methodology (Extreme Programming (XG) Agile Model)

Extreme Programming (XP) - Extreme Programming (XP) is an Agile software development methodology that emphasizes:

a. Customer satisfaction through regular feedback from peers,

b. Technical excellence through engineering practices,

c. Close collaboration among team members,

d. Continuous testing, feedback, and improvement.

Project Characteristics	**XP Relevance**
Real-time performance requirements	XP emphasizes optimized, clean code and frequent testing to ensure responsiveness.
Continuous feature additions (e.g., stop-loss, live charting)	XP supports short development cycles with continuous integration.
High code quality and low latency is critical	XP uses Test-Driven Development (TDD) and refactoring to keep the codebase clean and efficient.
Risk of downtime or failure (order loss, price mismatch)	XP emphasizes automated testing, pair programming, and collective ownership to reduce bugs.
Strong developer collaboration needed (WebSockets, Redis, UI)	XP promotes pair programming, shared responsibility, and communication.

3.6.1 Phase 1: Requirement Analysis and Planning

a. Identifying key functionalities such as order book management, market data processing, and real-time trade execution.

b. Researching industry-standard security measures including OAuth authentication, two-factor authentication (2FA), and API rate limiting.

c. Evaluating cloud deployment strategies to ensure high availability and scalability.

3.6.2 Phase 2: System Architecture Design

a. Designing a microservices-based architecture to separate concerns such as order execution, market data handling, and user authentication.

b. Selecting Next.js for the frontend to optimize performance and support server-side rendering (SSR).

c. Implementing an event-driven architecture using Redis Pub/Sub for real-time data updates.

d. Building the backend with Node.js, Express, and WebSockets to handle high-frequency trading.

e. Integrating PostgreSQL for trade history and Redis for caching live order book data.

f. Using Azure Virtual Machines and Load Balancer for cloud deployment and scalability.

3.6.3 Phase 3: Frontend Development

a. Developing a user-friendly interface with Next.js and TailwindCSS.

b. Implementing real-time market updates using WebSockets for a seamless trading experience.

c. Creating responsive and interactive dashboards for portfolio management, live order book, and trade execution.

3.6.4 Phase 4: Backend Development

Order Matching Engine

a. The order matching engine is the core component responsible for executing trades by matching buy and sell orders.

b. Uses an in-memory order book for fast order execution, reducing database query delays.

c. **Matching Algorithm:**

    i. Price-Time Priority Algorithm: Orders are executed based on price priority, followed by time priority.

    ii. FIFO (First In, First Out) Algorithm: Ensures older orders get executed before newer ones at the same price level.

    iii. Maker-Taker Model: Benefits liquidity providers (makers) by charging lower fees while applying higher fees to liquidity takers.

    iv. Market Impact Calculation: Prevents high slippage and ensures optimal order execution.

d. **Types of Orders Supported:**

    i. Market Orders: Execute immediately at the best available price.

    ii. Limit Orders: Execute only at a specified price or better.

    iii. Stop-Loss Orders: Automatically trigger an order when a specific price threshold is met.

Market Data Service

a. Provides real-time price feeds, order book depth, trade history, and liquidity analysis.

b. Implements WebSocket API endpoints to stream real-time updates to traders.

c. Retrieves external market data from Binance, Coinbase, and stock market exchanges.

Security and Authentication

a. Implements JWT-based authentication and OAuth integration for secure identity management.

b. Enforces Two-Factor Authentication (2FA) for account security.

c. Uses Role-Based Access Control (RBAC) for user permissions.

d. Implements API rate limiting and DDoS protection to prevent malicious activities.

e. Encrypts sensitive data using AES-256 encryption.

Trade Execution and Settlement

a. Uses atomic transaction handling to ensure complete and fail-safe trade execution.

b. Implements failure recovery mechanisms to rollback failed trades and prevent inconsistencies.

c. Provides detailed trade logs and audit trails to ensure transparency and compliance

Scalability and Performance Optimization

a. Implements Redis-based caching for high-speed order book retrieval and trade processing.

b. Uses Kafka for event-driven trade execution and to distribute trade-related events across microservices.

c. Auto-scaling mechanisms are implemented using Azure Kubernetes Service (AKS) to dynamically allocate resources based on traffic demands.

d. Optimized WebSocket connections to efficiently handle thousands of concurrent users without latency issues.

    i. Maintain order book integrity in case of failures.

    ii. Supporting different order types: market orders, limit orders, and stop-loss orders.

e. Market Data Service:

    i. Fetching real-time price updates and market depth information

    ii. Providing API endpoints for order book data and recent trades.

3.6.5 Phase 5: Integration & Testing

a. Unit Testing: Testing individual components, such as the order matching engine and API endpoints.

b. Integration Testing: Ensuring seamless interaction between frontend, backend, and third-party APIs.

c. Security Audits: Conducting penetration tests to identify vulnerabilities.

d. Load Testing: Simulating high-frequency trading scenarios to evaluate system performance under heavy traffic.

3.6.6 Phase 6: Deployment & Monitoring

a. Containerizing applications using Docker and Kubernetes for efficient deployment.

b. Hosting on Azure VMs with Load Balancer for fault tolerance and high availability.

c. Setting up CI/CD pipelines for automated builds and deployments.

d. Using Prometheus and Grafana for real-time monitoring and alerting on system performance.

3.6.7 Phase 7: Maintenance & Continuous Improvement

a. Regularly updating system components to ensure security compliance and performance enhancements.

b. Collecting user feedback for UI/UX improvements and feature enhancements.

c. Scaling infrastructure dynamically based on trading volume and demand.

d. Implementing machine learning models to predict market trends and assist in algorithmic trading.

3.6.8 Phase 8: Documentation

a. **Code Documentation:** Thoroughly comment the code to explain functionality.

b. **User Manual:** Guides users on how to navigate and use the trading platform.

c. **Final Report:** Summarizes project objectives, methodology, challenges, and outcomes.

3.6.9 Phase 9: Final Review

a. Review: Go through the entire codebase to ensure all features are properly implemented.

b. Refinement: Make necessary adjustments based on review feedback.

3.7 PERT Chart

Figure 8 illustrates the project schedule and dependencies using a PERT chart. The tasks and dependencies are visually represented, ensuring a structured workflow.

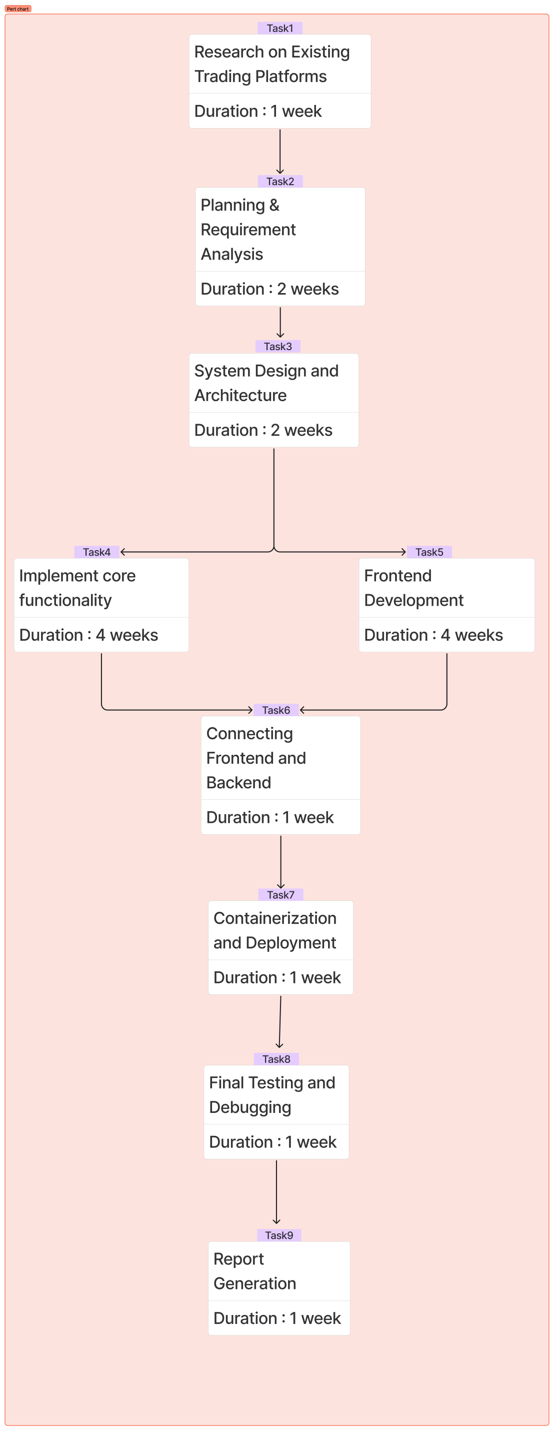


Figure 8: PERT Chart showing task dependencies and schedule

4 Implementation

4.1 Project Structure

Our project is organized into a modular, scalable, and maintainable monorepo architecture. Each major functionality—such as the API layer, matching engine, database operations, frontend, and WebSocket server—is encapsulated in its own directory. The structure facilitates separation of concerns, easy testing, and independent development of components.

4.1.1 Root-Level Overview

- `api/`: Handles REST API routes and communication with the matching engine via Redis.
- `db/`: Contains database logic for seeding data, refreshing views, and processing trade records via Redis.
- `docker/`: Holds the Docker Compose file for setting up services like PostgreSQL and Redis locally.
- `engine/`: Implements the core matching engine and order management logic in TypeScript.
- `frontend/`: A Next.js + Tailwind CSS application that renders the trading interface for users.
- `mm/`: Market-making bot logic (scaffolding provided).
- `ws/`: WebSocket server logic managing real-time subscriptions and user connections.

4.1.2 Detailed Directory Descriptions

`api/`

- Implements Express.js routes for orders, tickers, market depth, and Kline data.
- Communicates asynchronously with the engine via Redis pub/sub and list queues.
- Routes are organized under `src/routes/`.
- Uses TypeScript and compiles into `dist/`.

`db/`

- Includes scripts like `seed-db.ts` to initialize database schemas and materialized views.
- `cron.ts` refreshes Kline views periodically.
- `index.ts` listens to Redis queue `db_processor` to persist trades.
- All PostgreSQL credentials are loaded from the `.env` file (not hardcoded).

`docker/`

- Contains `docker-compose.yml` to orchestrate Redis, PostgreSQL, and potentially TimescaleDB services locally.

`engine/`

- The matching logic is written in `Engine.ts`.
- Uses in-memory orderbooks, snapshot saving, and Redis messaging for communication.
- Includes test cases under `src/tests/`.

`frontend/`

- Built with React (Next.js) and styled using Tailwind CSS.
- Pages are defined in `app/`, while reusable components are organized under `components/`.
- Contains utilities for HTTP/WS client connections, chart management, and layout rendering.

`mm/`

- Placeholder for market maker logic; basic file structure prepared for future implementation.

`ws/`

- Contains the WebSocket server logic handling:
  - User session management (`User.ts`, `UserManager.ts`)
  - Channel subscriptions (`SubscriptionManager.ts`)
  - Message routing based on user subscriptions
- Integrates with Redis pub/sub for real-time data broadcasting.

