Object Design Document: Market Connect

Project: Market Connect – An Online Marketplace Platform

Version: 1.0

1. Introduction

- **1.1 Purpose:** The purpose of this document is to detail the object-level design of the **Market Connect** platform. It describes the system architecture, component decomposition, data management strategies, and the internal design of key objects (Entities and Controllers) required to implement the functional requirements.
- **1.2 Scope:** The system encompasses a full-featured e-commerce marketplace including buyer/seller workflows, real-time auctions, secure payment processing (Razorpay), and an AI-driven customer support chatbot.

2. High-Level Architecture

The project adopts a hybrid architecture that combines elements of **Monolithic**, **Layered**, and **Client-Server** designs to ensure clarity and maintainability.

A. Monolithic & Microservice Hybrid

- **Core Backend:** The primary business logic (Auth, Marketplace, Orders) is built as a unified Node.js application. This ensures simple deployment and low inter-module latency.
- **AI Microservice:** A separate Python/Flask service handles the Chatbot logic, communicating with the core backend via HTTP to fetch product/FAQ context.

B. Layered Architecture (MVC Pattern)

The backend is organized into logical layers, adhering to the **Separation of Concerns** principle:

1. **Presentation Layer (Routes):** Handles HTTP requests, performs initial validation using Joi middleware, and routes requests to controllers.
2. **Control/Service Layer (Controllers):** Contains the core business logic (e.g., Auction timing, Order calculations, Refund processing). It remains independent of direct database queries where possible.
3. **Data Access Layer (Models):** Manages interactions with MongoDB via Mongoose schemas. It defines data structure and validation rules.

C. Client–Server Architecture

- **Client (Frontend):** A React.js application providing the UI for buyers, sellers, and admins.

- **Server (Backend):** A Node.js/Express server that exposes RESTful APIs and WebSocket endpoints.

3. Component Identification and Allocation

This section identifies the major system components and their specific responsibilities.

1. Frontend Application (Client Layer)

- **Technology:** React.js
- **Responsibilities:**
    - Renders dynamic UI for Product Catalogs, Dashboards, and Auctions.
    - Manages client-side state (Cart, User Session).
    - Establishes WebSocket connections for live bidding.

2. Backend API Server (Server Layer)

- **Technology:** Node.js + Express
- **Responsibilities:**
    - **Authentication:** JWT generation, Google OAuth integration, and Role-based protection (Buyer/Seller/Admin).
    - **Marketplace Logic:** Product CRUD, Category management, and Review handling.
    - **Order Management:** Cart processing, Tax/Shipping calculation, and Order status lifecycle.

3. Real-Time Engine (Socket Layer)

- **Technology:** Socket.io
- **Responsibilities:**
    - Manages "Auction Rooms" where users join specific product channels.
    - Broadcasts bidUpdate events to all connected clients in real-time.
    - Triggers automatic order creation when an auction expires.

4. AI Support Service (Microservice)

- **Technology:** Python + Flask + Groq SDK
- **Responsibilities:**
    - Processes natural language user queries.
    - Retrieves context (FAQs, Products) from the Core Backend.
    - Generates intelligent responses using the Llama 3.1 model.

5. Database Layer (Data Storage)

- **Technology:** MongoDB Atlas
- **Responsibilities:** Persists Users, Products, Orders, Bids, and Chats.

6. External Integrations

- **Razorpay:** Handles payment processing and refunds.
- **Cloudinary:** Stores and serves optimized product images.
- **Google OAuth:** Provides secure social login.

4. Database Storage Strategy

Market Connect uses **MongoDB** for all structured data storage due to its flexibility with evolving schemas (e.g., varied Product Specs).

Data Types & Storage

1. **User Data:** Stores profile, hashed passwords, roles, and address sub-documents.
2. **Catalog Data:** Products are stored with references to Category and User (Seller). Auction details are embedded directly within the Product document for atomic updates.
3. **Transactional Data:** Orders and Payments are stored with strong consistency requirements.
4. **Media:** Images are **not** stored in the database; they are stored in **Cloudinary**, with only the secure URL retained in MongoDB.

