Documentation

DEX Order Execution Engine

Eterna Labs
Backend Assignment

Rajneesh Kumar
220121047
rajneesh.kumar@iitg.ac.in

December 14, 2025

Contents

1 Introduction

Decentralized exchanges (DEXs) allow users to trade tokens directly without relying on a central authority. However, executing trades across multiple DEX platforms is not straightforward. Each DEX can offer different prices, liquidity, and fees at any given time. To achieve the best execution, a system must compare these options and route the order accordingly.

In addition, trading systems must handle multiple orders arriving at the same time while providing clear feedback to users. Without proper design, such systems may suffer from delays, failed executions, or lack of visibility into order progress.

This project focuses on building an Order Execution Engine that processes orders in a controlled and observable manner, while providing real-time updates to users throughout the order lifecycle.

1.1 Problem Statement

The main challenge addressed in this project is the design of a backend system that can reliably execute trading orders while handling concurrency, routing decisions, and execution status tracking.

The system must:

- Accept user orders through a REST API
- Process orders asynchronously without blocking incoming requests
- Compare execution prices across multiple DEXs
- Select the best execution route based on price and liquidity
- Provide real-time status updates to users
- Handle failures and retries in a predictable manner

Without such a system, users have limited visibility after submitting an order and no assurance that the best available execution path was chosen.

1.2 Project Overview

The **DEX Order Execution Engine** is a backend application designed to execute **market orders**. Market orders were chosen because they represent the most basic and commonly used order type, making them ideal for demonstrating execution flow, routing logic, and real-time updates.

When a user submits an order, it is placed into an internal queue and processed asynchronously. The engine fetches price quotes from Raydium and Meteora, compares them, and routes the order to the DEX offering the better execution. Throughout this process, the user receives live status updates through a WebSocket connection.

The system uses a mock implementation for DEX interaction, allowing focus on architecture, concurrency handling, and execution flow rather than blockchain-specific complexities. The same design can be extended in the future to support limit and sniper orders with minimal changes.

1.3 Key Features

- Market order execution with asynchronous processing
- Price comparison and routing between Raydium and Meteora
- Queue-based execution to manage concurrent orders
- Real-time order status updates using WebSockets
- Retry mechanism with failure handling
- Basic slippage protection during execution

1.4 Technology Stack

The Order Execution Engine is developed using a Java-based backend and a simple web interface for monitoring orders and receiving live updates.

- **Backend:** Spring Boot 3.2 with Java 17, used to build REST APIs, handle business logic, and process orders asynchronously.
- **Frontend:** Thymeleaf, used to create server-side rendered pages and dashboards.
- **Database:** PostgreSQL, used to store order details and execution history.
- **Queue Management:** An in-memory `LinkedBlockingQueue` is used to manage order flow and concurrent processing.
- **WebSocket Communication:** Spring WebSocket with STOMP is used to send real-time order status updates to clients.

2 Setup and Execution

2.1 Prerequisites

Before running the application, ensure the following software is installed:

- **Java 17**
- **Maven**
- **PostgreSQL**

2.2 Application Configuration

The application settings are defined in the `application.properties` file as shown below:

```
spring.application.name=Order Execution Engine

server.port=5000
server.address=0.0.0.0

```

```
spring.datasource.url=jdbc:postgresql://${PGHOST}/${PGDATABASE}?sslmode=disable
spring.datasource.username=${PGUSER}
spring.datasource.password=${PGPASSWORD}
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

spring.thymeleaf.cache=false
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html

spring.task.execution.pool.core-size=10
spring.task.execution.pool.max-size=20
spring.task.execution.pool.queue-capacity=100
```

2.3 Running the Application Locally

The application can be started using the following command:

```
mvn spring-boot:run
```

After the application starts, the dashboard can be accessed at:

`http://localhost:5000`

2.4 Environment Variables

The following environment variables must be set for database connectivity:

- `PGHOST` – PostgreSQL host (for example, localhost)
- `PGDATABASE` – Database name (for example, order_engine)
- `PGUSER` – Database username
- `PGPASSWORD` – Database password

3 Frontend Overview

The frontend dashboard provides the following features:

- **Order Form:** Allows users to submit token pairs, amount, and slippage.
- **Live Orders:** Displays real-time order status, executed price, and transaction hash.
- **Queue Statistics:** Shows the number of active and queued orders.
- **Activity Log:** Displays WebSocket connection status and order-related events.