4.2 Code Explanation

4.2.1 Engine (Backend)

```
import fs from "fs";
import { RedisManager } from "../RedisManager";
import { ORDER_UPDATE, TRADE_ADDED } from "../types/index";
import { CANCEL_ORDER, CREATE_ORDER, GET_DEPTH, GET_OPEN_ORDERS, MessageFromApi, ON_RAMP }
    from "../types/fromApi";
import { Fill, Order, Orderbook } from "./Orderbook";

export const BASE_CURRENCY = "INR";

interface UserBalance {
    [key: string]: {
        available: number;
        locked: number;
    }
}

export class Engine {
    private orderbooks: Orderbook[] = [];
    private balances: Map<string, UserBalance> = new Map();

    constructor() {
        let snapshot = null
        try {
            if (process.env.WITH_SNAPSHOT) {
                snapshot = fs.readFileSync("./snapshot.json");
            }
        } catch (e) {
            console.log("No snapshot found");
        }

        if (snapshot) {
            const snapshotSnapshot = JSON.parse(snapshot.toString());
            this.orderbooks = snapshotSnapshot.orderbooks.map((o: any) => new Orderbook(o.
    baseAsset, o.bids, o.asks, o.lastTradeId, o.currentPrice));
            this.balances = new Map(snapshotSnapshot.balances);
        } else {
            this.orderbooks = [new Orderbook(`TATA`, [], [], 0, 0)];
            this.setBaseBalances();
        }
        setInterval(() => {
            this.saveSnapshot();
        }, 1000 * 3);
    }

    saveSnapshot() {
        const snapshotSnapshot = {
            orderbooks: this.orderbooks.map(o => o.getSnapshot()),
            balances: Array.from(this.balances.entries())
        }
        fs.writeFileSync("./snapshot.json", JSON.stringify(snapshotSnapshot));
    }

    process({ message, clientId }: {message: MessageFromApi, clientId: string}) {
        switch (message.type) {
            case CREATE_ORDER:
                try {
                    const { executedQty, fills, orderId } = this.createOrder(message.data.
    market, message.data.price, message.data.quantity, message.data.side, message.data.
    userId);
                    RedisManager.getInstance().sendToApi(clientId, {
                        type: "ORDER_PLACED",
                        payload: {
                            orderId,
                            executedQty,
                            fills
                        }
                    });
                } catch (e) {
                    console.log(e);
                    RedisManager.getInstance().sendToApi(clientId, {
                        type: "ORDER_CANCELLED",
                        payload: {
                            orderId: "",
                            executedQty: 0,
                            remainingQty: 0
                        }
                    });
                }
                break;
            case CANCEL_ORDER:
                try {
                    const orderId = message.data.orderId;
                    const cancelMarket = message.data.market;
                    const cancelOrderbook = this.orderbooks.find(o => o.ticker() === cancelMarket);
                    const quoteAsset = cancelMarket.split("_")[1];
                    if (!cancelOrderbook) {
                        throw new Error("No orderbook found");
                    }

                    const order = cancelOrderbook.asks.find(o => o.orderId === orderId) || cancelOrderbook.bids.find(o => o.orderId === orderId);
                    if (!order) {
                        console.log("No order found");
                        throw new Error("No order found");
                    }

                    if (order.side === "buy") {
                        const price = cancelOrderbook.cancelBid(order)
                        const leftQuantity = (order.quantity - order.filled) * order.price;
                        //@ts-ignore
                        this.balances.get(order.userId)[BASE_CURRENCY].available += leftQuantity;
                        //@ts-ignore
                        this.balances.get(order.userId)[BASE_CURRENCY].locked -= leftQuantity;
                        if (price) {
                            this.sendUpdatedDepthAt(price.toString(), cancelMarket);
                        }
                    } else {
                        const price = cancelOrderbook.cancelAsk(order)
                        const leftQuantity = order.quantity - order.filled;
                        //@ts-ignore
                        this.balances.get(order.userId)[quoteAsset].available += leftQuantity;
                        //@ts-ignore
                        this.balances.get(order.userId)[quoteAsset].locked -= leftQuantity;
                        if (price) {
                            this.sendUpdatedDepthAt(price.toString(), cancelMarket);
                        }
                    }

                    RedisManager.getInstance().sendToApi(clientId, {
                        type: "ORDER_CANCELLED",
                        payload: {
                            orderId,
                            executedQty: 0,
                            remainingQty: 0
                        }
                    });

                } catch (e) {
                    console.log("Error hwile cancelling order", );
                    console.log(e);
                }
                break;
            case GET_OPEN_ORDERS:
                try {
                    const openOrderbook = this.orderbooks.find(o => o.ticker() === message.
    data.market);
                    if (!openOrderbook) {
                        throw new Error("No orderbook found");
                    }
                    const openOrders = openOrderbook.getOpenOrders(message.data.userId);

                    RedisManager.getInstance().sendToApi(clientId, {
                        type: "OPEN_ORDERS",
                        payload: openOrders
                    });
                } catch(e) {
                    console.log(e);
                }
                break;
            case ON_RAMP:
                const userId = message.data.userId;
                const amount = Number(message.data.amount);
                this.onRamp(userId, amount);
                break;
            case GET_DEPTH:
                try {
                    const market = message.data.market;
                    const orderbook = this.orderbooks.find(o => o.ticker() === market);
                    if (!orderbook) {
                        throw new Error("No orderbook found");
                    }
                    RedisManager.getInstance().sendToApi(clientId, {
                        type: "DEPTH",
                        payload: orderbook.getDepth()
                    });
                } catch (e) {
                    console.log(e);
                    RedisManager.getInstance().sendToApi(clientId, {
                        type: "DEPTH",
                        payload: {
                            bids: [],
                            asks: []
                        }
                    });
                }
                break;
        }
    }

    addOrderbook(orderbook: Orderbook) {
        this.orderbooks.push(orderbook);
    }

    createOrder(market: string, price: string, quantity: string, side: "buy" | "sell",
    userId: string) {

        const orderbook = this.orderbooks.find(o => o.ticker() === market)
```

```
        const baseAsset = market.split("_")[0];
        const quoteAsset = market.split("_")[1];

        if (!orderbook) {
            throw new Error("No orderbook found");
        }

        this.checkAndLockFunds(baseAsset, quoteAsset, side, userId, quoteAsset, price,
    quantity);

        const order: Order = {
            price: Number(price),
            quantity: Number(quantity),
            orderId: Math.random().toString(36).substring(2, 15) + Math.random().toString
    (36).substring(2, 15),
            filled: 0,
            side,
            userId
        }

        const { fills, executedQty } = orderbook.addOrder(order);
        this.updateBalance(userId, baseAsset, quoteAsset, side, fills, executedQty);

        this.createDbTrades(fills, market, userId);
        this.updateDbOrders(order, executedQty, fills, market);
        this.publisWsDepthUpdates(fills, price, side, market);
        this.publishWsTrades(fills, userId, market);
        return { executedQty, fills, orderId: order.orderId };
    }

    updateDbOrders(order: Order, executedQty: number, fills: Fill[], market: string) {
        RedisManager.getInstance().pushMessage({
            type: ORDER_UPDATE,
            data: {
                orderId: order.orderId,
                executedQty: executedQty,
                market: market,
                price: order.price.toString(),
                quantity: order.quantity.toString(),
                side: order.side,
            }
        });

        fills.forEach(fill => {
            RedisManager.getInstance().pushMessage({
                type: ORDER_UPDATE,
                data: {
                    orderId: fill.markerOrderId,
                    executedQty: fill.qty
                }
            });
        });
    }

    createDbTrades(fills: Fill[], market: string, userId: string) {
        fills.forEach(fill => {
            RedisManager.getInstance().pushMessage({
                type: TRADE_ADDED,
                data: {
                    market: market,
                    id: fill.tradeId.toString(),
                    isBuyerMaker: fill.otherUserId === userId, // TODO: Is this right?
                    price: fill.price,
                    quantity: fill.qty.toString(),
                    quoteQuantity: (fill.qty * Number(fill.price)).toString(),
                    timestamp: Date.now()
                }
            });
        });
    }

    publishWsTrades(fills: Fill[], userId: string, market: string) {
        fills.forEach(fill => {
            RedisManager.getInstance().publishMessage(`trade@${market}`, {
                stream: `trade@${market}`,
                data: {
                    e: "trade",
                    t: fill.tradeId,
                    m: fill.otherUserId === userId, // TODO: Is this right?
                    p: fill.price,
                    q: fill.qty.toString(),
                    s: market,
                }
            });
        });
    }

    sendUpdatedDepthAt(price: string, market: string) {
        const orderbook = this.orderbooks.find(o => o.ticker() === market);
        if (!orderbook) {
            return;
        }
        const depth = orderbook.getDepth();
        const updatedBids = depth?.bids.filter(x => x[0] === price);
        const updatedAsks = depth?.asks.filter(x => x[0] === price);

        RedisManager.getInstance().publishMessage(`depth@${market}`, {
            stream: `depth@${market}`,
            data: {
                a: updatedAsks.length ? updatedAsks : [[price, "0"]],
                b: updatedBids.length ? updatedBids : [[price, "0"]],
                e: "depth"
            }
        });
    }

    publisWsDepthUpdates(fills: Fill[], price: string, side: "buy" | "sell", market: string)
 {
        const orderbook = this.orderbooks.find(o => o.ticker() === market);
        if (!orderbook) {
            return;
        }
        const depth = orderbook.getDepth();
        if (side === "buy") {
```

```
        const updatedAsks = depth?.asks.filter(x => fills.map(f => f.price).includes(x[0].toString()));
        const updatedBid = depth?.bids.find(x => x[0] === price);
        console.log("publish ws depth updates")
        RedisManager.getInstance().publishMessage(`depth@${market}`, {
            stream: `depth@${market}`,
            data: {
                a: updatedAsks,
                b: updatedBid ? [updatedBid] : [],
                e: "depth"
            }
        });
    }
    if (side === "sell") {
       const updatedBids = depth?.bids.filter(x => fills.map(f => f.price).includes(x[0].toString()));
       const updatedAsk = depth?.asks.find(x => x[0] === price);
       console.log("publish ws depth updates")
       RedisManager.getInstance().publishMessage(`depth@${market}`, {
           stream: `depth@${market}`,
           data: {
               a: updatedAsk ? [updatedAsk] : [],
               b: updatedBids,
               e: "depth"
           }
       });
    }
}

updateBalance(userId: string, baseAsset: string, quoteAsset: string, side: "buy" | "sell", fills: Fill[], executedQty: number) {
    if (side === "buy") {
        fills.forEach(fill => {
            // Update quote asset balance
            //@ts-ignore
            this.balances.get(fill.otherUserId)[quoteAsset].available = this.balances.get(fill.otherUserId)?.[quoteAsset].available + (fill.qty * fill.price);

            //@ts-ignore
            this.balances.get(userId)[quoteAsset].locked = this.balances.get(userId)?.[quoteAsset].locked - (fill.qty * fill.price);

            // Update base asset balance

            //@ts-ignore
            this.balances.get(fill.otherUserId)[baseAsset].locked = this.balances.get(fill.otherUserId)?.[baseAsset].locked - fill.qty;

            //@ts-ignore
            this.balances.get(userId)[baseAsset].available = this.balances.get(userId)?.[baseAsset].available + fill.qty;

        });

    } else {
        fills.forEach(fill => {
            // Update quote asset balance
```

```
                //@ts-ignore
                this.balances.get(fill.otherUserId)[quoteAsset].locked = this.balances.get(fill.otherUserId)?.[quoteAsset].locked - (fill.qty * fill.price);

                //@ts-ignore
                this.balances.get(userId)[quoteAsset].available = this.balances.get(userId)?.[quoteAsset].available + (fill.qty * fill.price);

                // Update base asset balance

                //@ts-ignore
                this.balances.get(fill.otherUserId)[baseAsset].available = this.balances.get(fill.otherUserId)?.[baseAsset].available + fill.qty;

                //@ts-ignore
                this.balances.get(userId)[baseAsset].locked = this.balances.get(userId)?.[baseAsset].locked - (fill.qty);

            });
        }
    }

    checkAndLockFunds(baseAsset: string, quoteAsset: string, side: "buy" | "sell", userId: string, asset: string, price: string, quantity: string) {
        if (side === "buy") {
            if ((this.balances.get(userId)?.[quoteAsset]?.available || 0) < Number(quantity) * Number(price)) {
                throw new Error("Insufficient funds");
            }
            //@ts-ignore
            this.balances.get(userId)[quoteAsset].available = this.balances.get(userId)?.[quoteAsset].available - (Number(quantity) * Number(price));

            //@ts-ignore
            this.balances.get(userId)[quoteAsset].locked = this.balances.get(userId)?.[quoteAsset].locked + (Number(quantity) * Number(price));
        } else {
            if ((this.balances.get(userId)?.[baseAsset]?.available || 0) < Number(quantity)) {
                throw new Error("Insufficient funds");
            }
            //@ts-ignore
            this.balances.get(userId)[baseAsset].available = this.balances.get(userId)?.[baseAsset].available - (Number(quantity));

            //@ts-ignore
            this.balances.get(userId)[baseAsset].locked = this.balances.get(userId)?.[baseAsset].locked + Number(quantity);
        }
    }

    onRamp(userId: string, amount: number) {
        const userBalance = this.balances.get(userId);
        if (!userBalance) {
            this.balances.set(userId, {
                [BASE_CURRENCY]: {
                    available: amount,
                    locked: 0
                }
            });
        } else {
            userBalance[BASE_CURRENCY].available += amount;
        }
    }

    setBaseBalances() {
        this.balances.set("1", {
            [BASE_CURRENCY]: {
                available: 10000000,
                locked: 0
            },
            "TATA": {
                available: 10000000,
                locked: 0
            }
        });

        this.balances.set("2", {
            [BASE_CURRENCY]: {
                available: 10000000,
                locked: 0
            },
            "TATA": {
                available: 10000000,
                locked: 0
            }
        });

        this.balances.set("5", {
            [BASE_CURRENCY]: {
                available: 10000000,
                locked: 0
            },
            "TATA": {
                available: 10000000,
                locked: 0
            }
        });
    }

}
```

