Report 3
Course Number and Name: SENG 468, Software Systems Scalability

Table of Contents

1. Overview

The day trading system is a high-performance, scalable application designed to handle large-scale financial transactions with minimal latency. The platform ensures that buy and sell orders are processed efficiently while maintaining fairness and security. Given the volatile nature of stock markets, real-time trade execution is critical. Our system employs a microservices-based architecture that allows individual services to scale independently, preventing bottlenecks in trade execution. To improve response times and reduce load on the databases, Redis caching has been introduced to store authentication tokens, wallet balances, and frequently accessed data related to stock prices.

Initially, we moved from a Django-based authentication service to Flask for reduced processing overhead. Subsequently, we further enhanced our authentication service by rewriting it in Go (Golang) for even better performance and concurrency handling capabilities[6]. Alongside this change, we shifted the authentication data storage from PostgreSQL to MongoDB to streamline interactions and reduce overhead. Additionally, we use a write-through strategy for authentication data, writing to Redis and MongoDB simultaneously to maintain data consistency and high-speed lookups.

Sharding in the authentication service ensures better load distribution and higher reliability, complemented by horizontal scaling to handle large volumes of incoming requests. By replicating microservices, implementing advanced caching mechanisms, and scaling horizontally, the system can now support significantly higher traffic while ensuring fault tolerance. The matching engine has also been optimized for faster trade execution, ensuring fair price matching and market liquidity. We further split the matching engine into two separate services to handle different stocks (e.g., Google and Apple) independently using a modulo-based assignment strategy, which prevents a single matching engine from becoming a bottleneck for all trading activities.

These collective optimizations, combined with strategic resource allocation (e.g., allocating more CPU and memory to the matching engine), ensure the system operates efficiently even under high-load scenarios.

2. Functional Requirements

The system is designed to provide core trading functionalities with a strong emphasis on efficiency, reliability, and security. Users must be able to create accounts, log in, and manage their stock portfolios securely.

The system supports two types of stock orders:

- Market Buy Orders: Execute immediately at the best available price.
- Sell Limit Orders: Execute only when a matching buy order meets the seller's specified price.

Orders are processed in real-time using a FIFO algorithm, ensuring fair and efficient trade execution. Partial fulfillment is supported, allowing trades to be executed even when full quantities are unavailable. Additionally, users have access to a digital wallet where they can deposit funds, check balances, and track their stock holdings. To ensure financial accuracy, all transactions are logged and auditable.

The authentication process is reinforced using JWT-based authentication, and passwords are securely stored using bcrypt hashing. With our updated approach in Go, user authentication data is now stored in MongoDB, with a write-through to Redis for faster lookups and token management. Caching mechanisms in Redis further enhance system responsiveness by reducing redundant database queries[1], ensuring rapid login checks, and allowing quick retrieval of frequently accessed data such as wallet balances and stock prices.

3. Non-Functional Requirements

3.1 Performance & Scalability

Performance and scalability are key concerns for any high-frequency trading platform. Our system is designed to support thousands of concurrent users without performance degradation. Microservices are containerized using Docker, ensuring consistent deployment and easy scaling.

The introduction of Redis caching reduces database load by handling frequently accessed data in-memory [1]. By migrating the authentication service to Go and using MongoDB, we have improved concurrency and reduced overhead, making the system even more responsive under heavy loads. The authentication service has also been sharded, distributing login requests across multiple instances to balance the load and prevent slowdowns.

Additionally, the system employs horizontal scaling, allowing more instances of microservices to be deployed dynamically based on demand. Splitting the matching engine into two separate instances for different stocks ensures that if one matching engine faces heavy load, it does not affect the other.

3.2 Availability & Fault Tolerance

To ensure continuous availability, the system incorporates multiple failover mechanisms. Service replication is used to prevent disruptions in the event of service failures. Load balancing techniques are employed to distribute traffic evenly across different instances, preventing bottlenecks and downtime. The databases are configured with replication, ensuring that backup copies are always available in case of failure. Health checks monitor service performance, automatically restarting failing services to minimize disruptions. Redis caching ensures smooth performance even under heavy traffic by serving frequently requested data instantly.

In case of an overburdened matching engine, horizontal scaling quickly spins up additional matching engine instances. Our infrastructure also includes increased CPU and memory allocation for the matching engine services to handle intensive computational requirements associated with trade matching.

3.3 Security

Security is a critical aspect of any financial system. The system uses JWT-based authentication to secure user sessions, ensuring that only authorized users can access trading functionalities. Passwords are hashed using bcrypt, preventing data leaks in case of breaches. Rate limiting is enforced at the API Gateway level to protect against brute-force attacks. All API inputs undergo rigorous validation to guard against SQL injection and cross-site scripting (XSS) attacks. Additionally, microservices communicate securely through encrypted channels, preventing unauthorized data access.

The switch to a Go-based auth service, combined with MongoDB as its primary data store, introduces additional concurrency and performance benefits without compromising security. Our write-through strategy ensures that any updates (e.g., password changes) are immediately reflected in Redis and MongoDB, maintaining consistency while allowing high-speed lookups.

4. Hardware Specifications

The system is hosted on a high-performance cloud-based infrastructure with the following specifications:

- **Operating System:** Ubuntu 24.04
- **RAM:** 16GB
- **Storage:** 70GB SSD

- **CPU:** 8-core processor
- **Dedicated Redis and PostgreSQL instances** ensure optimal system performance.

5. Software Specifications

The system leverages a combination of databases, microservices, and modern frameworks to ensure high performance, scalability, and reliability.

5.1 Database Choices

The system uses multiple databases, each optimized for specific functionalities:

- **MongoDB –** A NoSQL database used to manage stock orders, trade transactions, portfolio records, and now user authentication data, allowing for flexible and scalable data storage. MongoDB's sharding and replication capabilities help distribute large amounts of data across multiple instances for better performance [3].
- **Redis –** A fast in-memory cache database used to store frequently accessed data such as authentication tokens, wallet balances, and stock prices, reducing the load on primary databases. We employ a write-through strategy in the auth service where data is written simultaneously to Redis and MongoDB to maintain data consistency.
- **SQLAlchemy –** A database toolkit and ORM (Object-Relational Mapper) traditionally used for Postgres connections in our older architecture. Components still relying on SQLAlchemy for structured data continue to benefit from improved data handling and query efficiency.

5.2 Microservices & Frameworks

The system is built using a microservices architecture, ensuring modular and independent scalability. Key technologies include:

- **Go (Golang) Auth Service** – Our updated authentication microservice is implemented using Go. This shift from Python/Flask to Go has provided enhanced concurrency handling and higher throughput, essential for large-scale user logins and token management.
- **Flask (Python)** – Still used for some backend services, handling API requests for certain portfolio or ancillary functions.
- **Docker** – Used for containerization to ensure consistent application deployment across different environments.
- **RabbitMQ** – A messaging broker that enables asynchronous communication between microservices using queues and the publish/subscribe model.
- **Nginx** – Functions as both an API Gateway and a Load Balancer, managing incoming traffic and distributing requests efficiently to prevent bottlenecks.
- **Database Scaling**
    - **Sharding** – Distributes large amounts of data across multiple database instances for better performance [3].
    - **Read Replicas** – Improves database read efficiency by creating multiple copies of the primary database, though this is now more relevant for MongoDB sharding and replication setups.