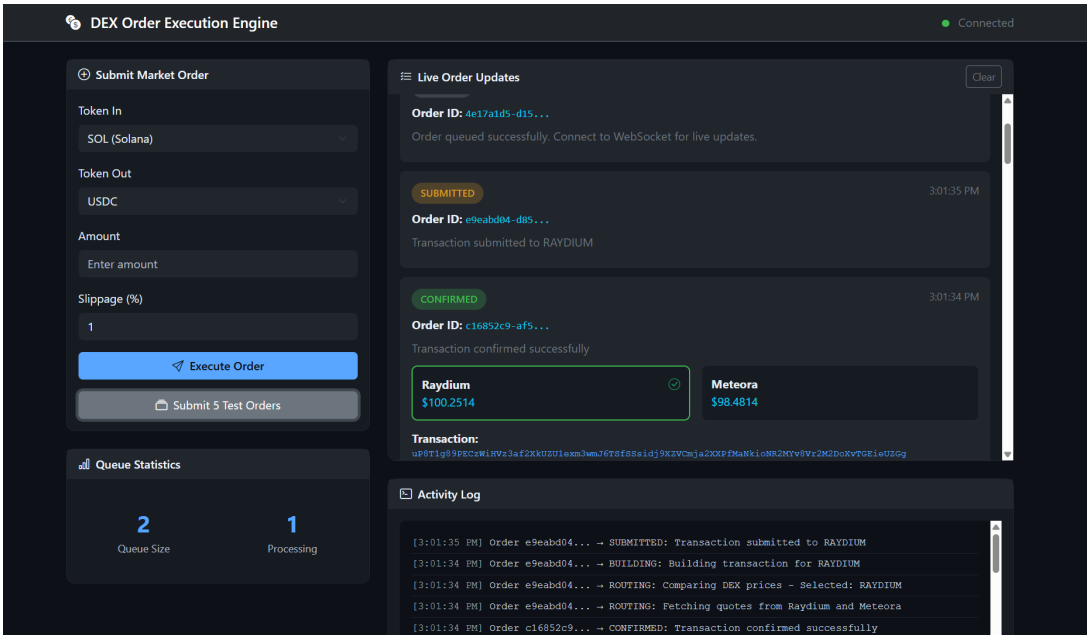
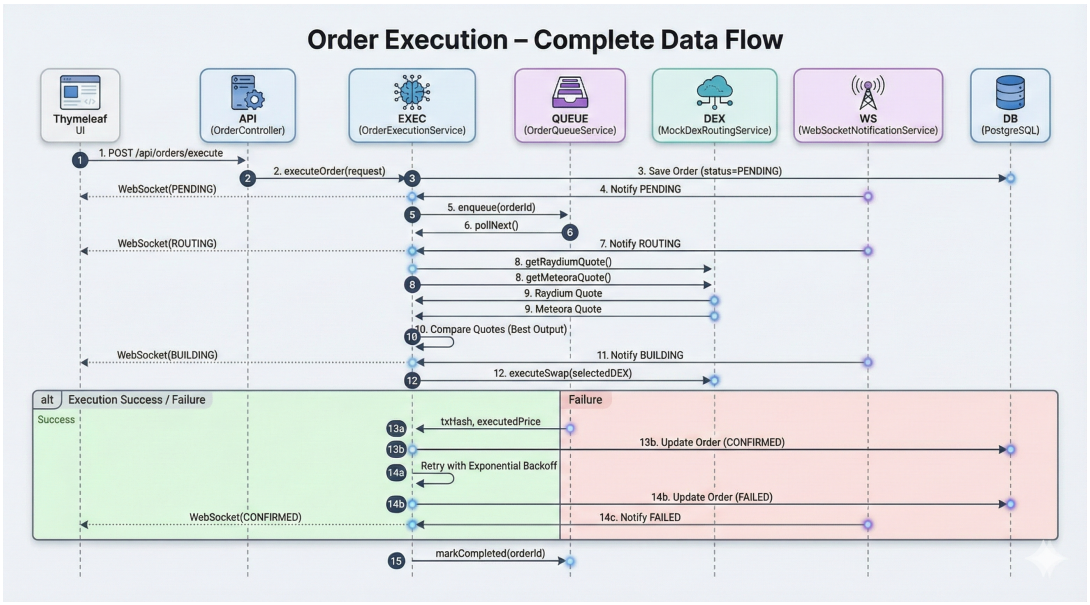