5. Interface Communication Definition

This section defines how the modules of our project interact.

Client-Server Communication

- **Protocol:** HTTPS (REST) and WSS (WebSockets).
- **Format:** JSON.
- **Key Flows:**
    - POST /api/users/login => Returns JWT Token.
    - POST /api/orders/create => Accepts Cart/Item data, Returns Order ID.

Real-Time Communication

- **Events:** joinAuctionRoom, placeBid, bidUpdate, auctionEnded.
- **Flow:** Client emits placeBid => Server validates & saves => Server broadcasts bidUpdate.

Internal Service Communication

- **Chatbot:** The Python service makes HTTP GET requests to the Node.js backend (/api/products, /api/faqs) to fetch context before answering user queries.

6. Low Level Design (Object Design)

This section details the internal design of key entities and algorithms.

6.1 Entity Objects (Domain Models)

- **User (Buyer/Seller/Admin):** The central entity. Stores role-specific data like sellerInfo (Shop Name, Address) for sellers and buyerInfo (Cart, Addresses, Wishlist) for buyers.
- **Order:** The core transactional entity. It links a Buyer to a Seller and contains a list of OrderItems. Crucially, it tracks the financial breakdown (itemsPrice, taxPrice, shippingPrice, totalPrice) and the lifecycle state (orderStatus).
- **Product:** Represents the inventory. Includes standard commerce attributes (price, stock, specs) and supports the specialized auction flags.
- **Payment:** An audit log for financial transactions. Stores the razorpayOrderId, cryptographic signatures, and payment status (captured, failed, refunded) to ensure financial integrity.
- **Return:** Represents a post-purchase dispute. It links back to a specific Order and Seller, tracking the reason, status, and calculated refund amount.
- **Cart:** (Embedded in User) A persistent holding area for products before purchase, tracking quantities and timestamps.

6.2 Control Objects (Controllers)

- **OrderController (Core):**
  - **Responsibilities:** The central engine of the marketplace.
    - **Dual-Mode Ordering:** Handles logic for both "Buy Now" (single item) and "Checkout Cart" (bulk items) flows.
    - **Pricing Engine:** Dynamically calculates totals including Tax (18% GST), Shipping logic (Free > ₹1000), and Coupon discounts.
    - **Inventory Locking:** Verifies and reserves stock before order creation to prevent overselling.

        - **Lifecycle Management:** Handles status transitions from "Payment Pending" to "Delivered" or "Cancelled".

- **PaymentController (Core):**
    - **Responsibilities:** Ensures secure financial processing.
        - **Security:** Generates and verifies Razorpay cryptographic signatures (HMAC-SHA256) to prevent transaction tampering.
        - **Idempotency:** Handles network race conditions to ensure a user isn't charged twice or stock isn't deducted multiple times for the same request.
        - **Refunds:** Orchestrates automated refunds for cancellations and returns.
- **ReturnController (Core):**
    - **Responsibilities:** Manages the complex reverse-logistics of a multi-seller marketplace.
        - **Split-Return Logic:** If an order contains items from Seller A and Seller B, this controller intelligently splits the return request so each seller manages only their own items.
        - **Refund Calculation:** Automatically calculates the precise refund amount, including proportional tax and shipping reversals.
- **CartController:**
    - **Responsibilities:** Manages the buyer's shopping session, enforcing stock limits when items are added and synchronizing product details (price/image) in real-time.
- **ProductController:**
    - **Responsibilities:** Manages catalog operations, including image upload handling (Cloudinary) and categorization.
- **AuctionController:**
    - **Responsibilities:** Manages the bidding timer and status transitions (Active/Completed) for auction-type products.
- **AssistantController:**
    - **Responsibilities:** Interacts with the AI microservice to provide customer support and product recommendations.

6.3 Object Behaviours

This section lists the primary methods and operations associated with each domain object, defining their functional capabilities.