Listing 1: Engine Implementation

The `Engine.ts` file is a core component of the backend system, implementing the **matching engine** that facilitates the core logic of the stock and cryptocurrency exchange platform. This engine is responsible for processing trade requests, managing orderbooks, updating balances, and communicating state changes via Redis.

Purpose

The `Engine` class handles:

- Order matching for different markets.

- Creation and cancellation of orders.
- Management of user balances (available vs locked).
- Broadcasting real-time updates to WebSocket clients.
- Periodic snapshot saving and recovery.

Functional Overview

Feature	**Description**
Orderbook Management	Maintains an array of `Orderbook` instances, one per market.
Balance Management	Tracks user balances in a `Map`, including available and locked funds.
Order Lifecycle	Supports order creation, cancellation, and open order retrieval.
Trade Settlement	Executes matched orders, updates balances, and logs trades.
Depth and Trade Broadcasting	Publishes orderbook depth and trade events to subscribed WebSocket streams.
Persistence	Saves orderbooks and balances to `snapshot.json` every 3 seconds.

Table 1: Matching Engine Functional Features

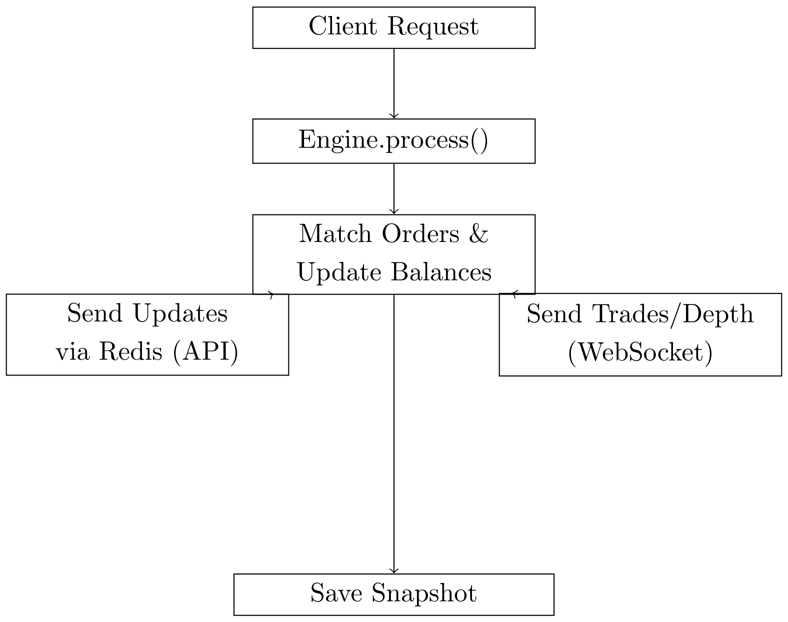
Workflow Description

1. On initialization, the engine attempts to load a saved snapshot of balances and orderbooks.
2. Incoming client messages are processed using the `process()` method.
3. Based on the message type:
   - `CREATE_ORDER`: Validates and locks user funds, places the order, performs matching, updates balances, and pushes updates to Redis.
   - `CANCEL_ORDER`: Cancels an order and unlocks any held funds.
   - `GET_OPEN_ORDERS`: Returns the list of all active orders for a user.
   - `GET_DEPTH`: Responds with the current orderbook depth for a specific market.
   - `ON_RAMP`: Credits fiat balance (INR) to a user's account.
4. Matched orders trigger real-time updates to WebSocket streams via Redis publish-subscribe.
5. Every 3 seconds, the engine persists its current state to disk.

Core Methods

- `createOrder()`: Places an order and performs trade matching.
- `updateBalance()`: Updates user balances post-trade.
- `checkAndLockFunds()`: Ensures sufficient funds before locking.
- `publishWsDepthUpdates()`, `publishWsTrades()`: Sends real-time data to clients.
- `saveSnapshot()`: Persists engine state periodically.

System Flow



Client Request
Engine.process()
Match Orders & Update Balances
Send Updates via Redis (API)
Send Trades/Depth (WebSocket)
Save Snapshot

The `Engine.ts` module serves as the heart of the trading platform by maintaining critical in-memory data structures, ensuring accurate order matching, and interacting with Redis for broadcasting system state. It integrates real-time responsiveness and persistence while providing a clear abstraction for trading operations.

4.2.2 Orderbook (Backend)

```
import { BASE_CURRENCY } from "./Engine";

export interface Order {
    price: number;
    quantity: number;
    orderId: string;
    filled: number;
    side: "buy" | "sell";
    userId: string;
}

export interface Fill {
    price: string;
    qty: number;
    tradeId: number;
    otherUserId: string;
    markerOrderId: string;
}

export class Orderbook {
    bids: Order[];
    asks: Order[];
    baseAsset: string;
    quoteAsset: string = BASE_CURRENCY;
    lastTradeId: number;
    currentPrice: number;

    constructor(baseAsset: string, bids: Order[], asks: Order[], lastTradeId: number,
    currentPrice: number) {
        this.bids = bids;
        this.asks = asks;
        this.baseAsset = baseAsset;
        this.lastTradeId = lastTradeId || 0;
        this.currentPrice = currentPrice ||0;
    }

    ticker() {
        return `${this.baseAsset}_${this.quoteAsset}`;
    }

    getSnapshot() {
        return {
            baseAsset: this.baseAsset,
            bids: this.bids,
            asks: this.asks,
            lastTradeId: this.lastTradeId,
            currentPrice: this.currentPrice
        }
    }

    addOrder(order: Order): {
        executedQty: number,
        fills: Fill[]
    } {
        if (order.side === "buy") {
            const {executedQty, fills} = this.matchBid(order);
            order.filled = executedQty;
            if (executedQty === order.quantity) {
                return { executedQty, fills }
            }
            this.bids.push(order);
            return { executedQty, fills }
        } else {
            const {executedQty, fills} = this.matchAsk(order);
            order.filled = executedQty;
            if (executedQty === order.quantity) {
                return { executedQty, fills }
            }
            this.asks.push(order);
            return { executedQty, fills }
        }
    }

    matchBid(order: Order): {fills: Fill[], executedQty: number} {
        const fills: Fill[] = [];
        let executedQty = 0;
        for (let i = 0; i < this.asks.length; i++) {
            if (this.asks[i].price <= order.price && executedQty < order.quantity) {
                const filledQty = Math.min((order.quantity - executedQty), this.asks[i].
    quantity);
                executedQty += filledQty;
                this.asks[i].filled += filledQty;
                fills.push({
                    price: this.asks[i].price.toString(),
                    qty: filledQty,
                    tradeId: this.lastTradeId++,
                    otherUserId: this.asks[i].userId,
                    markerOrderId: this.asks[i].orderId
                });
            }
        }
        for (let i = 0; i < this.asks.length; i++) {
            if (this.asks[i].filled === this.asks[i].quantity) {
                this.asks.splice(i, 1);
                i--;
            }
        }
        return { fills, executedQty };
    }

    matchAsk(order: Order): {fills: Fill[], executedQty: number} {
        const fills: Fill[] = [];
        let executedQty = 0;
        for (let i = 0; i < this.bids.length; i++) {
            if (this.bids[i].price >= order.price && executedQty < order.quantity) {
                const amountRemaining = Math.min(order.quantity - executedQty, this.bids[i].
    quantity);
                executedQty += amountRemaining;
                this.bids[i].filled += amountRemaining;
                fills.push({
                    price: this.bids[i].price.toString(),
                    qty: amountRemaining,
                    tradeId: this.lastTradeId++,
                    otherUserId: this.bids[i].userId,
                    markerOrderId: this.bids[i].orderId
                });
            }
        }
        for (let i = 0; i < this.bids.length; i++) {
            if (this.bids[i].filled === this.bids[i].quantity) {
                this.bids.splice(i, 1);
                i--;
            }
        }
        return { fills, executedQty };
    }

    getDepth() {
        const bids: [string, string][] = [];
        const asks: [string, string][] = [];

        const bidsObj: {[key: string]: number} = {};
        const asksObj: {[key: string]: number} = {};

        for (let i = 0; i < this.bids.length; i++) {
            const order = this.bids[i];
            if (!bidsObj[order.price]) bidsObj[order.price] = 0;
            bidsObj[order.price] += order.quantity;
        }

        for (let i = 0; i < this.asks.length; i++) {
            const order = this.asks[i];
            if (!asksObj[order.price]) asksObj[order.price] = 0;
            asksObj[order.price] += order.quantity;
        }

        for (const price in bidsObj) bids.push([price, bidsObj[price].toString()]);
        for (const price in asksObj) asks.push([price, asksObj[price].toString()]);

        return { bids, asks };
    }

    getOpenOrders(userId: string): Order[] {
        const asks = this.asks.filter(x => x.userId === userId);
        const bids = this.bids.filter(x => x.userId === userId);
        return [...asks, ...bids];
    }

    cancelBid(order: Order) {
        const index = this.bids.findIndex(x => x.orderId === order.orderId);
        if (index !== -1) {
            const price = this.bids[index].price;
            this.bids.splice(index, 1);
            return price
        }
    }

    cancelAsk(order: Order) {
        const index = this.asks.findIndex(x => x.orderId === order.orderId);
        if (index !== -1) {
            const price = this.asks[index].price;
            this.asks.splice(index, 1);
            return price
        }
    }
}
```

Listing 2: Orderbook

The `Orderbook.ts` module defines the data structures and logic necessary to manage market-specific orderbooks within the trading engine. This class is tightly integrated with the `Engine.ts` file and is responsible for recording, matching, and managing user orders in a given trading pair (e.g., TATA/INR).

Purpose

The `Orderbook` class manages two primary lists of orders:

- `bids`: Buy orders placed by users.
- `asks`: Sell orders placed by users.