Figure 1: DEX Order Execution Engine in action: submitting and executing market orders.

3.1 Order Execution



4 Package Structure

The project follows a standard layered package structure commonly used in Spring Boot applications. Each package has a specific responsibility, which helps in maintaining clean code, better readability, and easy future extension.

4.1 config

This package contains all configuration-related classes required by the application.

It includes:

- Asynchronous execution configuration
- WebSocket and STOMP configuration

These classes define how background tasks and real-time communication are handled by the system.

4.2 controller

The controller package handles all incoming HTTP requests from the client.

Its responsibilities include:

- Exposing REST APIs for order submission and status queries
- Handling web page requests for the Thymeleaf frontend
- Managing global exception handling

Controllers act as the entry point between the frontend and backend logic.

4.3 dto

The DTO (Data Transfer Object) package is used to transfer data between different layers of the application.

It helps in:

- Sending structured request data from the client
- Returning formatted responses to the frontend
- Exchanging WebSocket messages

Using DTOs keeps the internal data model separate from API responses.

4.4 model

The model package contains the core domain objects of the system.

This includes:

- Order entity
- Enums for order status, order type, and DEX type

These classes represent how data is stored and processed within the system.

4.5 repository

The repository package is responsible for database access.

It:

- Communicates with PostgreSQL using Spring Data JPA
- Handles saving and retrieving order records

This layer abstracts database operations from the business logic.

4.6 service

The service package contains the core business logic of the application.

Its responsibilities include:

- Order execution and state management
- Queue handling and concurrency control
- DEX price simulation and routing
- WebSocket-based real-time notifications

This package acts as the backbone of the entire order execution engine.

5 Model Layer

The model layer represents the core domain entities of the Order Execution Engine. It defines how data is structured, stored, and transitioned across different stages of order execution.

All models are implemented using **JPA entities** and **enumerations**, ensuring strong type safety and persistence consistency.

5.1 Order Entity

The `Order` entity represents a single trade request submitted by a user. It captures the complete lifecycle of an order, from submission to execution or failure.

5.1.1 Entity Definition

The `Order` class is mapped to the `orders` table using JPA annotations. Each order is uniquely identified using a UUID-based primary key.

5.1.2 Core Order Attributes

- **tokenIn**: Input token for the swap
- **tokenOut**: Output token for the swap
- **amount**: Quantity of input tokens to be swapped
- **slippage**: Maximum acceptable price deviation
- **orderType**: Type of order (Market, Limit, Sniper)
- **status**: Current execution state of the order

All monetary values use `BigDecimal` to avoid floating-point precision errors.

5.1.3 DEX Execution Metadata

To support intelligent routing and auditability, the order stores DEX-related information:

- **selectedDex**: DEX chosen for execution
- **raydiumQuote**: Quoted price from Raydium
- **meteorQuote**: Quoted price from Meteora
- **executedPrice**: Final execution price
- **txHash**: Blockchain transaction hash

These fields allow post-trade analysis and transparency.

5.1.4 Failure and Retry Handling

The order entity also includes fields for fault tolerance:

- **retryCount**: Number of execution attempts
- **errorMessage**: Reason for failure (if any)

This enables controlled retries and proper failure reporting.