5.3 Programming Languages & Frameworks

The following programming languages and frameworks are used in development:

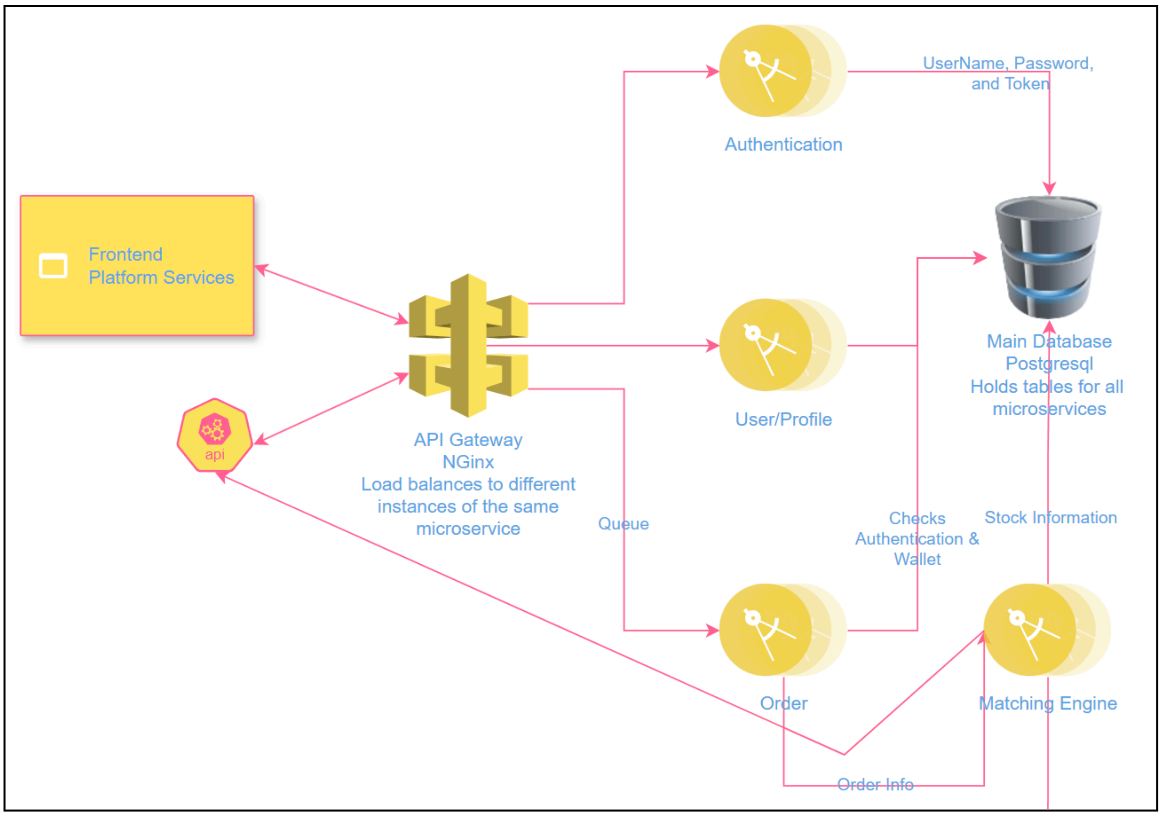
- **Go (Golang) –** Primary language for the new authentication microservice, leveraging Go’s high concurrency model, furthermore it’s lower memory consumption is very beneficial [6].
    - Efficient concurrency in Go, unlike Python, has no Global Interpreter Lock (GIL)—a mechanism in Python that prevents multiple threads from executing in parallel within a single process. This limitation hinders Python's ability to leverage multi-core processors, whereas Go excels at concurrent execution with its lightweight goroutines [6].
- **Python –** Used for other microservices, including the order and matching engine components, as well as certain backend logic.
- **Django –** Initially used for dynamic web content rendering and API development, though many of its functions have been replaced or augmented by Flask and Go services.
- **Flask –** Lightweight framework for backend API in some microservices and certain new functionalities.
- **HTML & CSS –** Frontend technologies for rendering the user interface.
- **SQL –** For any remaining relational database operations in PostgreSQL.
- **NoSQL –** Used for handling unstructured and semi-structured data in MongoDB.

5.4 Additional Technologies

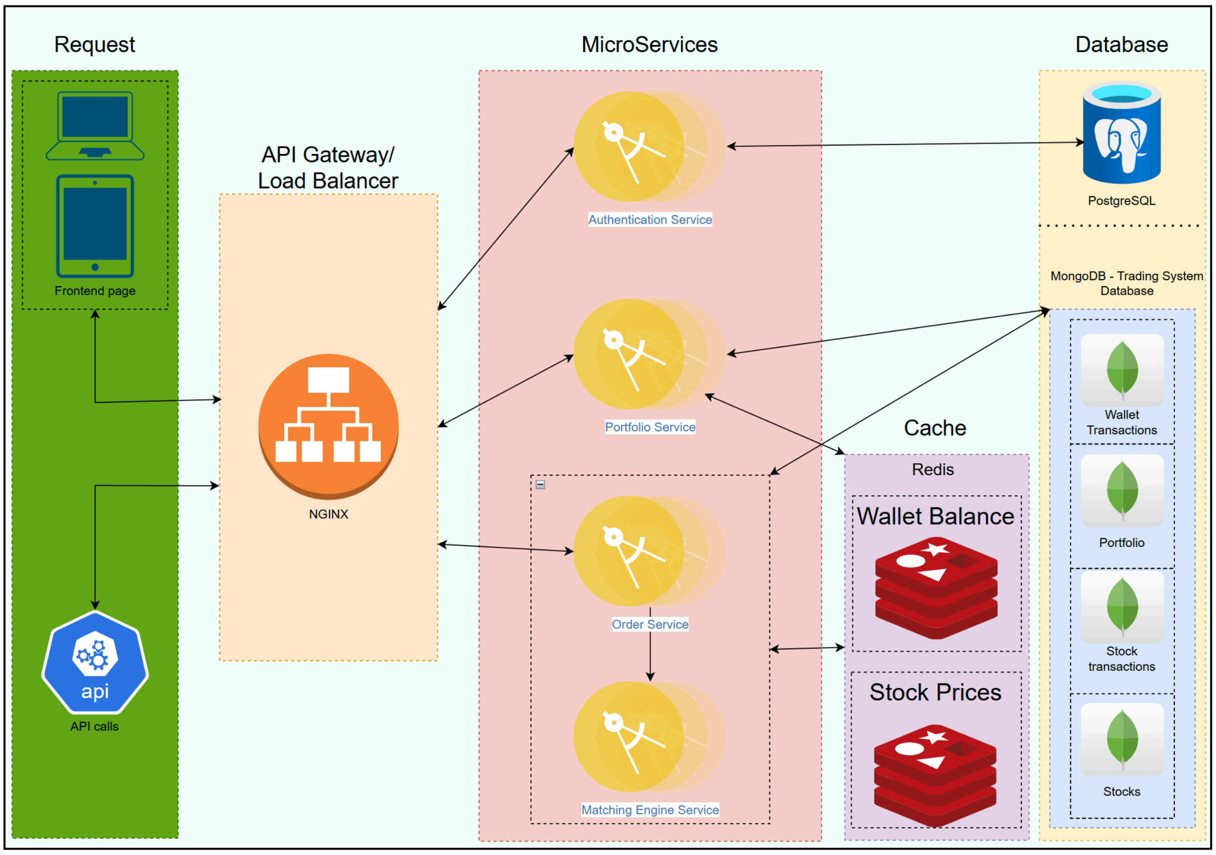
To enhance system functionality and security, the following technologies are implemented:

- **JWT (JSON Web Token) –** Used for secure authentication and session management.
- **JSON –** Standard format for API communication.
- **API Endpoints –** Well-defined RESTful APIs to interact with various system components
- **Flask-CORS –** Handles Cross-Origin Resource Sharing (CORS), allowing secure API calls from different domains.
- **Dotenv –** Manages environment variables securely for database credentials and API keys.

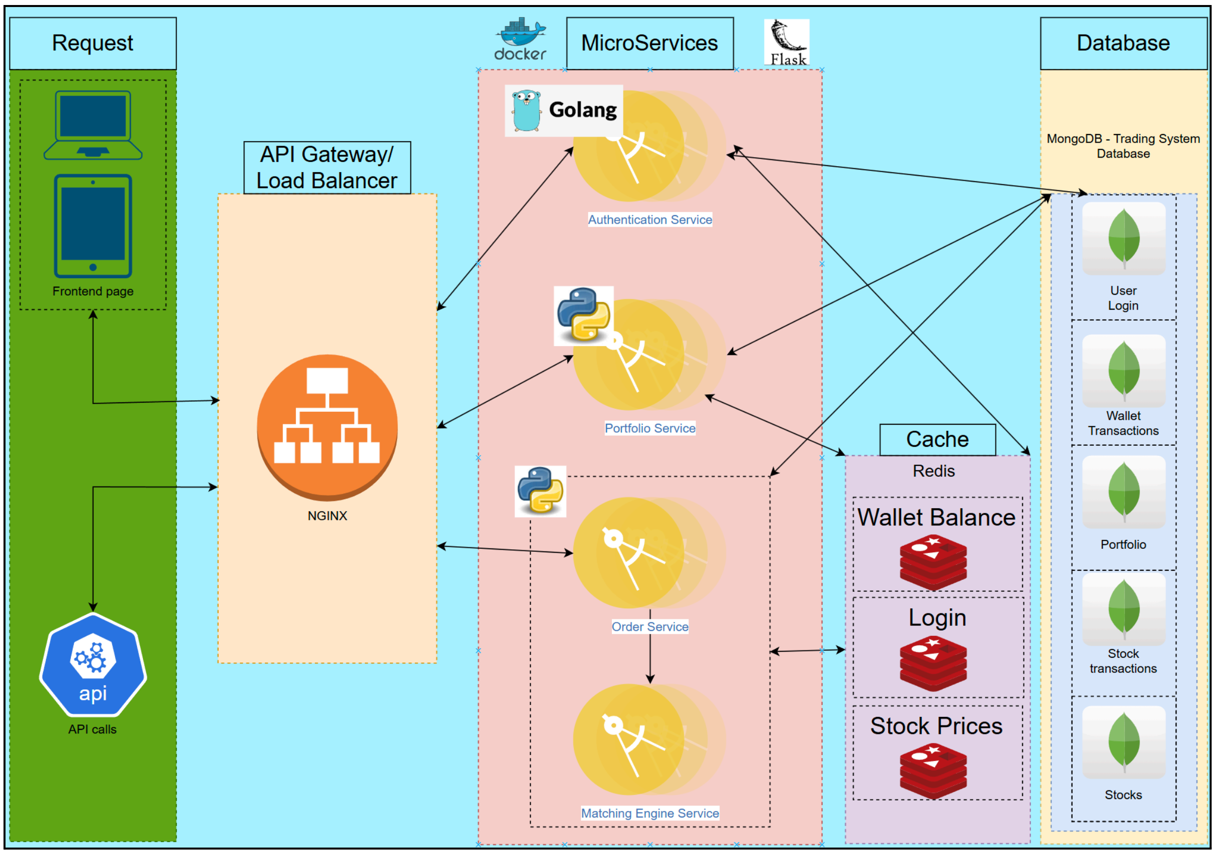
6. Architecture


Authentication
UserName, Password, and Token
Frontend Platform Services
User/Profile
Main Database Postgresql Holds tables for all microservices
api
API Gateway NGinx Load balances to different instances of the same microservice
Queue
Checks Authentication & Wallet
Stock Information
Order
Matching Engine
Order Info

- This above picture is the first implementation architecture for 1 User test



- Above Image is our 10K user architecture



- Architecture diagram for test run 3

The system follows a microservices-based architecture, designed to handle 10,000 concurrent users efficiently while ensuring modularity, independent scalability, and fault tolerance. This architecture consists of multiple key components that work together to provide real-time stock trading functionality, reduce latency, and maintain system reliability under high loads. Below is a detailed breakdown of the architecture:

6.1 API Gateway & Load Balancer (NGINX)

The API Gateway acts as the single entry point for all client requests, whether they come from the web-based frontend or direct API calls. It manages routing, authentication, and request forwarding to the appropriate microservices. This setup enhances security and performance by filtering requests before they reach the backend services.

The Load Balancer, implemented using NGINX, ensures that incoming traffic is distributed evenly among multiple instances of each microservice [2]. This prevents any single service from being overwhelmed by high request volumes, allowing the system to scale horizontally as needed.