It provides functionality to:

- Add and match new orders.
- Maintain market depth (aggregated bid/ask data).
- Retrieve open orders for users.

- Cancel specific orders.

Key Data Structures

- **Order**: Represents a limit order, containing fields like price, quantity, side (buy/sell), orderId, and userId.
- **Fill**: Represents an execution result of a matched order (i.e., a trade).

Constructor

- Initializes the orderbook with predefined bids, asks, and base asset.
- Each orderbook is specific to one trading pair (e.g., `TATA_INR`).
- The quote asset is set to a constant (`INR`), defined in `Engine.ts`.

Core Functionalities

- `addOrder(order)`:
  - Checks the order side (buy/sell) and delegates it to either `matchBid()` or `matchAsk()`.
  - If partially filled, adds the order to the orderbook.
- `matchBid(order)`:
  - Tries to fulfill a buy order by matching it with the lowest-priced asks.
  - Returns the total executed quantity and a list of `Fill` objects.
- `matchAsk(order)`:
  - Fulfills a sell order by matching it against the highest-priced bids.
  - Returns the trade fills and executed amount.
- `getDepth()`:
  - Aggregates bid and ask quantities at each price level.
  - Returns structured data used to display market depth in UI.
- `getOpenOrders(userId)`:
  - Returns all open (unfulfilled) bids and asks for a specific user.
- `cancelBid(order)` / `cancelAsk(order)`:
  - Finds and removes an existing order from the bids or asks list.
  - Returns the price of the removed order.

Matching Logic

- The matching process is linear and iterates through the opposing side of the orderbook.
- Matching is based on:
  - `price compatibility`: Bids must be greater than or equal to the ask for a match.
  - `quantity availability`: Orders are matched until one is filled.
- Once an order is fully matched, it is removed from the orderbook.

Example: Buy Order Matching

For a new buy order:

1. It checks existing `asks`.
2. If ask price $\leq$ bid price, the match is executed.
3. Fill details are recorded including matched quantity and price.
4. If the incoming order is partially matched, the remainder is added to `bids`.

The `Orderbook.ts` module plays a vital role in maintaining an accurate and efficient record of all market orders. It provides essential mechanisms for order matching, depth aggregation, and user-specific queries—functions critical to the integrity and speed of the exchange.

4.2.3 RedisManager (Backend)

```
import { DEPTH_UPDATE, TICKER_UPDATE } from "./trade/events";
import { RedisClientType, createClient } from "redis";
import { ORDER_UPDATE, TRADE_ADDED } from "./types";
import { WsMessage } from "./types/toWs";
import { MessageToApi } from "./types/toApi";

type DbMessage = {
    type: typeof TRADE_ADDED,
    data: {
        id: string,
        isBuyerMaker: boolean,
        price: string,
        quantity: string,
        quoteQuantity: string,
        timestamp: number,
        market: string
    }
} | {
    type: typeof ORDER_UPDATE,
    data: {
        orderId: string,
        executedQty: number,
        market?: string,
        price?: string,
        quantity?: string,
        side?: "buy" | "sell",
    }
}

export class RedisManager {
    private client: RedisClientType;
    private static instance: RedisManager;

    constructor() {
        this.client = createClient();
        this.client.connect();
    }

    public static getInstance() {
        if (!this.instance)  {
            this.instance = new RedisManager();
        }
        return this.instance;
    }

    public pushMessage(message: DbMessage) {
        this.client.lPush("db_processor", JSON.stringify(message));
    }

    public publishMessage(channel: string, message: WsMessage) {
        this.client.publish(channel, JSON.stringify(message));
    }

    public sendToApi(clientId: string, message: MessageToApi) {
        this.client.publish(clientId, JSON.stringify(message));
    }
}
```

Listing 3: RedisManager

The `RedisManager.ts` module abstracts and centralizes all interactions with the Redis server used for communication between various components of the exchange platform. Redis acts as a backbone for message passing, data queuing, and real-time publish-subscribe functionality.

Purpose

The main objective of the `RedisManager` class is to:

- Enable a unified and reusable interface to the Redis database.
- Push data to specific queues for background processing.
- Broadcast real-time updates via publish-subscribe channels.
- Send responses from the engine to connected API/WebSocket clients.

Key Responsibilities

Function	Description
`pushMessage()`	Sends trade or order data to a Redis queue (`db_processor`) for processing or persistence.
`publishMessage()`	Publishes formatted WebSocket messages to specific market channels (e.g., `depth@TATA_INR`).
`sendToApi()`	Sends direct messages to API clients using their unique client ID as the Redis channel name.

Table 2: RedisManager Method Overview

Implementation Details

- **Singleton Pattern**: The class implements a singleton pattern through the static `getInstance()` method. This ensures that only one Redis connection exists throughout the application lifecycle, reducing resource overhead.
- **Redis Client Initialization**:
  - The Redis client is created via `createClient()`.
  - The connection to Redis is initiated by calling `connect()` in the constructor.
- **Message Types (DbMessage)**:
  - `TRADE_ADDED`: Used to push trade execution information such as price, quantity, and trade time.
  - `ORDER_UPDATE`: Used to notify the system of order state changes like quantity filled, status, and side.

Role in the System Architecture

- **Decoupling Engine and Persistence**: Trade and order messages are placed on the `db_processor` queue, allowing a separate background service to store them in a database asynchronously.
- **Real-time Updates**: Market updates (orderbook depth, ticker changes, trades) are published to Redis channels. WebSocket servers subscribe to these channels and forward updates to connected clients.
- **Client Communication**: API responses (e.g., confirmation of order placement or errors) are sent to clients using their unique IDs as Redis pub/sub channels.

The `RedisManager.ts` module provides a clean and efficient interface for message-driven communication. It plays a central role in enabling scalable, real-time data flow across the system while separating concerns between components. Its singleton design ensures minimal resource use while facilitating high-throughput messaging critical for exchange performance.

4.2.4 User (WebSockets)

```
import { WebSocket } from "ws";
import { OutgoingMessage } from "./types/out";
import { SubscriptionManager } from "./SubscriptionManager";
import { IncomingMessage, SUBSCRIBE, UNSUBSCRIBE } from "./types/in";
```

```

export class User {
    private id: string;
    private ws: WebSocket;

    constructor(id: string, ws: WebSocket) {
        this.id = id;
        this.ws = ws;
        this.addListeners();
    }

    private subscriptions: string[] = [];

    public subscribe(subscription: string) {
        this.subscriptions.push(subscription);
    }

    public unsubscribe(subscription: string) {
        this.subscriptions = this.subscriptions.filter(s => s !== subscription);
    }

    emit(message: OutgoingMessage) {
        this.ws.send(JSON.stringify(message));
    }

    private addListeners() {
        this.ws.on("message", (message: string) => {
            const parsedMessage: IncomingMessage = JSON.parse(message);
            if (parsedMessage.method === SUBSCRIBE) {
                parsedMessage.params.forEach(s => SubscriptionManager.getInstance().
    subscribe(this.id, s));
            }

            if (parsedMessage.method === UNSUBSCRIBE) {
                parsedMessage.params.forEach(s => SubscriptionManager.getInstance().
    unsubscribe(this.id, parsedMessage.params[0]));
            }
        });
    }
}
```

Listing 4: User Handling WebSocket Communication

The `User.ts` module defines the structure and behavior of a connected WebSocket client. Each instance of the `User` class represents an individual user connection to the exchange's real-time data stream. It provides mechanisms for subscription management and message delivery, enabling clients to receive continuous updates (e.g., trades, depth, ticker) for specific markets.

Purpose

This module serves as the abstraction layer for a WebSocket-connected user, with responsibilities including:

- Managing real-time data subscriptions.
- Listening for client-issued subscription/unsubscription requests.
- Emitting outgoing messages (e.g., trade data, depth updates).

Key Components

- **WebSocket Instance** (`ws`): Represents the client's WebSocket connection.
- **User ID** (`id`): Uniquely identifies each user within the system.
- **Subscription List** (`subscriptions`): Tracks active market subscriptions for this user.

Core Functionalities

- `subscribe(subscription)`:
  - Adds a subscription topic (e.g., `depth@TATA_INR`) to the user's internal list.
  - Registers the user to that topic through the `SubscriptionManager`.
- `unsubscribe(subscription)`:
  - Removes the topic from the user's active subscription list.
  - Deregisters the user from the topic using the `SubscriptionManager`.
- `emit(message)`:
  - Serializes an outgoing message object and sends it over the user's WebSocket connection.
- `addListeners()`:
  - Attaches a listener for incoming messages.
  - Parses and processes incoming messages as `SUBSCRIBE` or `UNSUBSCRIBE` commands.

Message Flow

- Clients send JSON messages with a `method` field, such as `"SUBSCRIBE"` or `"UNSUBSCRIBE"`.
- The user instance parses these and updates its subscription state accordingly.
- On data updates (e.g., a new trade or price movement), messages are sent using `emit()`.

System Integration

- The `User` class works in coordination with the `SubscriptionManager`, which maintains mappings between users and channels.
- This integration enables efficient broadcasting of updates only to users who have explicitly subscribed to relevant topics.

The `User.ts` module encapsulates real-time interaction logic between the exchange and each connected WebSocket client. It enables a scalable, dynamic subscription model essential for delivering responsive and bandwidth-efficient market data streams.

4.2.5 User Manager (WebSockets)

```
import { WebSocket } from "ws";
import { User } from "./User";
import { SubscriptionManager } from "./SubscriptionManager";

export class UserManager {
    private static instance: UserManager;
    private users: Map<string, User> = new Map();

    private constructor() {

    }

    public static getInstance() {
        if (!this.instance)  {
            this.instance = new UserManager();
        }
        return this.instance;
    }

    public addUser(ws: WebSocket) {
        const id = this.getRandomId();
        const user = new User(id, ws);
        this.users.set(id, user);
        this.registerOnClose(ws, id);
        return user;
    }

    private registerOnClose(ws: WebSocket, id: string) {
        ws.on("close", () => {
            this.users.delete(id);
            SubscriptionManager.getInstance().userLeft(id);
        });
    }

    public getUser(id: string) {
        return this.users.get(id);
    }

    private getRandomId() {
        return Math.random().toString(36).substring(2, 15) + Math.random().toString(36).
    substring(2, 15);
    }
}
```

Listing 5: UserManager for Managing WebSocket Users

The `UserManager.ts` module implements centralized management of all WebSocket-connected users in the exchange platform. It coordinates user creation, lifecycle management, and ensures a clean disconnection process to maintain system stability and resource optimization.

Purpose

The primary responsibilities of the `UserManager` class are:

- Assign unique IDs to new WebSocket connections.

- Maintain a mapping of active users.
- Handle user disconnection and subscription cleanup.

Key Components

- **Singleton Pattern**:
  - Implements a singleton design using a static `getInstance()` method to ensure a single centralized user registry throughout the server.
- **User Mapping**:
  - Maintains a `Map` of user IDs to `User` instances for efficient lookup and management.

Core Functionalities

- `addUser(ws: WebSocket)`:
  - Generates a random unique ID.
  - Creates a new `User` instance tied to the WebSocket connection.
  - Stores the user in the internal `users` map.
  - Registers an event listener to handle connection closure.
- `registerOnClose(ws, id)`:
  - Listens for the WebSocket `"close"` event.
  - Upon disconnection, the user is removed from the users map.
  - Informs the `SubscriptionManager` that the user has left, ensuring proper subscription cleanup.
- `getUser(id)`:
  - Retrieves a specific user by their unique ID from the users map.
- `getRandomId()`:
  - Generates a pseudo-random alphanumeric string to assign as a unique user identifier.

System Flow

1. A new WebSocket connection is established.
2. `UserManager.addUser()` is invoked.
3. A new `User` is created and stored.
4. On connection closure, `registerOnClose()` cleans up user data and subscription mappings.