5.1.5 Lifecycle Timestamps

The entity maintains timestamps to track execution flow:

- **createdAt**: Order creation time
- **updatedAt**: Last modification time
- **completedAt**: Final completion or failure time

5.1.6 Automatic Timestamp Management

The `@PrePersist` and `@PreUpdate` hooks ensure:

- Automatic timestamp initialization
- Default status assignment
- Retry counter initialization

This removes manual boilerplate logic from service layers.

5.2 OrderType Enumeration

The `OrderType` enum defines supported order strategies:

- **MARKET**: Immediate execution at best available price
- **LIMIT**: Execution only at a specified price
- **SNIPER**: High-priority execution strategy

This design allows easy extension of order strategies in the future.

5.3 OrderStatus Enumeration

The `OrderStatus` enum represents the execution state machine of an order.

5.3.1 Defined States

- **PENDING**: Order accepted and queued
- **ROUTING**: Fetching prices from DEXs
- **BUILDING**: Transaction construction phase
- **SUBMITTED**: Transaction sent to blockchain
- **CONFIRMED**: Execution successfully completed
- **FAILED**: Execution failed after retries

5.3.2 State Design Rationale

Each state represents a real execution milestone. This fine-grained state model enables:

- Accurate real-time monitoring
- Better debugging and logging
- Clear UI status representation

It forms a reliable foundation for scalable order execution and monitoring.

6 Configuration Layer

The configuration layer defines system-wide infrastructure settings such as asynchronous execution, thread management, and real-time communication support. These configurations ensure scalability, responsiveness, and non-blocking order execution.

6.1 Asynchronous Execution Configuration

The `AsyncConfig` class enables asynchronous processing using Spring's `@EnableAsync` support. This allows order execution and DEX routing logic to run in parallel without blocking REST API threads.

6.1.1 Order Execution Thread Pool

A dedicated executor named `orderExecutor` is used for processing order execution tasks.

- Core pool size: 10 threads
- Maximum pool size: 20 threads
- Queue capacity: 100 tasks
- Thread name prefix: `OrderExecutor`

This executor is optimized for handling multiple order submissions concurrently while maintaining controlled resource usage.

6.1.2 DEX Routing Thread Pool

The `dexRoutingExecutor` is used specifically for routing orders to decentralized exchange adapters.

- Core pool size: 5 threads
- Maximum pool size: 10 threads
- Queue capacity: 50 tasks
- Thread name prefix: `DexRouter`

Separating DEX routing from order execution prevents routing delays from impacting overall system throughput.

6.1.3 Design Advantages

- Prevents API thread blocking
- Enables parallel order processing
- Isolates execution workloads
- Improves system responsiveness under high load

6.2 WebSocket Configuration

The `WebSocketConfig` class enables real-time bidirectional communication between the backend and frontend dashboard.

6.2.1 STOMP Message Broker Setup

A simple in-memory message broker is configured with the following destinations:

- `/topic` for broadcast messages
- `/queue` for point-to-point messages

The application destination prefix is set to:

`/app`

This prefix is used for messages sent from the client to the server.

6.2.2 User-Specific Messaging

The user destination prefix:

`/user`

allows sending private updates to individual clients, such as order-specific execution status.

6.2.3 WebSocket Endpoint Registration

The WebSocket endpoint is exposed at:

`/ws`

- Supports SockJS fallback for browser compatibility
- Allows cross-origin connections
- Enables real-time UI updates in the Thymeleaf dashboard

6.3 Configuration Layer Benefits

The configuration layer provides:

- Efficient thread management
- Real-time order status updates
- Scalable asynchronous execution
- Clean separation of infrastructure concerns

This layer forms the backbone for building a responsive and production-ready order execution engine.

7 Controller Layer

The controller layer acts as the entry point of the application. It receives client requests, validates input data, and forwards the requests to the service layer for processing.

