Real-Time Order Book Visualizer

Project Report

Deepanshu, IIT Delhi

November 4, 2025

Abstract

This report documents the design and implementation of a high-performance, real-time order book visualizer built with Next.js and TypeScript. The application streams live market data from Binance via its WebSocket API, renders a responsive order book with depth visualization, shows recent trades with directional highlighting, and maintains a smooth, low-latency user experience. The project fulfills all requirements specified in the assignment brief.

1 Introduction

The goal of this project was to build a robust, real-time financial UI that handles high-frequency data without UI jank. The application connects to Binance market streams to display:

- Live order book deltas (bids and asks).
- Aggregate trades for the recent trades feed.

The stack is intentionally simple and reliable: Next.js (React 18), TypeScript, and Tailwind CSS. The UI prioritizes clarity and performance over heavy visual effects.

2 Functionality and Features

Live WebSocket Feed

- Connects to Binance WebSocket streams for `<symbol>@aggTrade` and `<symbol>@depth@100ms`.
- Parses JSON payloads and updates local state incrementally.
- Removes price levels with zero quantity.

Order Book

- Two-column layout: bids (left, green) and asks (right, red).
- Sorting: bids in descending order by price; asks in ascending order.
- Columns per row: Price, Amount, and Total (cumulative).
- Spread displayed as: lowest ask minus highest bid.
- Depth visualization: a background bar per row with width proportional to the row's cumulative total relative to the side's maximum cumulative total.

Recent Trades

- Displays the 50 most recent trades.
- New top trade flashes color to indicate direction: green for market buy and red for market sell.
- Uses the `isBuyerMaker` flag from the stream to determine direction.

3 Architecture and Technology Choices

Framework and Language. Next.js (React 18) with TypeScript provides excellent DX, server integration, and strict typing for fewer runtime errors.

Styling. Tailwind CSS enables concise, consistent styling and a professional, responsive UI.

State and Data Structures. The order book stores price levels in JavaScript `Map` objects for near O(1) inserts, updates, and deletes keyed by price. Derived views (sorted arrays and cumulative totals) are memoized with `useMemo`.

Component Design. The WebSocket logic lives in a custom hook (`useBinanceSocket`). Presentation components such as `OrderBook`, `OrderBookRow`, `RecentTrades`, and `TradeRow` focus on rendering and are wrapped with `React.memo` where beneficial.

4 Implementation Details

4.1 WebSocket Hook

The custom hook:

1. Opens two sockets for a selected symbol: `@aggTrade` and `@depth@100ms`.
2. Parses messages and updates:
   - Trades: keeps a capped list of the latest 50.
   - Order book: applies deltas to bid and ask `Maps`; deletes price levels when quantity equals zero.
3. Cleans up sockets on unmount to prevent leaks.

4.2 Order Book Computation

For each side (bids and asks):

1. Convert the `Map` to an array and sort by price (desc for bids, asc for asks).
2. Slice to a reasonable window (e.g., 15 rows) for readability and performance.
3. Compute cumulative totals by a single linear pass.
4. Compute the maximum cumulative total for the side to normalize depth bar widths.

4.3 Recent Trades Display

Trades are rendered in reverse chronological order. The newest item temporarily receives a background flash based on direction. Formatting uses a monospaced font for numeric clarity and `toLocaleTimeString` for timestamps.

5 Performance Considerations

- **Memoization:** Sorting and cumulative totals are wrapped in `useMemo` to avoid recomputation on unrelated updates.
- **Minimal State:** Only essential data is kept in state; derived values are computed from source maps.
- **Granular Components:** Row components use `React.memo` to minimize rerenders.
- **Efficient Updates:** Order book deltas mutate `Map` copies and return new references to trigger React updates without rebuilding the entire structure.

6 Requirement Coverage

Assignment Requirement	Status
Connect to Binance WebSocket API (aggTrade and depth)	Implemented and verified
Parse events and maintain live order book with zero-qty removal	Implemented
Sort bids/asks, compute cumulative totals, and display columns	Implemented
Show spread between best ask and best bid	Implemented
Depth visualization with proportional bars	Implemented
Recent trades list (50) with directional flashing	Implemented
Responsive, non-blocking UI with memoization	Implemented
Clean TypeScript code and modular design	Implemented

7 Learning and Sources

At the start of this project, WebSockets were new to me. I learned how to subscribe, parse, and manage live streams by reading official documentation and watching introductory videos. The key resources were:

- Binance Spot API WebSocket Market Streams: Link
- General WebSocket explanations from publicly available tutorial videos (for example, Apna College channel introductions).

8 How to Run

1. Install dependencies: `npm install`
2. Start development server: `npm run dev`
3. Open: Link

Note: Ensure the selected symbol is valid for Binance spot markets (e.g., `BTCUSDT`, `ETHUSDT`).

9 Conclusion

The application meets the assignment goals by combining a clean UI, efficient state handling, and real-time data processing. It provides a clear, responsive visualization of the order book and recent trades, while the codebase remains concise and maintainable. Through this project, I strengthened practical skills in Next.js, TypeScript, Tailwind CSS, and WebSocket-based data flows.