Integration with Other Modules

- `UserManager` interacts with:
  - `User.ts`: For creating and managing individual users.
  - `SubscriptionManager.ts`: For handling subscription removal when users disconnect.

The `UserManager.ts` module is essential for the scalability and reliability of the exchange's WebSocket layer. It ensures organized user tracking, seamless session management, and clean resource deallocation, thus supporting efficient and stable real-time data distribution.

4.2.6 User Subscription (WebSockets)

```
import { RedisClientType, createClient } from "redis";
import { UserManager } from "./UserManager";

export class SubscriptionManager {
    private static instance: SubscriptionManager;
    private subscriptions: Map<string, string[]> = new Map();
    private reverseSubscriptions: Map<string, string[]> = new Map();
    private redisClient: RedisClientType;

    private constructor() {
        this.redisClient = createClient();
        this.redisClient.connect();
    }

    public static getInstance() {
        if (!this.instance)  {
            this.instance = new SubscriptionManager();
        }
        return this.instance;
    }

    public subscribe(userId: string, subscription: string) {
        if (this.subscriptions.get(userId)?.includes(subscription)) {
            return
        }

        this.subscriptions.set(userId, (this.subscriptions.get(userId) || []).concat(
    subscription));
        this.reverseSubscriptions.set(subscription, (this.reverseSubscriptions.get(
    subscription) || []).concat(userId));
        if (this.reverseSubscriptions.get(subscription)?.length === 1) {
            this.redisClient.subscribe(subscription, this.redisCallbackHandler);
        }
    }

    private redisCallbackHandler = (message: string, channel: string) => {
        const parsedMessage = JSON.parse(message);
        this.reverseSubscriptions.get(channel)?.forEach(s => UserManager.getInstance().
    getUser(s)?.emit(parsedMessage));
    }

    public unsubscribe(userId: string, subscription: string) {
        const subscriptions = this.subscriptions.get(userId);
        if (subscriptions) {
            this.subscriptions.set(userId, subscriptions.filter(s => s !== subscription));
        }
        const reverseSubscriptions = this.reverseSubscriptions.get(subscription);
        if (reverseSubscriptions) {
            this.reverseSubscriptions.set(subscription, reverseSubscriptions.filter(s => s
    !== userId));
            if (this.reverseSubscriptions.get(subscription)?.length === 0) {
                this.reverseSubscriptions.delete(subscription);
                this.redisClient.unsubscribe(subscription);
            }
        }
    }

    public userLeft(userId: string) {
        console.log("user left " + userId);
        this.subscriptions.get(userId)?.forEach(s => this.unsubscribe(userId, s));
    }

    getSubscriptions(userId: string) {
        return this.subscriptions.get(userId) || [];
    }
}
```

Listing 6: SubscriptionManager for Managing Redis-Based Subscriptions

The `SubscriptionManager.ts` module provides the core logic for managing real-time WebSocket subscriptions. It ensures efficient broadcasting of market updates only to users who have subscribed to specific data streams such as trades or depth updates.

Purpose

The `SubscriptionManager` class is responsible for:

- Managing user subscriptions to specific market channels.
- Managing reverse mappings from channels to users.
- Subscribing and unsubscribing from Redis pub/sub channels as needed.
- Routing published messages from Redis to appropriate connected users.

Key Components

- **subscriptions**:
  - A `Map` that stores a list of subscribed channels for each user ID.
- **reverseSubscriptions**:
  - A `Map` that stores a list of user IDs for each subscription topic.
- **Redis Client**:
  - A Redis client instance to subscribe to and listen for published messages.

Core Functionalities

- `subscribe(userId, subscription)`:
  - Registers a user to a specific channel.
  - If the channel has no prior subscribers, the server itself subscribes to the Redis channel.
- `unsubscribe(userId, subscription)`:
  - Removes the user's subscription.
  - If no users are left subscribed to a channel, the server unsubscribes from the Redis channel to optimize resource usage.
- `redisCallbackHandler(message, channel)`:
  - Handles incoming Redis-published messages.
  - Forwards the parsed message to all users subscribed to the corresponding channel using the `UserManager`.
- `userLeft(userId)`:
  - Called when a user disconnects.
  - Automatically unsubscribes the user from all active subscriptions to maintain a clean state.
- `getSubscriptions(userId)`:
  - Returns all active subscriptions for a given user.

System Flow

1. A client subscribes to a market topic (e.g., `depth@TATA_INR`).
2. `SubscriptionManager.subscribe()` adds the user to the internal maps and subscribes to the Redis channel if necessary.
3. When the Redis server publishes a new message on that channel:
   - The `redisCallbackHandler` parses it.
   - The message is forwarded to all users subscribed to that channel.
4. Upon client disconnection, `userLeft()` ensures proper cleanup.

Integration with Other Modules

- Works alongside `UserManager` to retrieve and emit messages to connected users.
- Relies on Redis pub/sub mechanism for efficient real-time communication between the backend and users.

The `SubscriptionManager.ts` module enables a scalable and dynamic subscription mechanism for real-time data streams. By efficiently managing user-channel relationships and leveraging Redis pub/sub, it ensures low-latency, event-driven data delivery to WebSocket clients, crucial for high-frequency trading applications.

4.2.7 Depth.ts (API Layer Routes)

```
import { Router } from "express";
import { RedisManager } from "../RedisManager";
import { GET_DEPTH } from "../types";

export const depthRouter = Router();

depthRouter.get("/", async (req, res) => {
    const { symbol } = req.query;
    const response = await RedisManager.getInstance().sendAndAwait({
        type: GET_DEPTH,
        data: {
            market: symbol as string
        }
    });

    res.json(response.payload);
});
```

Listing 7: Depth Route for Handling Order Book Depth Requests

4.2.8 Kline.ts (API Layer Routes)

```
import { Client } from 'pg';
import { Router } from "express";
import { RedisManager } from "../RedisManager";

const pgClient = new Client({
    user: 'your_user',
    host: 'localhost',
    database: 'my_database',
    password: 'your_password',
    port: 5432,
});
pgClient.connect();

export const klineRouter = Router();

klineRouter.get("/", async (req, res) => {
    const { market, interval, startTime, endTime } = req.query;

    let query;
    switch (interval) {
        case '1m':
            query = SELECT * FROM klines_1m WHERE bucket >= $1 AND bucket <= $2;
            break;
        case '1h':
            query = SELECT * FROM klines_1m WHERE  bucket >= $1 AND bucket <= $2;
            break;
        case '1w':
            query = SELECT * FROM klines_1w WHERE bucket >= $1 AND bucket <= $2;
            break;
        default:
            return res.status(400).send('Invalid interval');
    }

    try {
        //@ts-ignore
        const result = await pgClient.query(query, [new Date(startTime * 1000 as string),
    new Date(endTime * 1000 as string)]);
        res.json(result.rows.map(x => ({
            close: x.close,
            end: x.bucket,
            high: x.high,
            low: x.low,
            open: x.open,
            quoteVolume: x.quoteVolume,
            start: x.start,
            trades: x.trades,
            volume: x.volume,
        })));
    } catch (err) {
        console.log(err);
        res.status(500).send(err);
    }
});
```

Listing 8: Kline Route for Candlestick (OHLCV) Data

4.2.9 Order.ts (API Layer Routes)

```
import { Router } from "express";
import { RedisManager } from "../RedisManager";
import { CREATE_ORDER, CANCEL_ORDER, ON_RAMP, GET_OPEN_ORDERS } from "../types";

export const orderRouter = Router();

orderRouter.post("/", async (req, res) => {
    const { market, price, quantity, side, userId } = req.body;
    console.log({ market, price, quantity, side, userId })
    const response = await RedisManager.getInstance().sendAndAwait({
        type: CREATE_ORDER,
        data: {
            market,
            price,
            quantity,
            side,
            userId
        }
    });
    res.json(response.payload);
});

orderRouter.delete("/", async (req, res) => {
    const { orderId, market } = req.body;
    const response = await RedisManager.getInstance().sendAndAwait({
        type: CANCEL_ORDER,
        data: {
            orderId,
            market
        }
    });
    res.json(response.payload);
});

orderRouter.get("/open", async (req, res) => {
    const response = await RedisManager.getInstance().sendAndAwait({
        type: GET_OPEN_ORDERS,
        data: {
            userId: req.query.userId as string,
            market: req.query.market as string
        }
    });
    res.json(response.payload);
});
```

Listing 9: Order Route for Order Creation, Cancellation, and Open Orders

4.2.10 Ticker.ts (API Layer Routes)

```
import { Router } from "express";

export const tickersRouter = Router();

tickersRouter.get("/", async (req, res) => {
    res.json({});
});
```

Listing 10: Ticker Route Placeholder (Empty JSON)

4.2.11 Trade.ts (API Layer Routes)

```
import { Router } from "express";

export const tradesRouter = Router();

tradesRouter.get("/", async (req, res) => {
    const { market } = req.query;
    // get from DB
    res.json({});
})
```

Listing 11: Trade Route Placeholder for Future Trade Data Retrieval

The backend API layer of the exchange is implemented using the Express.js framework and is organized into multiple route modules, each responsible for exposing endpoints related to specific functionalities like order management, market depth, historical candlestick data (Klines), trade history, and ticker information.

1. Depth Route (Depth.ts)

The `depthRouter` handles requests for the current market depth (orderbook bids and asks).

- **Method:** GET
- **Endpoint:** `/`

- **Query Parameter:** `symbol` (e.g., `TATA_INR`)
- **Workflow:**
  - Sends a `GET_DEPTH` message to the Redis-backed engine.
  - Awaits and responds with the orderbook depth data.

2. Kline Route (Kline.ts)

The `klineRouter` serves historical candlestick (OHLCV) data for a market using PostgreSQL.

- **Method:** GET
- **Query Parameters:** `market`, `interval` (1m, 1h, 1w), `startTime`, `endTime`
- **Workflow:**
  - Constructs an SQL query based on the interval.
  - Fetches time-bucketed Kline data from PostgreSQL.
  - Transforms and returns the result in JSON format.
- **Note:** Currently, both `1m` and `1h` intervals use the same table (`klines_1m`), which can be improved for data efficiency.

3. Order Route (Order.ts)

The `orderRouter` manages order lifecycle operations such as placement, cancellation, and retrieval.

- **Endpoints:**
  - `POST /`: Creates a new order by sending a `CREATE_ORDER` message.
  - `DELETE /`: Cancels an order via the `CANCEL_ORDER` message.
  - `GET /open`: Retrieves all open orders for a given user and market using `GET_OPEN_ORDERS`.
- **Input Parameters:**
  - `market, price, quantity, side, userId` for order creation.
  - `orderId, market` for cancellation.
  - `userId, market` for fetching open orders.

4. Ticker Route (Ticker.ts)

The `tickersRouter` is currently a placeholder route.

- **Method:** GET
- **Endpoint:** `/`
- **Response:** Empty JSON object ({}), pending implementation of real-time ticker aggregation.

5. Trade Route (Trade.ts)

The `tradesRouter` is also a placeholder for serving recent trade history.

- **Method:** GET
- **Endpoint:** /
- **Query Parameter:** `market`
- **Response:** Empty JSON object ({}), pending future integration with trade storage.

These modular Express.js route handlers provide a clear and scalable interface for external clients to interact with the exchange. While core functionalities such as order placement and depth querying are fully implemented and integrated with Redis messaging, modules like `tickersRouter` and `tradesRouter` remain as stubs and provide room for future enhancements such as real-time ticker feeds and historical trade data delivery.