This project uses both:

- **REST controllers** for API-based interaction
- **Web controllers** for Thymeleaf-based UI rendering

7.1 Order REST Controller

The `OrderController` exposes REST endpoints to submit and monitor orders. All APIs are grouped under the base path `/api/orders`.

7.1.1 Execute Order Endpoint

Endpoint:

`/api/orders/execute`

Method: `POST`

This endpoint accepts a new order request and submits it to the execution engine.

- Request body is validated using `@Valid`
- On success, the order is placed into the execution queue
- A response is returned with order ID and initial status

This design ensures that invalid orders are rejected early before execution.

7.1.2 Get Order by ID

Endpoint:

`/api/orders/{orderId}`

Method: `GET`
This endpoint retrieves the complete execution details of a specific order.

- Returns order status, quotes, selected DEX, and execution result
- Returns `404 Not Found` if the order ID does not exist

7.1.3 Get Recent Orders

Endpoint:

`/api/orders`

Method: `GET`
Returns a list of recently created orders. This endpoint is mainly used for monitoring and dashboard display.

7.1.4 Queue Statistics Endpoint

Endpoint:

`/api/orders/queue/stats`

Method: `GET`
Provides real-time queue metrics, including:

- Current queue size
- Active processing count
- Maximum queue capacity
- Maximum concurrent executions

This endpoint is useful for system health monitoring.

7.2 Web Controller (Thymeleaf)

The `WebController` is responsible for rendering the dashboard UI using Thymeleaf.

7.2.1 Home Page Controller

Endpoint:

`/`

This controller:

- Loads recent orders from the database
- Fetches live queue statistics
- Passes data to the `index.html` Thymeleaf template

It allows users to monitor order execution status in real time through the browser.

7.3 Global Exception Handling

The `GlobalExceptionHandler` provides centralized error handling for all controllers.

7.3.1 Validation Error Handling

When request validation fails:

- Field-level error messages are collected
- A structured JSON response is returned
- HTTP status `400 Bad Request` is sent

This improves API usability and provides clear feedback to clients.

7.3.2 Generic Exception Handling

Any unexpected runtime exception:

- Is logged for debugging
- Returns a standard error response
- Uses HTTP status `500 Internal Server Error`

This prevents application crashes and ensures consistent error responses.

7.4 Controller Layer Design Benefits

The controller layer ensures:

- Clear separation between API and business logic
- Input validation at the boundary
- Consistent error handling
- Support for both REST APIs and web UI

This makes the system easy to extend and maintain.

8 Service Layer

The service layer contains the core business logic of the DEX Order Execution Engine. It handles order submission, queue processing, DEX price routing, execution flow, retry handling, and real-time notifications. Each service has a clear responsibility and communicates with others to complete the full order lifecycle.

8.1 OrderExecutionService

The `OrderExecutionService` is the core service of the system. It controls the complete lifecycle of an order, starting from submission, queueing, execution, retry handling, and final confirmation or failure.

8.1.1 Order Submission Flow

When a user submits a market order, the service:

- Creates a new `Order` entity
- Sets the initial status to `PENDING`
- Saves the order in the database

The order is then placed into an execution queue. If the queue is full, the order is immediately marked as `FAILED` and the user is informed.

Once successfully queued, a WebSocket message is sent to notify that the order is waiting for execution.

8.1.2 Queue Processing Mechanism

The queue is processed continuously using a scheduled task that runs every 100 milliseconds.

- The next order ID is fetched from the queue
- The order is executed asynchronously

This design allows the system to handle multiple orders without blocking the main application thread.

8.1.3 Asynchronous Order Execution

Each order is executed in a separate thread using an asynchronous executor. This ensures:

- Non-blocking order execution
- Better concurrency handling
- Smooth processing under high load

The order is first fetched from the database to ensure data consistency.

8.1.4 Routing and Quote Fetching

Once execution begins, the order status is updated to `ROUTING`. The system then requests price quotes from:

- Raydium
- Meteora

Both quotes are fetched in parallel using `CompletableFuture`. After receiving both responses, the system:

- Stores both quotes in the order
- Selects the DEX offering the better output

A routing update is sent to the user via WebSocket.