6.3.1 User Domain Behaviours

- **User:**
  - register(): Creates a new user record with a secure, hashed password.
  - login(): Authenticates user credentials and issues a JWT session token.
  - comparePassword(): Validates an entered password against the stored hash during login.
  - updateProfile(): Modifies user attributes like name, phone number, or shop details.
  - upgradeToSeller(): Adds necessary seller information (Shop Name, Address) to a buyer account.

6.3.2 Marketplace Domain Behaviors

- **Product:**
  - createProduct(): Initializes a new product listing with images, price, and specifications.
  - updateStock(): Decrements inventory count automatically after a verified order placement.
  - checkAvailability(): Verifies if the requested quantity is available in stock before adding to cart.
- **Review:**
  - addReview(): Links a user's rating and comment to a specific product.
  - calculateAverageRating(): Triggers an aggregation update to refresh the product’s overall rating score.

6.3.3 Transaction Domain Behaviors

- **Cart:**
  - addToCart(): Adds a new item or increments the quantity of an existing item.
  - calculateTotal(): Computes the subtotal of all items currently in the cart.
  - clearCart(): Removes all items from the cart instance after a successful checkout.
- **Order:**
  - createOrder(): Generates a persistent order record including tax, shipping, and final total.

  - updateStatus(): Transitions the order lifecycle (e.g.,

    "Payment Pending" → "Placed" → "Shipped").
  - cancelOrder(): Marks the order as cancelled and triggers the refund and stock restoration workflows.
- **Coupon:**
  - isValid(): Checks if the coupon code is active, within date limits, and meets minimum order value requirements.
  - calculateDiscount(): Returns the exact monetary value to be deducted from the order total.

6.3.4 Auction Domain Behaviors

- **Auction (Socket Wrapper):**
  - joinRoom(): Connects a user's WebSocket session to a specific product's auction channel.
  - placeBid(): Validates and accepts a new bid if it is higher than the current bid.
  - endAuction(): Finalizes the auction upon timer expiry and triggers order creation for the highest bidder.

6.4 Object Relationships

This section defines the cardinality and associations between the system's primary objects.

- **User ←→ Order (One-to-Many):**
  - A **User** (Buyer) can place multiple **Orders** over time.
  - Each **Order** is linked to exactly one Buyer.
- **User ← → Product (One-to-Many):**
  - A **User** (Seller) can list multiple **Products** in the marketplace.
  - Each **Product** is owned by exactly one Seller.
- **Product ← →Review (One-to-Many):**
  - A single **Product** can have multiple **Reviews** written by different users.
  - Each **Review** is linked to exactly one Product.
- **Order ← →Payment (One-to-One):**
  - Each **Order** has exactly one associated **Payment** record (representing the Razorpay transaction).
  - A **Payment** record is strictly linked to a unique Order ID to ensure auditability.

- **Product ← →Bid (One-to-Many):**
  - An Auction **Product** maintains a history of multiple **Bids**.
  - Each **Bid** is associated with one Product and one User.
- **Order ← →Return (One-to-Many):**
  - A single **Order** can generate multiple **Return** requests (e.g., if the order contains items from multiple different sellers).
  - Each **Return** request links back to the specific Order and the specific Seller responsible for the item.

6.5 Object Interaction Scenarios

This section describes the sequence of interactions between objects to fulfill key user requirements.

6.5.1 Scenario: Purchasing a Product (Checkout)

1. **User** triggers addToCart() to populate the **Cart**.
2. **User** initiates checkout; **OrderController** calls checkAvailability() on the **Product**.
3. **CouponController** (optional) runs isValid() and calculateDiscount().
4. **Order** object is instantiated with status "Payment Pending".
5. **PaymentController** creates a **Payment** record and initiates the Razorpay gateway.
6. Upon successful signature verification, **Order** calls updateStatus("Placed").
7. **Product** calls updateStock() to decrement inventory.

