Coding Assessment

Timeline: 2 Days
Submission: Push to GitHub and email link to web@royalclass.group

Implement a **minimum viable** live bidding module for a car auction system focusing on core functionality that demonstrates your technical skills within the 2-day timeframe.

Building a Scalable Live Car Bidding System

Objective:

You are required to implement a live bidding module for a car auction system. The system should be able to handle high-frequency bids from multiple users in real-time. You must ensure the system is scalable, handles concurrency correctly, and is resilient against DDoS attacks.

Backend (NestJS)

1. WebSocket Gateway:

a. Implement a WebSocket gateway using NestJS and Socket.IO. The gateway should handle the following events:

- **joinAuction**: Clients (bidders) should be able to join a specific auction room.
- **placeBid**: Clients should place bids in real-time. The current highest bid should be broadcast to all connected clients in the auction room.
- **auctionEnd**: When the auction ends, notify all clients with the final winning bid.

b. Use Socket.IO with NestJS WebSocket decorators. Ensure that the connection is efficient and can handle multiple clients simultaneously.

c. **Hint**: You might need to use NestJS services and guards to manage auction rooms and their current bids.

2. Database Integration (PostgreSQL OR MongoDB):

Option A: PostgreSQL with Prisma ORM

a. Implement a PostgreSQL schema using Prisma to store:

- **Auctions** (Auction ID, Car ID, Start time, End time, Starting Bid, Current Highest Bid, Winner ID, Status, etc.)
- **Bids** (Bid ID, User ID, Auction ID, Bid Amount, Timestamp)
- **Users** (User ID, Username, Email, etc.)

b. Use Prisma transactions to ensure each bid placed is immediately stored in the PostgreSQL database.

c. **Concurrency Management**:

- Ensure that bids are processed correctly even when multiple users place bids at the same time using Prisma transactions.
- Implement database-level constraints and optimistic locking to avoid race conditions when multiple bids are placed simultaneously.

Option B: MongoDB with Mongoose ODM

a. Implement MongoDB schemas using Mongoose to store:

- **Auctions** (Auction ID, Car ID, Start time, End time, Starting Bid, Current Highest Bid, Winner ID, Status, etc.)
- **Bids** (Bid ID, User ID, Auction ID, Bid Amount, Timestamp)
- **Users** (User ID, Username, Email, etc.)

b. Use MongoDB transactions to ensure each bid placed is immediately stored in the database.

c. **Concurrency Management**:

- Ensure that bids are processed correctly using MongoDB's atomic operations and transactions.
- Implement proper indexing and use findOneAndUpdate with appropriate options to handle race conditions.

3. Redis Caching and Pub/Sub:

a. Use Redis with NestJS to cache the current highest bid for each auction to minimize database load.

b. Implement Redis Pub/Sub for real-time communication:

- Use Redis channels to broadcast bid updates across multiple server instances
- Implement cache invalidation: Whenever a new highest bid is placed, publish to Redis channels to update all connected clients
- Use Redis for session management and real-time data synchronization

c. **Redis Pub/Sub Implementation Requirements**:

- Create Redis publishers for bid events
- Implement Redis subscribers to listen for bid updates
- Handle connection failures and reconnection logic
- Use Redis channels for different auction rooms

5. Message Queue Integration (Choose One - RabbitMQ Preferred):

Option A: RabbitMQ (Preferred)

a. Implement RabbitMQ with NestJS for reliable message processing:

- Use RabbitMQ exchanges and queues for bid processing
- Implement dead letter queues for failed bid processing
- Use RabbitMQ for auction event notifications (start, end, bid updates)
- Implement message acknowledgments and retry mechanisms

b. **RabbitMQ Queue Structure**:

- **Bid Processing Queue**: Handle incoming bids with proper ordering
- **Notification Queue**: Send real-time notifications to users
- **Audit Queue**: Log all auction activities for compliance
- **Dead Letter Queue**: Handle failed message processing

Option B: Apache Kafka (Alternative)

a. Implement Kafka with NestJS for high-throughput event streaming:

- Use Kafka topics for different types of auction events
- Implement producers for bid events and consumers for processing
- Use Kafka partitioning for scalability across auction rooms
- Implement proper offset management and consumer groups

b. **Kafka Topics Structure**:

- **bid-events**: Stream of all bid placements
- **auction-notifications**: Real-time auction updates
- **audit-logs**: Complete audit trail of all activities

6. DDoS Mitigation Strategy:

a. Implement rate limiting and filtering using NestJS guards and interceptors to prevent DDoS attacks. This should include:

- Limiting WebSocket connections per user/IP using custom guards
- Request throttling for bids to prevent high-frequency spamming of bids from a