8.1.5 Transaction Build and Submission

After selecting the best DEX:

- The order status is set to `BUILDING`
- The transaction is prepared
- The order is then marked as `SUBMITTED`

These steps simulate real DEX transaction preparation and submission delays.

8.1.6 Swap Execution

The actual swap is performed by the routing service. The result can be:

- Successful execution
- Execution failure (network or timeout error)

On success:

- Status is updated to `CONFIRMED`
- Executed price and transaction hash are stored
- Completion time is recorded

The queue is updated and a confirmation message is sent to the user.

8.1.7 Retry Logic with Exponential Backoff

If execution fails, the service retries the order up to **3 times**.

The retry delay follows an exponential backoff strategy:

$$Delay = 1000 \times 2^{(retryCount-1)} \; milliseconds$$

This results in retry delays of:

- 1 second (1st retry)
- 2 seconds (2nd retry)
- 4 seconds (3rd retry)

During retry:

- Order status is reset to `PENDING`
- The order is re-enqueued
- User is notified through WebSocket

If all retries fail, the order is marked as `FAILED`.

8.1.8 Failure Handling

Two types of failures are handled:

- Execution failures during swap
- Unexpected system errors

In both cases, the retry mechanism is triggered. If retries are exhausted, the order is permanently failed and removed from the queue.

8.1.9 Order Query APIs

The service also provides read-only methods:

- Fetch a single order by ID
- Fetch the 100 most recent orders

These methods convert internal order data into response DTOs suitable for APIs and dashboards.

8.1.10 Design Benefits

This service design provides:

- Clear order state transitions
- Reliable retry handling
- Real-time user feedback
- Scalable asynchronous execution

It closely reflects real-world order execution engines while remaining simple and testable.

8.2 OrderQueueService

The `OrderQueueService` is responsible for managing order flow and controlling concurrency in the system. It ensures that:

- The system does not process too many orders at the same time
- Orders are executed in a controlled and fair manner
- Queue overflow situations are handled safely

8.2.1 Queue Design

The service uses a bounded blocking queue to store pending orders.

- Maximum queue size: **100 orders**
- Each entry stores only the `orderId`

This keeps memory usage low and avoids loading full order objects into the queue.

8.2.2 Concurrency Control

To limit system load, the service allows only a fixed number of orders to be processed at the same time.

- Maximum concurrent executions: **10**
- Controlled using an `AtomicInteger`

Before an order is dequeued, the service checks whether the current processing count has reached the limit.

8.2.3 Active Order Tracking

The service maintains a thread-safe map of active orders:

- Key: `orderId`
- Value: `Order` object

This map allows:

- Quick lookup of running orders
- Accurate tracking of execution state

8.2.4 Enqueue Operation

When a new order is submitted:

- The service first checks if the queue is full
- If full, the order is rejected
- Otherwise, the order ID is added to the queue

The order is also added to the active order map for tracking.

8.2.5 Dequeue and Execution Control

Orders are fetched using the `pollNext()` method:

- If the maximum concurrent limit is reached, no order is dequeued
- If allowed, one order ID is removed from the queue
- Processing count is incremented

This mechanism ensures that the system never exceeds its execution capacity.

8.2.6 Completion and Failure Handling

After execution finishes, the service updates its internal state.

- On success, `markCompleted()` is called
- On failure, `markFailed()` is called

In both cases:

- The order is removed from active tracking
- Processing count is decremented

This frees capacity for new orders.

8.2.7 Monitoring and Statistics

The service provides real-time queue statistics, including:

- Current queue size
- Number of orders being processed
- Active order count
- Maximum limits

These values are useful for:

- Debugging
- System monitoring
- Admin dashboards

8.2.8 Why This Design Works

This queue-based approach provides:

- Back-pressure when load is high
- Safe multi-threaded execution
- Predictable system behavior

It closely matches real-world order execution engines while keeping the implementation simple and reliable.