6.2 Microservices Layer

Each core functionality is encapsulated within its own microservice, enabling modularity, independent scaling, and fault isolation. The microservices communicate asynchronously, reducing system bottlenecks.

a) Authentication Service (Go-based)

- Rewritten in Go (Golang) to reduce overhead and improve concurrency and response times.
- Uses MongoDB as the primary database for storing user credentials and session data.
- Implements sharding, meaning authentication requests are distributed across multiple database instances, preventing slowdowns under heavy load.
- Uses JWT-based authentication, ensuring secure, stateless session management.
- Employs Redis for token caching and high-speed lookups, using a write-through strategy to keep Redis and MongoDB in sync.

b) Portfolio Service

- Manages user portfolios, tracking stock holdings, and available balance.
- Uses MongoDB for flexible, scalable data storage.
- Frequently accessed data (e.g., wallet balances) is cached in Redis to speed up portfolio lookups.
- Ensures consistency between buy/sell transactions and user portfolios.

c) Order Service

- Handles user orders (buy/sell), forwarding them to the Matching Engine(s) for execution.
- Uses RabbitMQ for asynchronous processing, improving system responsiveness.
- Stores all transaction records in MongoDB, ensuring accurate order tracking and auditing.
- Implements sharding and replication, allowing orders to be processed in parallel across multiple instances [3].

d) Matching Engine Service (Split Implementation)

- Core logic for executing trades, using a FIFO-based matching algorithm to process buy and sell orders fairly.
- Now split into two separate engines (e.g., one handles Google stocks, the other handles Apple stocks), using a modulo or stock-based routing mechanism to decide which engine processes a given order. This split approach provides improved scalability and fault isolation.
- Maintains an order book, tracking pending buy and sell orders.
- Uses Redis caching to store stock prices, wallet balances, and order book snapshots, reducing database load.
- Automatically scales up during high trade volumes by spawning additional matching engine instances.
- Additional memory (RAM) and CPU resources are allocated to the matching engine services to handle the computational intensity of matching trades in real time.

6.3 Database Layer

To support high throughput and low latency, the system uses a hybrid database approach:

- **MongoDB**: Stores trading-related data such as stock transactions, portfolios, order details, and now user authentication. We also leverage sharding and replication to efficiently handle large volumes of data.
- **Redis**: Used for caching frequently accessed data, such as wallet balances, stock prices, and active orders, reducing database query times. The write-through strategy in the authentication service ensures consistent data across Redis and MongoDB.

6.4 Caching Layer (Redis)

Caching is implemented at multiple points to optimize performance:

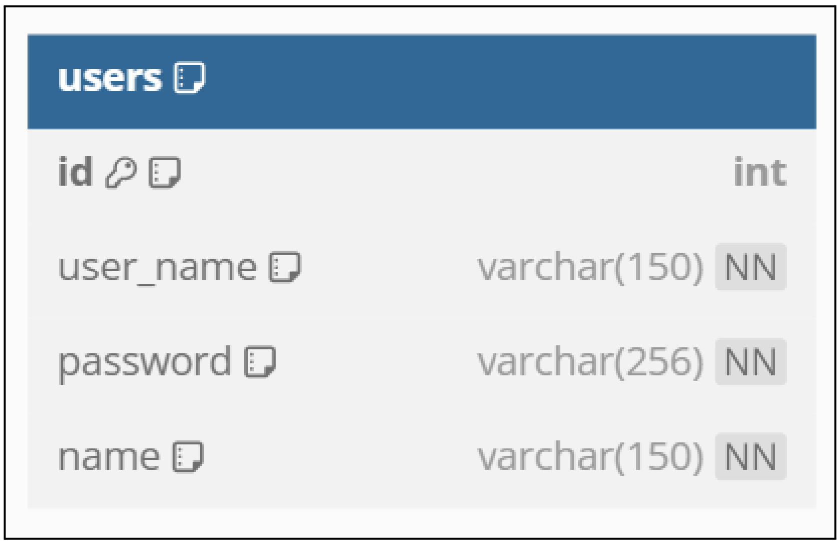
- **Wallet Balance Cache**: Ensures near-instant retrieval of user balances, avoiding frequent database queries.
- **Stock Prices Cache**: Keeps real-time price updates readily available for order execution and display in the frontend.
- **User Authentication Cache**: Reduces login time by storing session tokens in Redis, with updates written through to MongoDB.

6.5 Data Flow & Request Processing

1. A user interacts with the frontend (web app or API).
2. The request is routed through the API Gateway (NGINX), which validates and forwards it to the correct microservice.
3. If authentication is required, the Go-based Authentication Service verifies the user's credentials by checking Redis and, if needed, querying MongoDB.
4. For portfolio-related requests, the Portfolio Service retrieves cached data or fetches it from MongoDB.
5. For buy/sell orders, the Order Service validates the request and sends it to the correct Matching Engine (based on stock type or a modulo-based routing scheme).
6. The Matching Engine processes the order and updates the Portfolio Service and Wallet Balance Cache in Redis.
7. Once a trade is executed, the system logs the transaction and notifies the frontend.
8. The Load Balancer ensures even traffic distribution across service instances, preventing congestion.

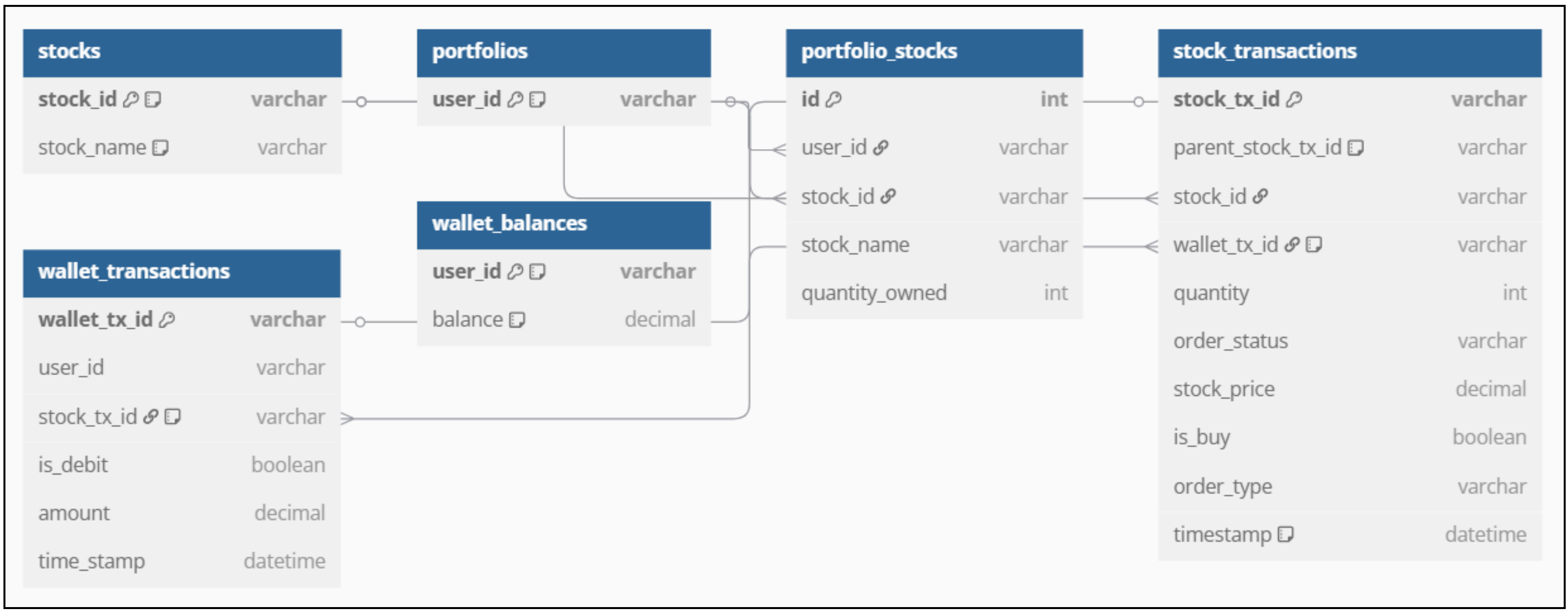
6.6 Database Schema

PostgreSQL schema (Removed from Test run 3)



users		
id	int	
user_name	varchar(150)	NN
password	varchar(256)	NN
name	varchar(150)	NN