6.5.2 Scenario: Placing a Bid in Real-Time

1. **User** establishes a socket connection and calls joinRoom() for a specific **Product**.
2. **User** triggers the placeBid event with an amount.
3. **SocketServer** validates: Bid Amount > Product.currentBid AND Product.auctionStatus == Active.
4. **Bid** object is created and saved to the database.
5. **Product** object updates its currentBid and highestBidder attributes.
6. **SocketServer** broadcasts the bidUpdate event to all other **Users** in the room.

6.5.3 Scenario: Returning a Product

1. **User** (Buyer) calls requestReturn() for specific items within an **Order**.
2. **ReturnController** iterates through items and groups them by **Seller**.
3. A separate **Return** object is created for each Seller involved in the request.

4. **Seller** reviews and calls approveReturn().
5. **PaymentController** triggers the processRefund logic via Razorpay.
6. **Product** stock is incremented, and **Order** status is updated to "Returned" or "Partially Returned".

6.6 Important Algorithms

1. Atomic Bidding Algorithm:

- **Goal:** Prevent race conditions where two users bid the same amount simultaneously.
- **Logic: -**

Input: productId, bidAmount, userId

Query: Find Product WHERE _id = productId AND status = 'Active' AND currentBid < bidAmount

Update: SET currentBid = bidAmount, PUSH bidHistory

Result: If document updated -> Success; Else -> Fail (Bid too low or Auction ended).

2. Order Pricing & Validation Algorithm

- **Goal:** Ensure financial accuracy and stock integrity during checkout.
- **Logic:**

Input: User Cart, CouponCode

1. Validation: Loop through Cart Items -> Check if Product.stock >= Cart.quantity.

2. Pricing: Calculate Base Price = Sum(Item.price * quantity).

3. Discount: If CouponCode valid -> Deduct Discount Amount (Validator: MinOrderValue, Expiry).

4. Tax: Calculate GST (18%) on discounted price.

5. Shipping: If Price < 1000 -> Add ₹50; Else -> ₹0.

6. Total: Base - Discount + Tax + Shipping.

7. Result: Create "Payment Pending" Order and return Payment Gateway initiation data.

2. Multi-Seller Return Splitting Algorithm

- **Goal:** Allow users to return an entire mixed order while ensuring sellers only process their own items.
- **Logic:**

Input: OrderID, List of Items to Return

1. Fetch Order and grouping: Group requested items by their 'sellerId'.

2. Iterate through each Seller Group:

a. Calculate 'SellerTotal' = Sum(Item.price * quantity).

b. Calculate 'ProportionalTax' = (SellerTotal / OrderTotal) * OrderTax.

c. Calculate 'ProportionalShipping' = (SellerTotal / OrderTotal) * OrderShipping.

d. Create separate Return Request for this Seller with calculated RefundAmount.

3. Update Order Status to "Returned".

3. Refund & Stock Restoration Algorithm:

- **Goal:** Ensure financial and inventory consistency during cancellations/returns.
- **Logic: -**

Input: orderId, refundAmount

Action 1: Call Razorpay API to process refund.

Action 2: If Success -> Update Payment status to 'refunded'.

Action 3: Loop through OrderItems -> Increment Product.stock by quantity.

Action 4: Update Order status to 'Cancelled' or 'Returned'.

4. Payment Verification & Idempotency Algorithm

- **Goal:** Prevent double-spending and ensure stock is deducted exactly once.
- **Logic:**

Input: RazorpayPaymentId, OrderId, Signature

1. Security Check: Generate local HMAC-SHA256 signature and match with Input Signature.

2. If Invalid -> Mark Payment 'Failed' & Return Error.

3. Idempotency Check: Fetch Order. If status is already "Order Placed" -> Return Success (Do nothing else).

4. Success Flow:

a. Update Payment status to 'Captured'.

b. Update Order status to 'Order Placed'.

c. Deduct Stock: For each item in Order -> Product.stock = Product.stock - quantity.

7. Dynamic Modeling (Diagram Reference)

[Kindly refer to the attached Sequence Diagrams and Class Diagrams submitted alongside for visual representation of:

Prepared by: -
Hriday Asnani (202301282)