8.3 MockDexRoutingService

The `MockDexRoutingService` simulates the behavior of decentralized exchanges (DEXs) without connecting to a real blockchain network. It is used to generate price quotes, compare different DEXs, and simulate swap execution with realistic delays and failures.

8.3.1 DEX Price Quote Generation

This service provides price quotes from two mock DEXs:

- Raydium
- Meteora

A base SOL price of **100.00** is used for all calculations. Random variations are applied to simulate real market movement.

Raydium Quote Logic For Raydium, the price multiplier is calculated as:

$$Multiplier_R = 0.98 + random(0, 0.04)$$

The final price is:

$$Price_R = BasePrice \times Multiplier_R$$

Raydium charges a fixed fee of **0.3%**. The output amount is calculated as:

$$Output_R = Amount \times Price_R \times (1 - 0.003)$$

A random response delay between 150–250 ms is added to simulate network latency.

Meteora Quote Logic For Meteora, the price multiplier is:

$$Multiplier_M = 0.97 + random(0, 0.05)$$

The final price is:

$$Price_M = BasePrice \times Multiplier_M$$

Meteora charges a lower fee of **0.2%**. The output amount is calculated as:

$$Output_M = Amount \times Price_M \times (1 - 0.002)$$

A response delay between 180–300 ms is applied.

8.3.2 Best DEX Selection

After receiving quotes from both DEXs, the service compares their output amounts. The DEX that provides the higher output value is selected for execution.

$$BestDEX = \{\ Raydium, if Output_R > Output_M Meteora, otherwise$$

This ensures the user gets the maximum return for their order.

8.3.3 Swap Execution Simulation

The swap execution process is simulated with:

- Execution delay of 2–3 seconds
- A 5% probability of execution failure

If a failure occurs, the order is marked unsuccessful with a timeout error.

Slippage Calculation To simulate price movement during execution, slippage is applied:

$$SlippageFactor = 1 - random(0, 0.01)$$

The executed price becomes:

$$ExecutedPrice = QuotedPrice \times SlippageFactor$$

This models real-world price fluctuation during transaction execution.

8.3.4 Transaction Hash Generation

On successful execution, a mock transaction hash is generated. The hash consists of 88 randomly selected Base58 characters, matching the format used by Solana transactions.

8.3.5 Purpose of Mock Implementation

This service allows:

- Testing full order execution flow
- Validating routing and retry logic
- Demonstrating system behavior without real funds

It keeps the project focused on backend architecture and execution logic rather than blockchain connectivity.

8.4 WebSocketNotificationService

The `WebSocketNotificationService` is responsible for sending real-time order updates to connected clients. It allows users to monitor order progress without repeatedly calling REST APIs.

This service uses **Spring WebSocket with STOMP protocol** to push updates whenever the order state changes.

8.4.1 Purpose of WebSocket Communication

Order execution is a long-running process that passes through multiple stages such as routing, submission, and confirmation. WebSocket communication is used to:

- Provide live status updates
- Reduce unnecessary API polling
- Improve user experience on the dashboard

8.4.2 Message Structure

All WebSocket updates are sent using a `WebSocketMessage` object. Each message contains:

- Order ID
- Current order status
- Human-readable message
- Timestamp
- Optional execution details (DEX, price, quotes, transaction hash)

This keeps message formatting consistent across the system.

8.4.3 Order Status Updates

The `notifyOrderStatus()` method sends basic status updates such as:

- Order received
- Queued for execution
- Transaction building
- Transaction submitted

Each update is broadcast to:

- A specific order channel
- A global orders channel

8.4.4 DEX Routing Updates

During price comparison, the `notifyRouting()` method sends:

- Raydium quote
- Meteora quote
- Selected DEX

This allows the frontend to visually show how routing decisions are made.

8.4.5 Confirmation Notifications

When an order is successfully executed, `notifyConfirmed()` is triggered. The message includes:

- Executed price
- Selected DEX
- Transaction hash
- Final confirmation timestamp

This marks the completion of the order lifecycle.