4.2.12 RedisManager.ts (API Layer Routes)

```
import { RedisClientType, createClient } from "redis";
import { MessageFromOrderbook } from "./types";
import { MessageToEngine } from "./types/to";

export class RedisManager {
    private client: RedisClientType;
    private publisher: RedisClientType;
    private static instance: RedisManager;

    private constructor() {
        this.client = createClient();
        this.client.connect();
        this.publisher = createClient();
        this.publisher.connect();
    }

    public static getInstance() {
        if (!this.instance)  {
            this.instance = new RedisManager();
        }
        return this.instance;
    }

    public sendAndAwait(message: MessageToEngine) {
        return new Promise<MessageFromOrderbook>((resolve) => {
            const id = this.getRandomClientId();
            this.client.subscribe(id, (message) => {
                this.client.unsubscribe(id);
                resolve(JSON.parse(message));
            });
            this.publisher.lPush("messages", JSON.stringify({ clientId: id, message }));
        });
    }

    public getRandomClientId() {
        return Math.random().toString(36).substring(2, 15) + Math.random().toString(36).
    substring(2, 15);
    }
}
```

Listing 12: RedisManager for Request-Response Pattern via Redis Pub/Sub

This variant of the `RedisManager.ts` module is specifically used within the API layer to facilitate asynchronous communication with the matching engine over Redis. It enables the API to send messages to the engine and await responses through temporary Redis channels, effectively simulating a request-response mechanism on top of Redis pub/sub and list push.

Purpose

The key responsibilities of this module are:

- Sending structured messages to the matching engine via a Redis list queue.
- Creating temporary Redis channels to await responses for each client request.
- Ensuring that message-response correlation is preserved through unique client IDs.

Architecture Overview

- **Message Queue:** Messages are sent to the engine by pushing them to a Redis list named `"messages"`.
- **Pub/Sub Channel:** Each request generates a unique temporary Redis channel identified by a random client ID.
- **Engine Response:** The engine processes the message and publishes a response to the respective client ID channel.

Core Functionalities

- `sendAndAwait(message)`:
  - Generates a unique `clientId`.
  - Subscribes to a temporary Redis channel using this ID.
  - Pushes the original message to the Redis list `"messages"` with the client ID included.
  - Resolves the promise when a response is received and then unsubscribes from the channel.
- `getRandomClientId()`:
  - Generates a pseudo-random alphanumeric string used as a unique channel name for the current request.

Example Workflow

1. A client sends a request via an API endpoint (e.g., to place an order or get market depth).
2. The API constructs a `MessageToEngine` and calls `sendAndAwait()`.
3. The message is pushed into the Redis list `"messages"` along with a unique client ID.

4. The engine reads from this list, processes the request, and publishes the result to the `clientId` channel.
5. The API receives the message on the subscribed channel, resolves the promise, and sends a response back to the client.

This implementation of `RedisManager.ts` is essential for enabling real-time request-response communication between the API and the exchange engine. By leveraging Redis pub/sub and message queuing patterns, it allows the API to function asynchronously while maintaining performance and scalability.

4.2.13 cron.ts (Database Operations)

```
import { Client } from 'pg';

const client = new Client({
    user: 'Username',
    host: 'localhost',
    database: 'dbname',
    password: 'mypass',
    port: 5432,
});
client.connect();

async function refreshViews() {
    await client.query('REFRESH MATERIALIZED VIEW klines_1m');
    await client.query('REFRESH MATERIALIZED VIEW klines_1h');
    await client.query('REFRESH MATERIALIZED VIEW klines_1w');
    console.log("Materialized views refreshed successfully");
}

refreshViews().catch(console.error);

setInterval(() => {
    refreshViews()
}, 1000 * 10 );
```

Listing 13: cron.ts: Refreshing Materialized Views

4.2.14 seed-db.ts (Database Operations)

```
const { Client } = require('pg');

const client = new Client({
    user: 'Username',
    host: 'localhost',
    database: 'dbname',
    password: 'mypass',
    port: 5432,
});

async function initializeDB() {
    await client.connect();

    await client.query(`
        DROP TABLE IF EXISTS "tata_prices";
        CREATE TABLE "tata_prices"(
            time TIMESTAMP WITH TIME ZONE NOT NULL,
            price DOUBLE PRECISION,
            volume DOUBLE PRECISION,
            currency_code VARCHAR (10)
        );
        SELECT create_hypertable('tata_prices', 'time', 'price', 2);
    `);

    await client.query(`
        CREATE MATERIALIZED VIEW IF NOT EXISTS klines_1m AS
        SELECT
            time_bucket('1 minute', time) AS bucket,
            first(price, time) AS open,
            max(price) AS high,
            min(price) AS low,
            last(price, time) AS close,
            sum(volume) AS volume,
            currency_code
        FROM tata_prices
        GROUP BY bucket, currency_code;
    `);

    await client.query(`
        CREATE MATERIALIZED VIEW IF NOT EXISTS klines_1h AS
        SELECT
            time_bucket('1 hour', time) AS bucket,
            first(price, time) AS open,
            max(price) AS high,
            min(price) AS low,
            last(price, time) AS close,
            sum(volume) AS volume,
            currency_code
        FROM tata_prices
        GROUP BY bucket, currency_code;
    `);

    await client.query(`
        CREATE MATERIALIZED VIEW IF NOT EXISTS klines_1w AS
        SELECT
            time_bucket('1 week', time) AS bucket,
            first(price, time) AS open,
            max(price) AS high,
            min(price) AS low,
            last(price, time) AS close,
            sum(volume) AS volume,
            currency_code
        FROM tata_prices
        GROUP BY bucket, currency_code;
    `);

    await client.end();
    console.log("Database initialized successfully");
}

initializeDB().catch(console.error);
```

Listing 14: seed-db.ts: Initialize TimescaleDB and Materialized Views

4.2.15 index.ts (Database Operations)

```
import { Client } from 'pg';
import { createClient } from 'redis';
import { DbMessage } from './types';

const pgClient = new Client({
    user: 'Username',
    host: 'localhost',
    database: 'dbname',
    password: 'mypass',
    port: 5432,
});
pgClient.connect();

async function main() {
    const redisClient = createClient();
    await redisClient.connect();
    console.log("connected to redis");

    while (true) {
        const response = await redisClient.rPop("db_processor" as string);
        if (!response) {
            // No data to process
        } else {
            const data: DbMessage = JSON.parse(response);
            if (data.type === "TRADE_ADDED") {
                console.log("adding data");
                console.log(data);
                const price = data.data.price;
                const timestamp = new Date(data.data.timestamp);
                const query = 'INSERT INTO tata_prices (time, price) VALUES ($1, $2)';
                // TODO: How to add volume?
                const values = [timestamp, price];
                await pgClient.query(query, values);
            }
        }
    }
}
```

Listing 15: index.ts: Insert Trade Data from Redis into PostgreSQL

Above are the backend database components used in the exchange platform. The system uses **PostgreSQL** along with **TimescaleDB** for efficient time-series data storage and querying, and **Redis** for asynchronous data processing.

Important Note:
All database credentials, including username, password, and database name, are managed securely via environment variables defined in the `.env` file. These values are not hardcoded in production code.

a. Database Seeder (seed-db.ts)

The `seed-db.ts` script initializes the PostgreSQL database with the following:

- **Table Creation:** A table named `tata_prices` is created to store real-time price data of the TATA asset, with columns for timestamp, price, volume, and currency code.

- **Hypertable Conversion:** Using TimescaleDB's `create_hypertable()`, the table is optimized for time-series operations.
- **Materialized Views:** Three materialized views (`klines_1m`, `klines_1h`, `klines_1w`) are created for storing candlestick (OHLCV) data at 1-minute, 1-hour, and 1-week intervals respectively.

This setup enables efficient historical data analysis for generating candlestick charts and performing technical analysis.

b. Materialized View Refresher (cron.ts)

The `cron.ts` script periodically refreshes the materialized views to keep candlestick data up to date.

- Connects to the PostgreSQL database.
- Refreshes each of the 3 materialized views every 10 seconds using `REFRESH MATERIALIZED VIEW`.
- This ensures that Kline data remains current as new price entries are inserted.

c. Trade Data Listener (index.ts)

The `index.ts` script acts as a background worker that continuously pulls trade messages from a Redis queue and writes them into the database.

- Connects to Redis and PostgreSQL.
- Listens to the Redis list `db_processor` using `rPop()`.
- On receiving a `TRADE_ADDED` message:
  - Extracts trade timestamp and price.
  - Inserts the data into the `tata_prices` table.
- This architecture decouples real-time trading from database write operations, enhancing scalability and fault tolerance.

System Workflow

1. The matching engine publishes trade messages to the Redis list `db_processor`.
2. The `index.ts` listener reads these messages and writes price data into the database.
3. The `cron.ts` script updates the materialized views every 10 seconds.
4. The API layer queries these views to serve Kline (OHLCV) data to the frontend.

The database layer leverages PostgreSQL with TimescaleDB to efficiently store and query historical trading data. By using Redis for asynchronous processing and materialized views for aggregation, the system ensures that both real-time responsiveness and historical analytics are well supported.

4.3 Results and Outputs

4.3.1 Backend Architecture

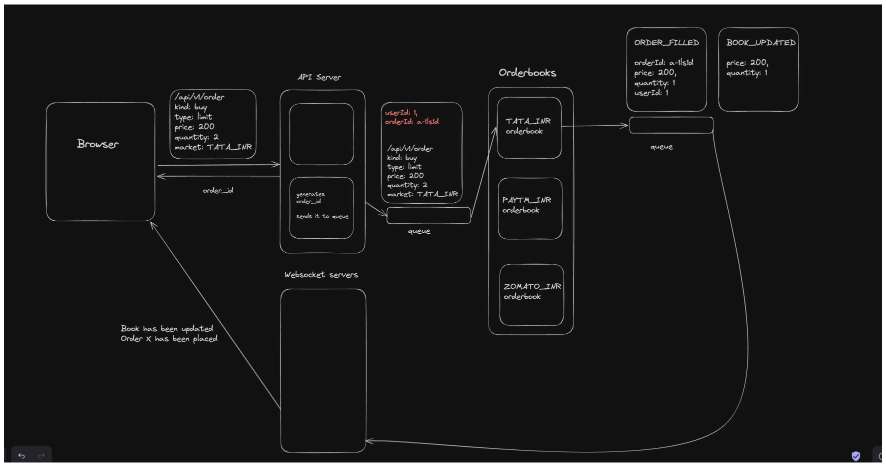


Figure 9: Backend Architecture

4.3.2 System Design

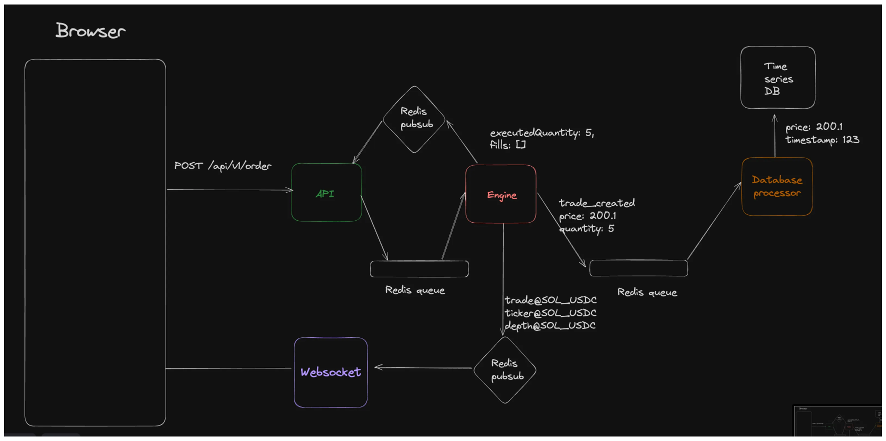
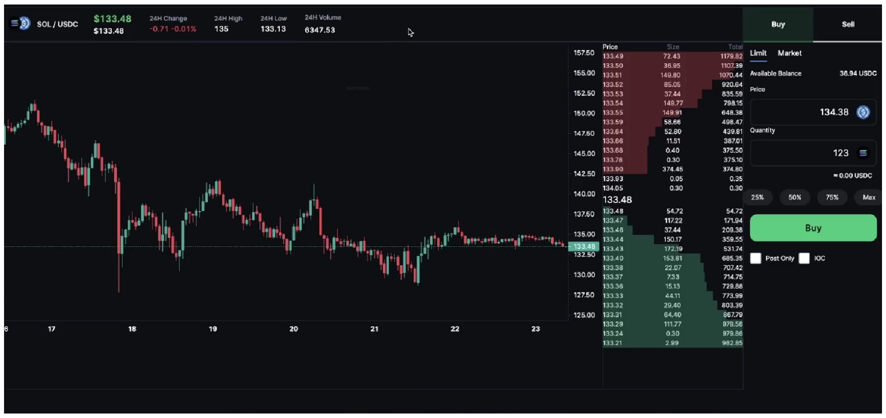