MongoDB and Redis



stocks	
stock_id	varchar
stock_name	varchar

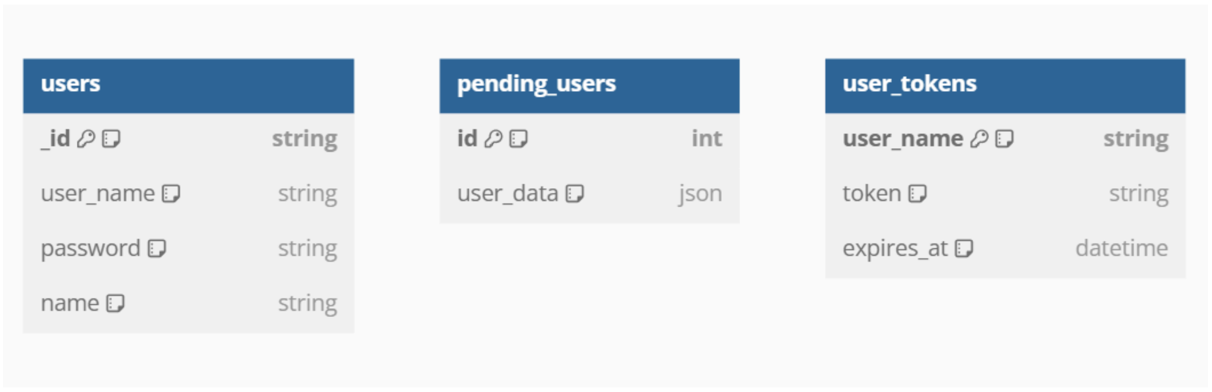
portfolios	
user_id	varchar

portfolio_stocks	
id	int
user_id	varchar
stock_id	varchar
stock_name	varchar
quantity_owned	int

stock_transactions	
stock_tx_id	varchar
parent_stock_tx_id	varchar
stock_id	varchar
wallet_tx_id	varchar
quantity	int
order_status	varchar
stock_price	decimal
is_buy	boolean
order_type	varchar
timestamp	datetime

wallet_balances	
user_id	varchar
balance	decimal

wallet_transactions	
wallet_tx_id	varchar
user_id	varchar
stock_tx_id	varchar
is_debit	boolean
amount	decimal
time_stamp	datetime

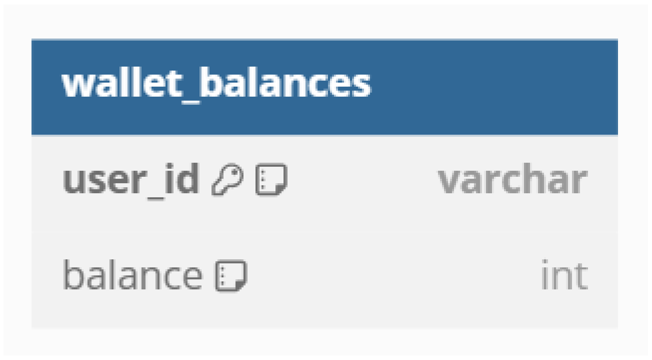


users	
_id	string
user_name	string
password	string
name	string

pending_users	
id	int
user_data	json

user_tokens	
user_name	string
token	string
expires_at	datetime

- Add wallet balance and portfolio stocks redis database schema



wallet_balances	
user_id	varchar
balance	int

MongoDB and Redis

- MongoDB collections for user authentication, orders, portfolios, and executed trades.
- Redis for caching tokens, stock prices, wallet balances, and partial order book snapshots.

7. Matching Engine

The Matching Engine is responsible for executing trades efficiently and ensuring fairness in stock transactions. It processes buy and sell orders in real time, ensuring that users get the best possible price while maintaining market integrity. The system is designed to handle high-frequency trading, minimize latency, and support partial order fulfillment.

With the introduction of a split matching engine (one engine for each major stock, e.g., Google and Apple), we ensure improved performance by isolating orders for different assets. This also allows us to quickly scale out an engine if a particular stock experiences sudden high-volume trading.

7.1 Order Matching Process

The matching engine follows a FIFO (First In, First Out) priority system:

- **Market Orders**: Executed immediately at the best available price.
- **Limit Orders**: Placed in the order book and executed only when the specified price is met.

- **Partial Orders**: If an order cannot be fully matched, a portion is executed while the rest remains in the queue.

Order Matching Algorithm:

1. A buy order is placed; if it's a market order, it matches the lowest available sell price.
2. A sell order is placed; if it's a market order, it matches the highest available buy price.
3. The order book maintains time priority (older orders execute first).
4. If there are no matching orders, the order remains open until a match is found.

7.2 Edge Cases and Handling Complex Scenarios

The system handles various edge cases to ensure smooth trading:

- **No Matching Orders**:
    - If a market buy order is placed but no sellers are available, the order remains queued.
    - If a market sell order is placed but no buyers exist, the order remains open until matched.

- **Insufficient Funds**:
    - If a buyer does not have enough balance, the system calculates how many shares can be bought and processes only that portion.

- **Large Orders That Cannot Be Fully Filled**:
    - If a buy order of 500 shares is placed but only 300 shares are available, a partial match occurs, and the remaining 200 shares stay in the queue.

- **Self-Trading Prevention**:
    - If a user accidentally tries to buy and sell their own stock, the system skips the transaction.

- **Order Cancellation**:
    - Users can cancel orders that have not been executed.
    - Once an order is partially executed, only the remaining quantity can be canceled.

7.3 Data Storage and Execution

- Orders are stored in MongoDB, ensuring reliability and persistence.
- Stock prices and wallet balances are cached in Redis for fast retrieval, reducing database load.
- Order transactions are logged for auditing and regulatory compliance.

How Orders Are Stored:

- **Buy Orders:** `{ "user_id": "uuid", "stock_id": "AAPL", "price": 150, "quantity": 10 }`

- **Sell Orders:** `{ "user_id": "uuid", "stock_id": "AAPL", "price": 155, "quantity": 5 }`
- **Executed Trades:** `{ "buyer_id": "uuid", "seller_id": "uuid", "stock_id": "AAPL", "price": 152, "quantity": 5 }`

7.4 Performance Optimizations

The matching engine uses various performance enhancements to handle high trading volumes efficiently:

- **Redis Caching**:
  Stores real-time stock prices and wallet balances to reduce database queries and improve execution speed.
- **Multithreading / Concurrency**:
  Trades are processed in parallel, ensuring multiple users can place orders at the same time.
- **Asynchronous Processing with RabbitMQ**:
  Ensures non-blocking order execution, reducing delays.
- **Database Indexing**:
  Uses MongoDB indexes for fast lookups, improving query speed.