8.4.6 Failure Notifications

If execution fails after retries, `notifyFailed()` is used to:

- Notify the user about failure
- Share a readable error message

This ensures failure states are clearly visible on the UI.

8.4.7 WebSocket Destinations

Messages are sent to two types of STOMP destinations:

- `/topic/orders` for global monitoring
- `/topic/orders/{orderId}` for order-specific updates

This design supports both individual order tracking and system-wide dashboards.

8.4.8 Why This Design Works

This service provides:

- Real-time feedback
- Clear separation of communication logic
- Scalable event-driven updates

It integrates smoothly with the execution and queue services, completing the real-time order execution flow.

9 Order Lifecycle

The order lifecycle describes the sequential and concurrent stages that an order undergoes from submission by the client to execution on a decentralized exchange (DEX) and final status reporting.

9.1 Order Submission

- The client submits an order via the REST API endpoint `/api/orders/execute`.
- The `OrderExecutionService` validates the request and creates a new `Order` entity in the database with status `PENDING`.
- The order is then enqueued in `OrderQueueService` for asynchronous processing.
- A WebSocket notification is sent to the client indicating that the order has been queued successfully.

9.2 Queue Management

- `OrderQueueService` maintains a bounded queue (`MAX_QUEUE_SIZE = 100`) and tracks active processing orders (`MAX_CONCURRENT_ORDERS = 10`).
- Orders are dequeued and assigned to the asynchronous executor `orderExecutor` when processing slots are available.
- If the queue is full, orders are rejected, and the client is notified of the failure.

9.3 Order Routing and Quote Fetching

- The system requests price quotes from multiple DEXs (`Raydium` and `Meteora`) via `MockDexRoutingService`.
- Quotes are fetched asynchronously using `CompletableFuture`s, which allows simultaneous retrieval without blocking the main thread.
- Each quote includes:
  - Estimated price (`price`)
  - Transaction fee (`fee`)
  - Expected output amount (`outputAmount`)
  - Response time in milliseconds (`responseTimeMs`)
- The best quote is selected based on maximum expected output.
- WebSocket notifications are sent to the client indicating the selected DEX and the quotes compared.

9.4 Transaction Building

- The order status is updated to `BUILDING`.
- The system simulates transaction construction for the selected DEX.
- This step ensures that all transaction parameters, such as slippage and execution price, are finalized.
- Clients receive real-time updates via WebSocket.

9.5 Order Submission to DEX

- The constructed transaction is submitted to the selected DEX.
- The status is updated to `SUBMITTED`.
- The execution occurs asynchronously through `dexRoutingExecutor`.
- A simulated delay models network and blockchain latency.

9.6 Order Execution and Confirmation

- The execution result is captured in `ExecutionResult`:
  - `success`: true if transaction succeeded, false otherwise
  - `txHash`: simulated transaction hash
  - `executedPrice`: actual execution price with possible slippage
  - `errorMessage`: reason for failure
- On success:
  - Status is updated to `CONFIRMED`
  - Completion timestamp is recorded
  - WebSocket notification is sent with full order execution details
- On failure:
  - Retry logic is invoked with exponential backoff
  - Maximum retry count is `MAX_RETRY_COUNT = 3`
  - If retries are exhausted, status is updated to `FAILED` and an error message is sent via WebSocket

9.7 Post-Execution

- Active order is removed from `OrderQueueService` tracking.
- Completed orders remain in the database for reporting and analytics.
- The client can query the final order status via REST API (`/api/orders/{orderId}`) or subscribe to WebSocket updates.

9.8 Summary of Status Transitions

Status	Description
PENDING	Order received and queued
ROUTING	Fetching and comparing DEX quotes
BUILDING	Transaction being constructed
SUBMITTED	Transaction submitted to DEX
CONFIRMED	Execution successful, finalized
FAILED	Execution failed after retries

10 Extending the Engine

- Limit Orders: Monitor price, execute when target reached
- Sniper Orders: Trigger on token launch events