Order Management System - Node.js Implementation

# 1. Project Overview

This project implements an Order Management System (OMS) using Node.js. The OMS handles incoming orders, processes them with a throttling mechanism, ensures orders are only sent during a defined trading window, supports order modifications and cancellations, and logs response latency to a file.

# 2. Key Features

• Supports Request Types: New, Modify, Cancel.

• Throttling: Limits order sending to 100 per second.

• Trading Window: Only allows orders from 10:00 AM to 1:00 PM IST.

• Queue: Excess orders are queued and sent in the next available second.

• Modify/Cancel support for queued orders.

• Latency Logging: Logs response time to 'order\_responses.log'.

# 3. Technologies Used

• Node.js (JavaScript)

• Built-in fs module for file I/O

• setInterval and async/await for timed and asynchronous logic

# 4. Throttling Mechanism

The system limits the number of orders sent to the exchange to 100 per second. If more than 100 orders are received in one second, the extra orders are stored in a queue. Every second, the system resets the counter and sends queued orders.

# 5. Code Explanation

## 5.1 enums.js

Defines enums for RequestType and ResponseType to categorize order actions and responses.

## 5.2 OrderManagement.js

Contains the logic for throttling, queue management, time checking, logon/logout, and response logging.

## 5.3 index.js

Simulates sending 105 orders, a modify, a cancel, and an async response using await and delay.

# 6. Async/Await vs setTimeout

The original response simulation used setTimeout. This was replaced with an async function using delay(ms), which returns a Promise and allows the use of await to simulate delayed response in a cleaner way.

# 7. Conclusion

This Order Management System simulates real-world exchange interaction, respecting constraints on trading hours and rate limits. It supports various request types and demonstrates core principles of event-driven and asynchronous programming in Node.js.

# 8. Code Organization and File Breakdown

The system follows a modular file structure, separating responsibilities into independent components. This not only improves readability and maintenance, but also allows better testing and scalability.

## Main Files:

• OrderManagement.js – Core logic for routing, throttling, queue handling, and trading window checks.

• OrderSender.js – Responsible for sending orders and tracking throttle limits per second.

• QueueManager.js – Manages enqueuing, modifying, and canceling of orders before they are sent.

• TradingWindow.js – Tracks current time and triggers logon/logout events based on schedule.

• ResponseLogger.js – Logs order response latency to 'order\_responses.log'.

• enums.js – Enum-like object holding RequestType and ResponseType values.

• index.js – Entry point. Simulates sending orders and validates system behavior.

# 9. Best Practices Followed

• Modularization: Logic is split across multiple files based on functionality (Single Responsibility Principle).

• Readability: Class and method names are descriptive and consistently use camelCase.

• Error Handling: The code checks for edge cases like trading window closure or non-existent orders.

• Logging: Order response time is logged to a file for analysis and debugging.

• Time-based Logic: Proper use of `setInterval()` and throttling ensures time-sensitive functionality is reliable.

• Export/Import: Uses `module.exports` and `require` consistently to manage module dependencies.

# 10. Suggested Improvements

• Add input validation for all order parameters (price, qty, etc.).

• Implement error logging in a separate file instead of console.

• Extend the project with HTTP API using Express.js for external integration.

• Add unit tests using a test framework like Mocha or Jest.

• Implement persistent storage for queued and sent orders (e.g., JSON file or database).