- **Sharded Order Book**:
  Orders are distributed across multiple database instances, ensuring efficient scaling.
- **Split Matching Engine Instances**:
  By dividing matching engines for different stocks (e.g., Google vs. Apple), we reduce contention and allow each engine to scale independently.

8. Security

To scale the application for 150k users, we have reinforced security measures across authentication, data protection, attack prevention, and monitoring.

8.1 Authentication & Authorization

- **JWT-based** authentication is implemented for session management, allowing stateless and scalable authentication.
- **Bcrypt** is used for password hashing to enhance security against brute-force attacks.
- **Independent scaling** of the Go-based authentication service ensures high availability during peak traffic.

8.2 Input Validation & Attack Prevention

- Strict input validation prevents SQL injection and XSS attacks.
- Sanitization of all API inputs ensures only expected data formats are processed.

8.3 Data Protection & Encryption

- MongoDB handles structured user data (previously in PostgreSQL), applying encryption at rest where necessary.
- Redis caches authentication tokens securely, minimizing exposure risks.
- Inter-service communication is fully encrypted to prevent unauthorized access.

8.4 API Security & Rate Limiting

- Nginx serves as an API Gateway, efficiently managing traffic and enforcing security rules.
- Rate limiting is enforced at the API level to prevent brute-force attacks and excessive API calls.
- Mutual TLS (mTLS) authentication is applied between microservices, ensuring only authorized components communicate.

8.5 Caching & In-Memory Security

- Redis is used for wallet balance caching, improving performance while securing financial data.
- Short expiration times are applied to cached authentication and portfolio data to reduce risks.

8.6 Logging, Monitoring & Incident Response

- Enhanced logging of user actions, such as order placements and authentication attempts, ensures traceability.
- Automated monitoring detects anomalies and raises alerts for suspicious activities.
- Audit logs for sensitive operations (e.g., modifying stock transactions) provide an additional layer of security.

These updates strengthen the system against cyber threats while maintaining scalability, reliability, and data integrity for high-frequency trading.

9. Fault Tolerance

To ensure uninterrupted service, the system employs various fault-tolerance mechanisms. Service replication guarantees that critical components remain operational even if individual instances fail. Database replication ensures data availability, while automated health checks identify and restart failing services. Redis caching prevents system slowdowns during peak traffic, allowing for smooth operations.

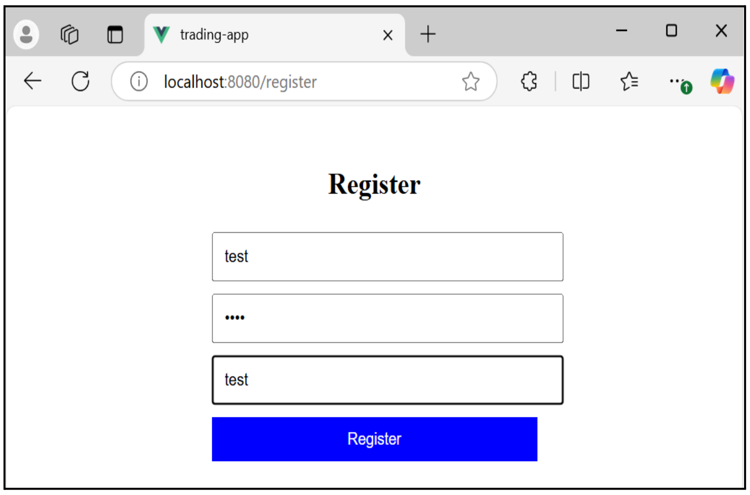
If a particular matching engine (e.g., the one handling Google stocks) experiences issues, the system can either restart that engine or route Google stock orders temporarily to a backup or secondary matching engine instance. Scaling out additional matching engine nodes or spinning up new microservice instances quickly is key to maintaining robust, fault-tolerant operation. Additionally, provisioning increased CPU and memory resources helps mitigate performance bottlenecks and reduce the risk of outages under extreme load.

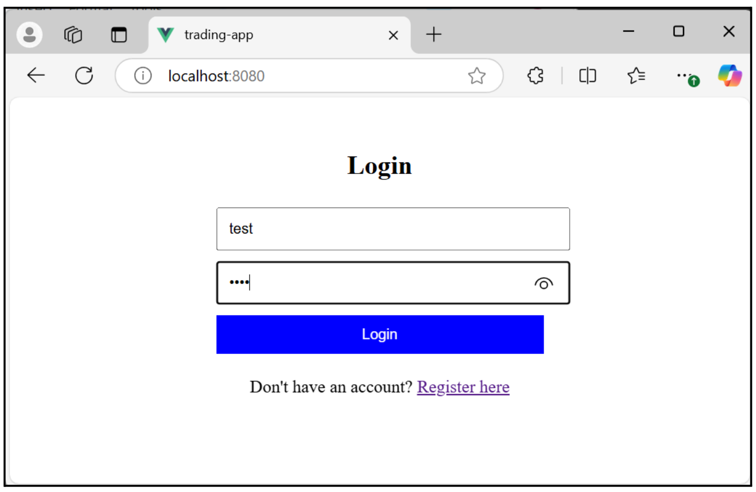
10. Database Management

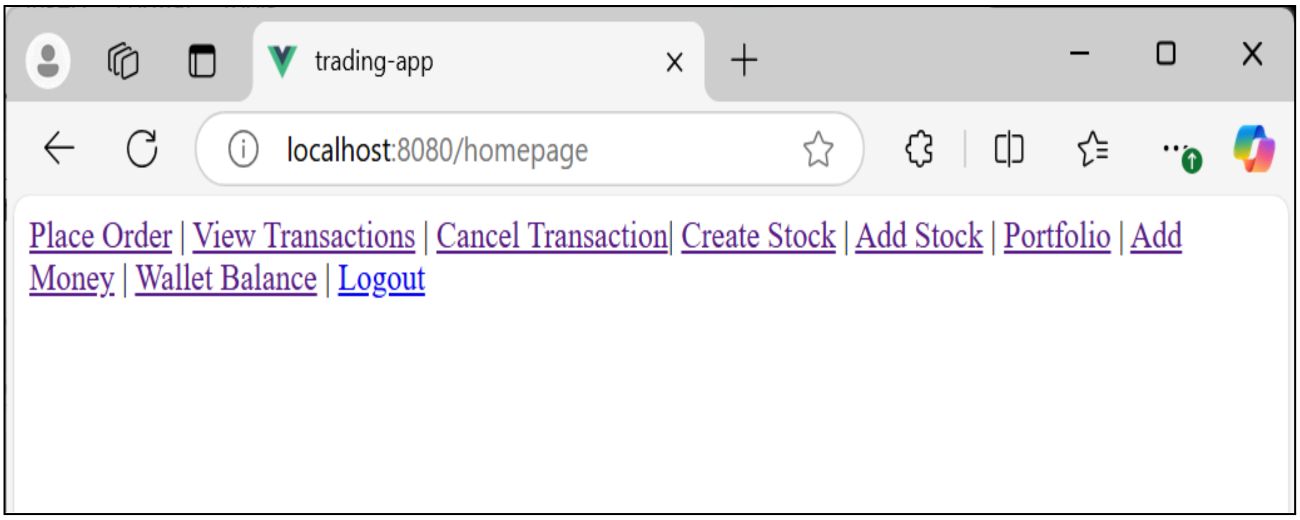
- **MongoDB**: Now central for authentication (Go-based microservice), transaction history, and stock portfolio data. Its sharding and replication capabilities are heavily utilized to handle large-scale data operations efficiently.
- **Redis**: Caches frequently accessed data, ensuring fast system responses. A write-through approach is used in the authentication service to maintain consistency between Redis and MongoDB.
- **Sharding and replication**: Enhance database performance and scalability.

