Backend Task 2: Order Execution Engine

Problem Statement Build an order execution engine that processes ONE order type (market, limit, or sniper - your choice) with DEX routing and WebSocket status updates. You can choose between real devnet execution OR mock implementation.

How It Works - Order Execution Flow

1. **Order Submission**
    - User submits order via `POST /api/orders/execute`
    - API validates order and returns `orderId`
    - Same HTTP connection upgrades to WebSocket for live updates
    - Briefly document **why** you chose that order type and outline (1-2 sentences in README) how the same engine can be extended to support the other two.

2. **DEX Routing**
    - System fetches quotes from both Raydium and Meteora pools
    - Compares prices and selects best execution venue
    - Routes order to DEX with better price/liquidity

3. **Execution Progress (via WebSocket)**
    - "pending" - Order received and queued
    - "routing" - Comparing DEX prices
    - "building" - Creating transaction
    - "submitted" - Transaction sent to network
    - "confirmed" - Transaction successful (includes txHash)
    - "failed" - If any step fails (includes error)

4. **Transaction Settlement**
    - Executes swap on chosen DEX (Raydium/Meteora)
    - Handles slippage protection
    - Returns final execution price and transaction hash

Implementation Options

Option A: Real Devnet Execution (Bonus Points)

- Use actual Raydium/Meteora SDKs
- Execute real trades on devnet
- Deal with network latency and failures

Option B: Mock Implementation (Recommended)

- Simulate DEX responses with realistic delays (2-3 seconds)
- Focus on architecture and flow
- Mock price variations between DEXs (~2-5% difference)

Resources & References for real devnet

- Solana Libraries: @solana/web3.js, @solana/spl-token
- DEX SDKs: @raydium-io/raydium-sdk-v2, @meteora-ag/dynamic-amm-sdk
- Docs:
    - Raydium: https://github.com/raydium-io/raydium-sdk-V2-demo
    - Meteora: https://docs.meteora.ag/
- If using devnet: https://faucet.solana.com

Core Requirements

1. **Order Types (Choose ONE)**
    - Market Order - Immediate execution at current price
    - Limit Order - Execute when target price reached
    - Sniper Order - Execute on token launch/migration
2. **DEX Router Implementation**
    - Query both Raydium and Meteora for quotes
    - Route to best price automatically
    - Handle wrapped SOL for native token swaps
    - Log routing decisions for transparency
3. **HTTP → WebSocket Pattern**
    - Single endpoint handles both protocols
    - Initial POST returns orderId
    - Connection upgrades to WebSocket for status streaming

4. **Concurrent Processing**
    - Queue system managing up to 10 concurrent orders
    - Process 100 orders/minute
    - Exponential back-off retry (≤3 attempts). If still unsuccessful, emit "failed" status and persist failure reason for post-mortem analysis

Tech Stack

- Node.js + TypeScript
- Fastify (WebSocket support built-in)
- BullMQ + Redis (order queue)
- PostgreSQL (order history) + Redis (active orders)

Evaluation Criteria

- DEX router implementation with price comparison
- WebSocket streaming of order lifecycle
- Queue management for concurrent orders
- Error handling and retry logic
- Code organization and documentation

Deliverables

1. GitHub repo with clean commits.
    a. API with order execution and routing
    b. WebSocket status updates
    c. If real execution: Transaction proof (Solana Explorer link)
2. Link to GitHub docs/readme with basic documentation explaining design decisions and setup instructions
3. Deploy to free hosting - include public URL in README
4. 1-2 min public youtube video link that shows functionality
    a. Order flow through your system and design decisions
    b. Submit 3-5 orders simultaneously
    c. WebSocket showing all status updates (pending → routing → confirmed)
    d. DEX routing decisions in logs/console
    e. Queue processing multiple orders
5. ✅ Postman/Insomnia collection **plus** ≥10 unit/integration tests covering routing logic, queue behaviour, and WebSocket lifecycle

Please note all deliverables are **required**, we'll consider the application incomplete if any of the deliverables are missing.

Note: Focus on demonstrating solid architecture, routing logic, and real-time updates. Blockchain integration is optional - choose based on your comfort level with Solana development.

Mock Implementation Guide

```
class MockDexRouter {
  async getRaydiumQuote(tokenIn: string, tokenOut: string, amount: number)
{
    // Simulate network delay
    await sleep(200);
    // Return price with some variance
    return { price: basePrice * (0.98 + Math.random() * 0.04), fee: 0.003
};
  }

  async getMeteorQuote(tokenIn: string, tokenOut: string, amount: number) {
    await sleep(200);
    return { price: basePrice * (0.97 + Math.random() * 0.05), fee: 0.002
};
  }

  async executeSwap(dex: string, order: Order) {
    // Simulate 2-3 second execution
    await sleep(2000 + Math.random() * 1000);
    return { txHash: generateMockTxHash(), executedPrice: finalPrice };
  }
}
```

Real Implementation Examples

Raydium Swap Pattern:

```
// Initialize SDK
const raydium = await Raydium.load({
  owner: wallet,
  connection,
  cluster: 'devnet',
  blockhashCommitment: 'finalized'
});

// Get pool info and calculate swap
const { poolInfo, poolKeys, rpcData } = await
raydium.cpmm.getPoolInfoFromRpc(poolId);
const swapResult = CurveCalculator.swap(inputAmount, baseReserve,
quoteReserve, tradeFeeRate);

// Execute swap
const { execute } = await raydium.cpmm.swap({
  poolInfo,
  inputAmount,
  swapResult,
  slippage: 0.01
});
const { txId } = await execute({ sendAndConfirm: true });
```

Meteora Swap Pattern:

```
// Initialize SDK
const dynamicAmmSdk = await AmmImpl.create(connection, poolPubkey);
```

```
// Get quote
const quote = dynamicAmmSdk.getSwapQuote(tokenMint, amountIn,
slippageTolerance);

// For native SOL, wrap first
const wrapSolTx = new Transaction().add(
  SystemProgram.transfer({ fromPubkey: wallet.publicKey, toPubkey:
wrappedSolAccount, lamports }),
  createSyncNativeInstruction(wrappedSolAccount)
);

// Execute swap
const swapTx = await dynamicAmmSdk.swap(wallet.publicKey, tokenAccount,
amountIn, minAmountOut);
```