Figure 10: System Design

4.3.3 User Interface


SOL / USDC

$133.48
$133.48

24H Change
-0.71 -0.01%

24H High
135

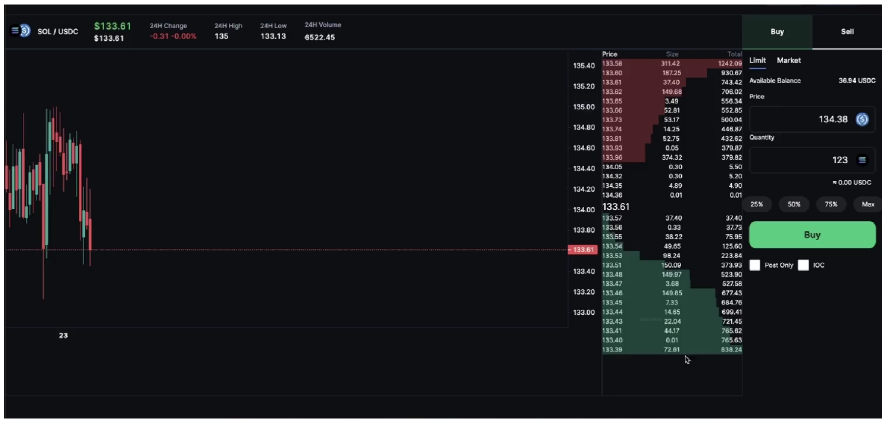
24H Low
133.13

24H Volume
6347.53

Price	Size	Total
133.49	72.43	1179.82
133.50	36.95	1107.39
133.51	149.80	1070.44
133.52	85.05	920.64
133.53	37.44	835.59
133.54	149.77	798.15
133.55	149.91	648.38
133.59	58.66	498.47
133.64	52.80	439.81
133.66	11.51	387.01
133.68	0.40	375.50
133.78	0.30	375.10
133.90	374.45	374.80
133.93	0.05	0.35
134.05	0.30	0.30
133.48		
133.48	54.72	54.72
133.47	117.22	171.94
133.46	37.44	209.38
133.44	150.17	359.55
133.43	172.19	531.74
133.40	153.61	685.35
133.38	22.07	707.42
133.37	7.33	714.75
133.36	15.13	729.88
133.33	44.11	773.99
133.32	29.40	803.39
133.31	64.40	867.79
133.29	111.77	979.56
133.24	0.30	979.86
133.21	2.99	982.85

Buy Sell
Limit Market
Available Balance 36.94 USDC
Price 134.38
Quantity 123
≈ 0.00 USDC
25% 50% 75% Max
Buy
Post Only IOC

Figure 11: SOL/USDC Trading interface with orderbook


SOL / USDC

$133.61
$133.61

24H Change
-0.31 -0.00%

24H High
135

24H Low
133.13

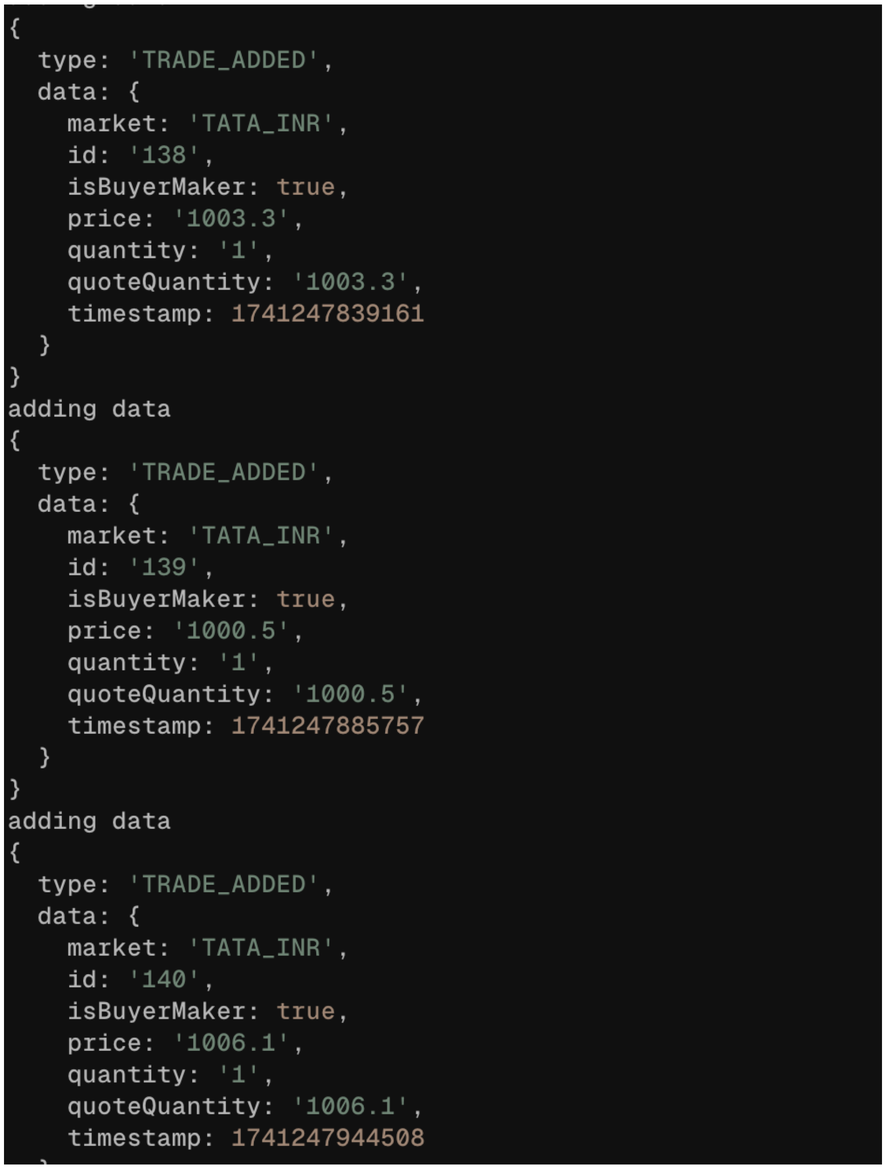
24H Volume
6522.45

Price	Size	Total
133.58	311.42	1242.09
133.60	187.25	930.67
133.61	37.40	743.42
133.62	149.68	706.02
133.65	3.49	556.34
133.66	52.81	552.85
133.73	53.17	500.04
133.74	14.25	446.87
133.81	52.75	432.62
133.93	0.05	379.87
133.96	374.32	379.82
134.05	0.30	5.50
134.32	0.30	5.20
134.35	4.89	4.90
134.36	0.01	0.01
133.61		
133.57	37.40	37.40
133.56	0.33	37.73
133.55	38.22	75.95
133.54	49.65	125.60
133.53	98.24	223.84
133.51	150.09	373.93
133.48	149.97	523.90
133.47	3.68	527.58
133.46	149.85	677.43
133.45	7.33	684.76
133.44	14.65	699.41
133.43	22.04	721.45
133.41	44.17	765.62
133.40	0.01	765.63
133.39	72.61	838.24

Buy Sell
Limit Market
Available Balance 36.94 USDC
Price 134.38
Quantity 123
≈ 0.00 USDC
25% 50% 75% Max
Buy
Post Only IOC

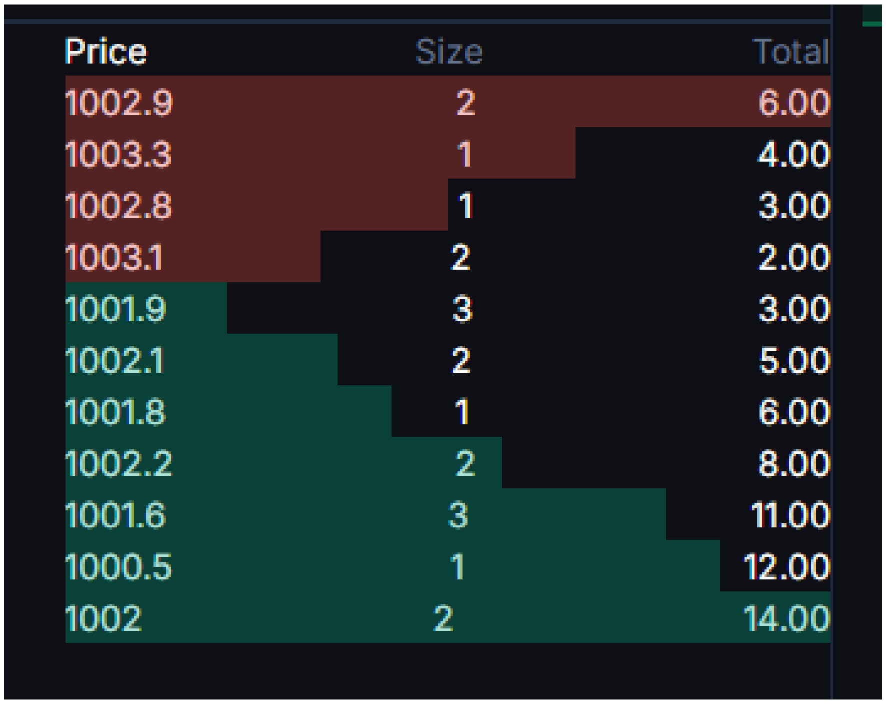
Figure 12: SOL/USDC Trading interface with orderbook

4.3.4 Order book



```
{
  type: 'TRADE_ADDED',
  data: {
    market: 'TATA_INR',
    id: '138',
    isBuyerMaker: true,
    price: '1003.3',
    quantity: '1',
    quoteQuantity: '1003.3',
    timestamp: 1741247839161
  }
}
adding data
{
  type: 'TRADE_ADDED',
  data: {
    market: 'TATA_INR',
    id: '139',
    isBuyerMaker: true,
    price: '1000.5',
    quantity: '1',
    quoteQuantity: '1000.5',
    timestamp: 1741247885757
  }
}
adding data
{
  type: 'TRADE_ADDED',
  data: {
    market: 'TATA_INR',
    id: '140',
    isBuyerMaker: true,
    price: '1006.1',
    quantity: '1',
    quoteQuantity: '1006.1',
    timestamp: 1741247944508
```