11. UI & UI Mockup

The user interface is designed to be intuitive and responsive. Real-time stock updates are displayed using API polling, ensuring that users have access to the latest market data. Secure session management is enforced using JWT authentication, preventing unauthorized access to trading functionalities.

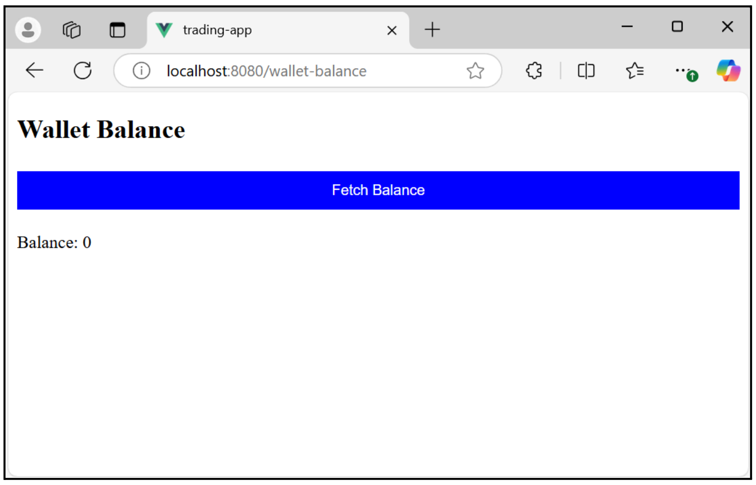





trading-app

localhost:8080/homepage

Place Order | View Transactions | Cancel Transaction| Create Stock | Add Stock | Portfolio | Add Money | Wallet Balance | Logout

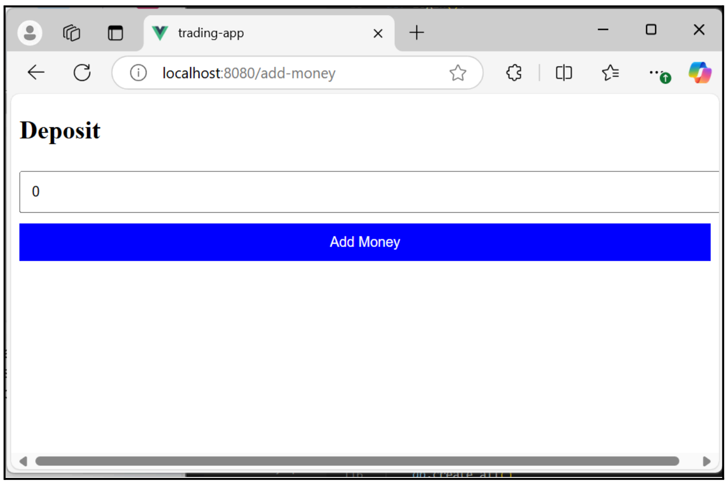

trading-app

localhost:8080/wallet-balance

Wallet Balance

Fetch Balance

Balance: 0

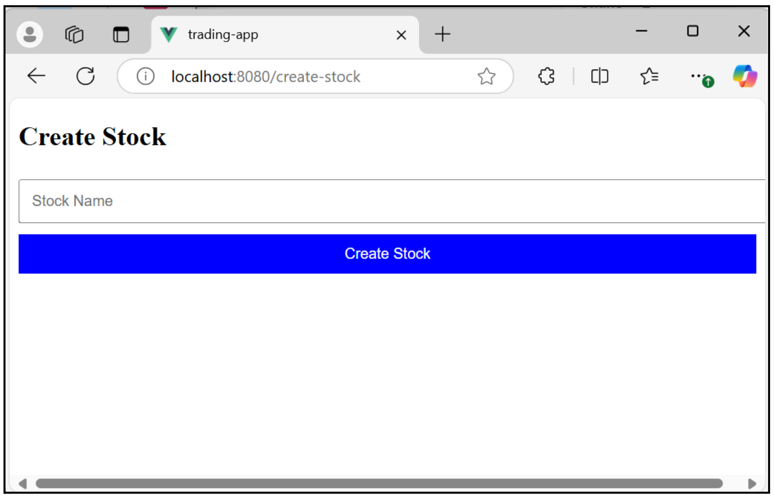

trading-app

localhost:8080/add-money

Deposit

0

Add Money

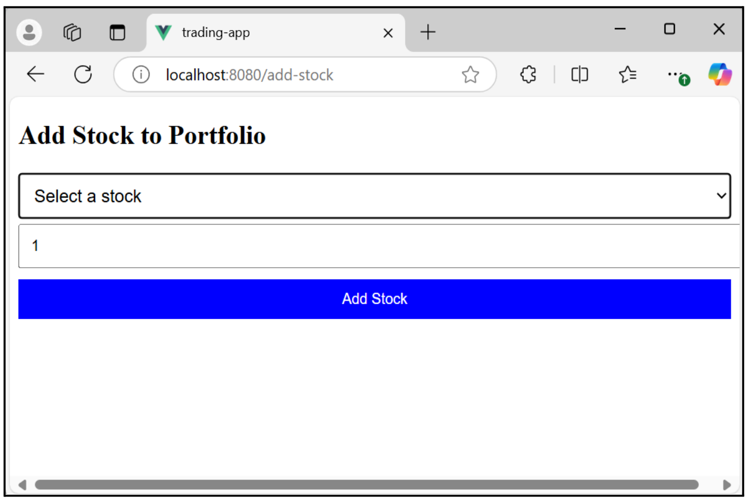

trading-app

localhost:8080/create-stock

Create Stock

Stock Name

Create Stock


trading-app

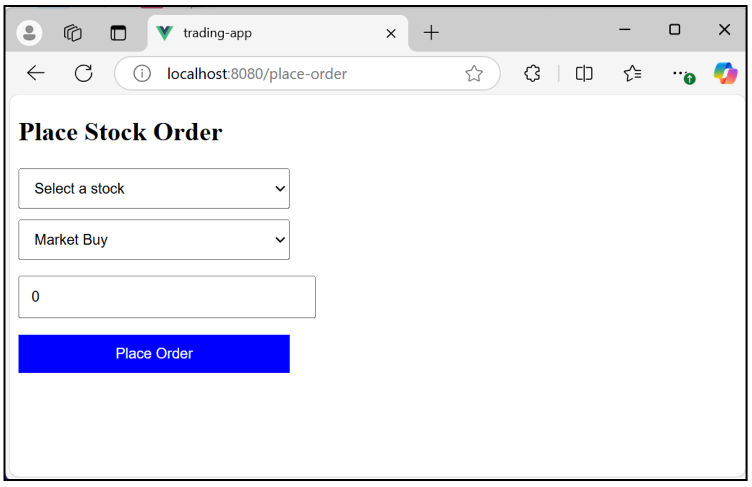
localhost:8080/add-stock

Add Stock to Portfolio

Select a stock

1

Add Stock



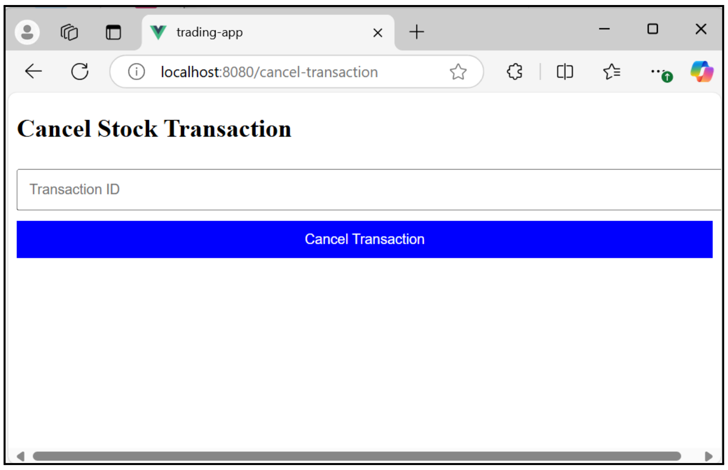
Place Stock Order

Select a stock

Market Buy

0

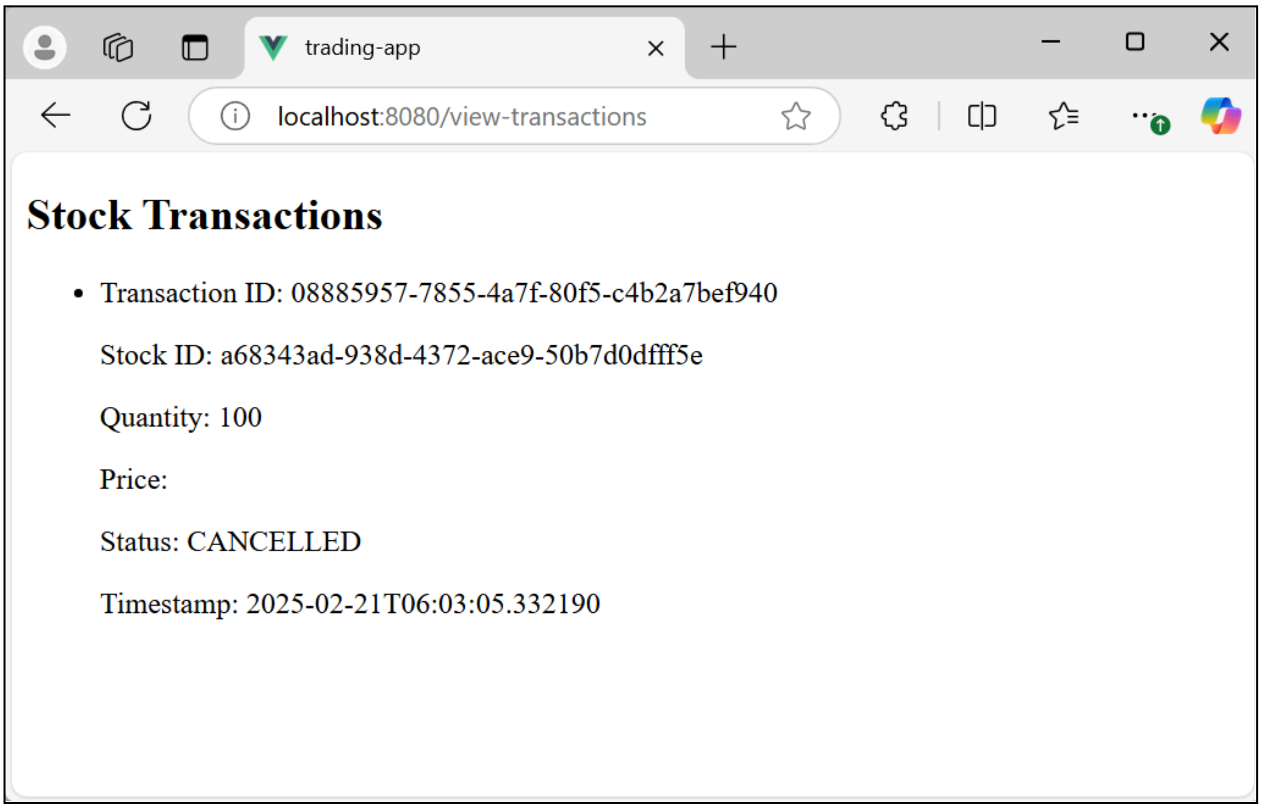
Place Order



Cancel Stock Transaction

Transaction ID

Cancel Transaction



Stock Transactions

- Transaction ID: 08885957-7855-4a7f-80f5-c4b2a7bef940

  Stock ID: a68343ad-938d-4372-ace9-50b7d0dfff5e

  Quantity: 100

  Price:

  Status: CANCELLED

  Timestamp: 2025-02-21T06:03:05.332190

12. References and Documentation

[1] Redis, "Scaling Redis — Redis Documentation," Redis.io, Available: https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/. [Accessed: 31-Jan-2025].

[2] GeeksforGeeks, "Can we use both Load Balancer and API Gateway?," GeeksforGeeks, Available: https://www.geeksforgeeks.org/can-we-use-both-load-balancer-and-api-gateway/. [Accessed: 31-Jan-2025].

[3] MongoDB, "Replication — MongoDB Manual," MongoDB Documentation, Available: https://www.mongodb.com/docs/manual/replication/. [Accessed: 31-Jan-2025].

[4] Timescale, "Guide to PostgreSQL Scaling," *Timescale*, [Online]. Available: https://www.timescale.com/learn/guide-to-postgresql-scaling. [Accessed: Jan. 31, 2025].

[5] Timescale, "Understanding ACID Compliance," *Timescale Documentation*, Available: https://www.timescale.com/learn/understanding-acid-compliance. [Accessed: 31-Jan-2025].

[6] Planeks, "Is Python Good for Microservices?" *Planeks.net*, [Online]. Available: https://www.planeks.net/is-python-good-for-microservices/. [Accessed: 27-Mar-2025].