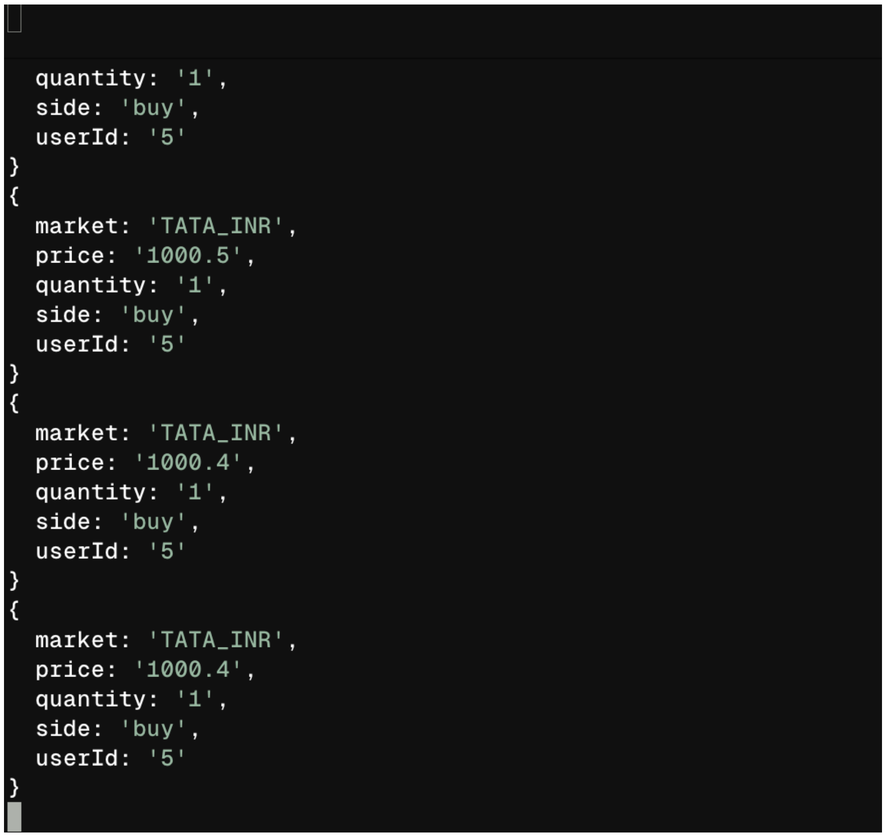
Figure 13: Orderbook



Price	Size	Total
1002.9	2	6.00
1003.3	1	4.00
1002.8	1	3.00
1003.1	2	2.00
1001.9	3	3.00
1002.1	2	5.00
1001.8	1	6.00
1002.2	2	8.00
1001.6	3	11.00
1000.5	1	12.00
1002	2	14.00

Figure 14: Order Book

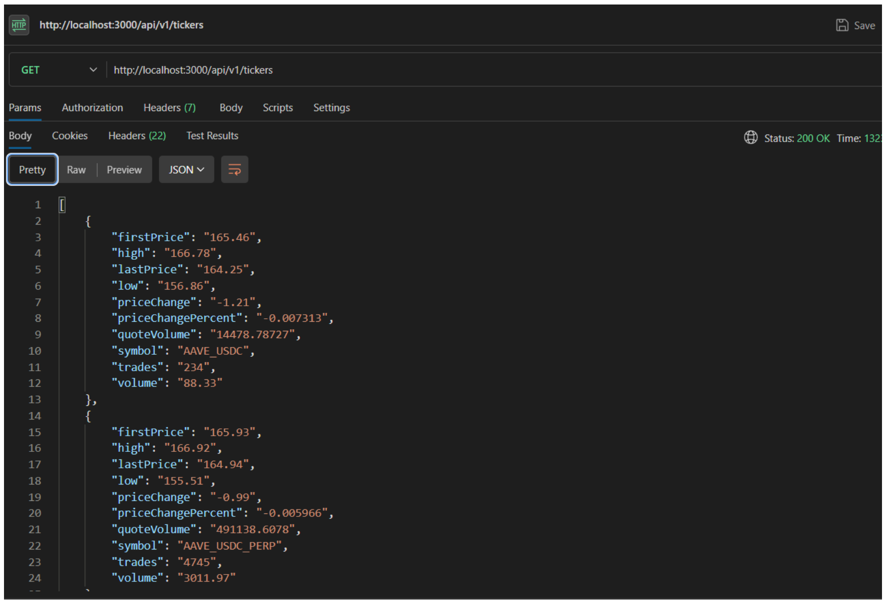
4.3.5 Order Engine



```
  quantity: '1',
  side: 'buy',
  userId: '5'
}
{
  market: 'TATA_INR',
  price: '1000.5',
  quantity: '1',
  side: 'buy',
  userId: '5'
}
{
  market: 'TATA_INR',
  price: '1000.4',
  quantity: '1',
  side: 'buy',
  userId: '5'
}
{
  market: 'TATA_INR',
  price: '1000.4',
  quantity: '1',
  side: 'buy',
  userId: '5'
}
```

Figure 15: Order Engine

4.3.6 Ticker Json in Db

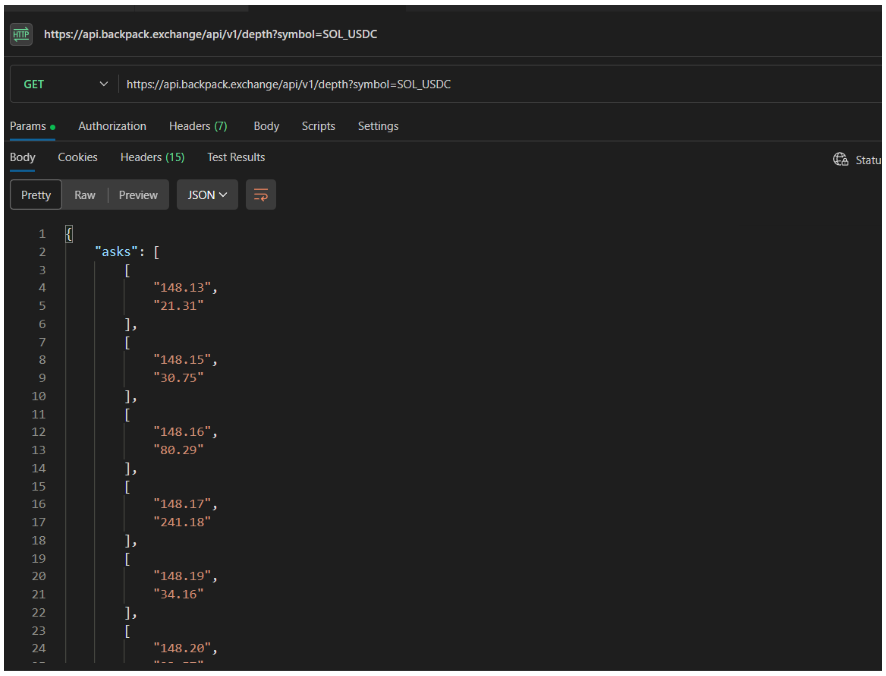


```
GET http://localhost:3000/api/v1/tickers
```

```
[
    {
        "firstPrice": "165.46",
        "high": "166.78",
        "lastPrice": "164.25",
        "low": "156.86",
        "priceChange": "-1.21",
        "priceChangePercent": "-0.007313",
        "quoteVolume": "14478.78727",
        "symbol": "AAVE_USDC",
        "trades": "234",
        "volume": "88.33"
    },
    {
        "firstPrice": "165.93",
        "high": "166.92",
        "lastPrice": "164.94",
        "low": "155.51",
        "priceChange": "-0.99",
        "priceChangePercent": "-0.005966",
        "quoteVolume": "491138.6078",
        "symbol": "AAVE_USDC_PERP",
        "trades": "4745",
        "volume": "3011.97"
```

Figure 16: Ticker data in db

4.3.7 Asks and Bids info in db



```
GET https://api.backpack.exchange/api/v1/depth?symbol=SOL_USDC
```

```
{
    "asks": [
        [
            "148.13",
            "21.31"
        ],
        [
            "148.15",
            "30.75"
        ],
        [
            "148.16",
            "80.29"
        ],
        [
            "148.17",
            "241.18"
        ],
        [
            "148.19",
            "34.16"
        ],
        [
            "148.20",
```

Figure 17: Asks and Bids info in db

5 System Requirements

The system requirements define the hardware and software specifications necessary for users to run and access the trading platform efficiently.

5.1 Hardware Requirements

a. **Processor:** Minimum Intel i3 / AMD Ryzen 3.

b. **Memory (RAM):** Minimum 4GB (Recommended: 8GB for smoother performance during high trading volume).

c. **Internet Connection:** Minimum 10 Mbps broadband (Recommended: 50 Mbps or higher for real-time updates and trading execution).

5.2 Software Requirements

5.2.1 Operating System

a. Windows 10 or later (Recommended: Windows 11 for latest security features).

b. macOS 11 (Big Sur) or later.

c. Linux (Ubuntu 20.04 LTS or later).

5.2.2 Web Browser

a. Google Chrome

b. Mozilla Firefox

c. Microsoft Edge (Chromium-based).

d. Safari

5.2.3 Additional Software

a. Latest JavaScript-enabled browser for smooth UI interactions.

b. WebSocket support enabled for real-time market updates.

6 References

References

[1] Investopedia, ”High-Frequency Trading Explained,” [Online]. Available: https://www.investopedia.com/terms/h/high-frequency-trading.asp.

[2] Devexperts, ”Ultra-Low Latency Crypto Exchange,” [Online]. Available: https://devexperts.com/blog/ultra-low-latency-crypto-exchange/.

[3] Alibaba Cloud, ”A Guide to Ultra-Low Latency Crypto Trading on the Cloud - Part 1: Infrastructure Fundamentals,” [Online]. Available: https://www.alibabacloud.com/blog/a-guide-to-ultra-low-latency-crypto-trading-on-the-cloud-part-1---infrastructure-fundamentals_601851.

[4] Binance API Documentation, ”WebSocket Market Streams,” [Online]. Available: https://developers.binance.com/docs/binance-spot-api-docs.

[5] AWS, ”Event-Driven Architecture Patterns,” [Online]. Available: https://aws.amazon.com/event-driven-architecture/.

[6] Investopedia, ”Market Depth,” [Online]. Available: https://www.investopedia.com/terms/m/marketdepth.asp.

[7] Ethereum Foundation, ”Smart Contracts and Decentralized Exchanges,” [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/.

[8] M. Fowler, ”Microservices Architecture Guide,” [Online]. Available: https://martinfowler.com/articles/microservices.html.

[9] AWS, ”Microservices on AWS,” [Online]. Available: https://aws.amazon.com/microservices/.

[10] Kubernetes Documentation, ”Container Orchestration for Scalability,” [Online]. Available: https://kubernetes.io/docs/home/. [Accessed: 10-Feb-2025].

[11] OAuth, ”Open Standard for Secure API Authentication,” [Online]. Available: https://oauth.net/2/.

[12] Microminder Cyber Security, ”Common DDoS Mitigation Strategies: A Comprehensive Guide,” [Online]. Available: https://www.micromindercs.com/blog/common-ddos-mitigation-strategies-a-comprehensive-guide.

[13] Investopedia, ”Market Makers and Their Role in Trading,” [Online]. Available: https://www.investopedia.com/terms/m/marketmaker.asp.

[14] Gemini, ”What Is a Liquidity Pool? Crypto Market Liquidity,” [Online]. Available: https://www.gemini.com/cryptopedia/what-is-a-liquidity-pool-crypto-market-liquidity.

[15] Google Cloud, ”Load Balancing Strategies for High-Traffic Applications,” [Online]. Available: https://cloud.google.com/load-balancing/.

[16] AWS Lambda, ”Serverless Computing for Trading Applications,” [Online]. Available: https://aws.amazon.com/lambda/.

[17] K. S. Vanitha, S. V. UMA and S. K. Mahidhar, ”Distributed denial of service: Attack techniques and mitigation,” 2017 International Conference on Circuits, Controls, and Communications (CCUBE), Bangalore, India, 2017, pp. 226-231, doi: 10.1109/CCUBE.2017.8394146.