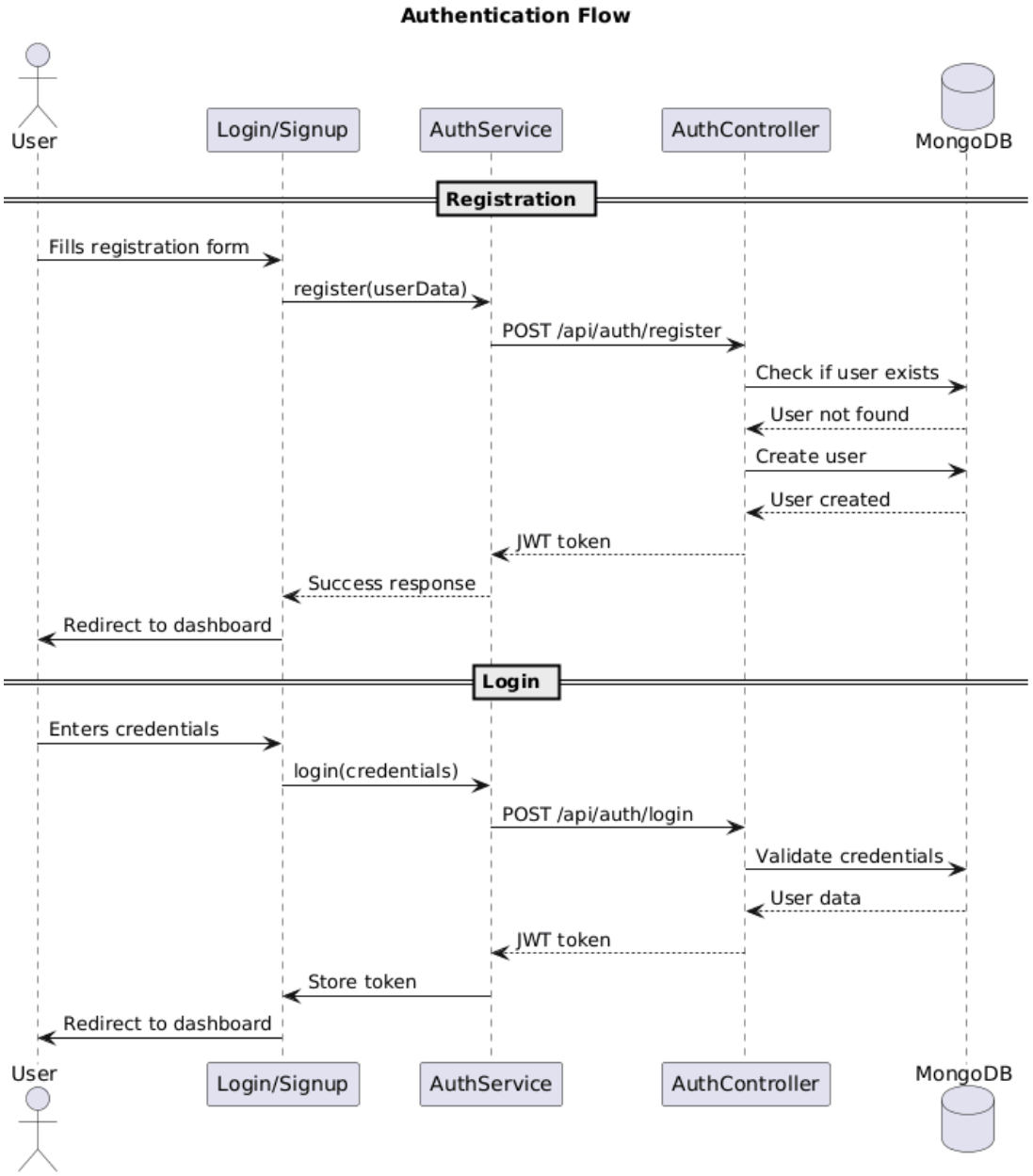
Market_Connect

System Design

1.1 Authentication Flow



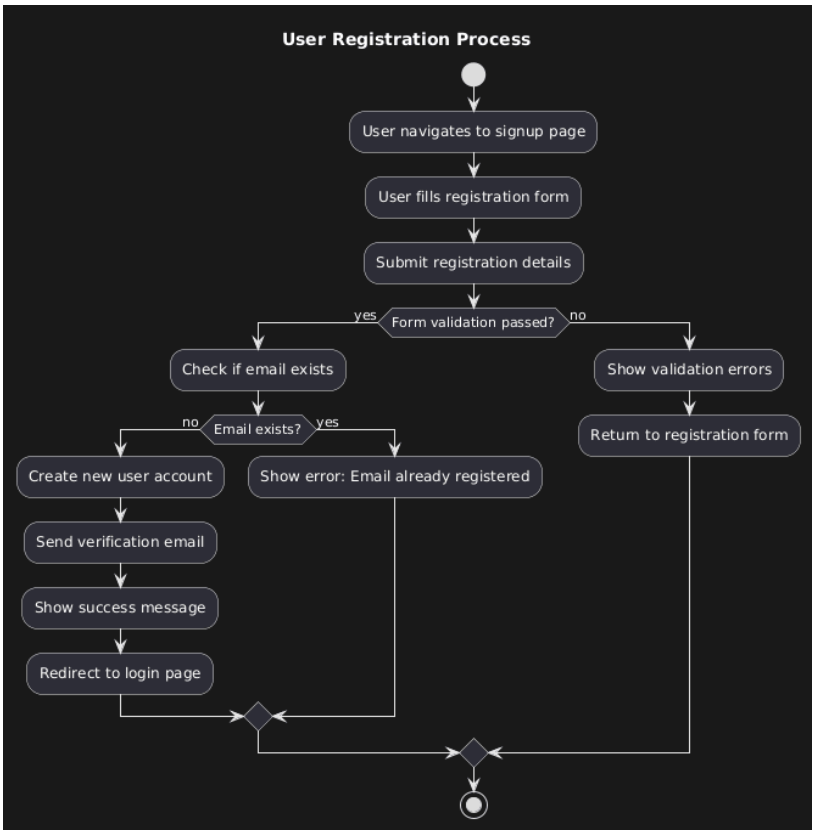
Authentication Strategy: The system uses a dual-authentication mechanism:

Local Auth: Users register via email/password. Passwords are hashed using bcrypt before storage. Upon login, a JWT (JSON Web

Token) is issued, which must be included in the Authorization header for protected routes.

OAuth 2.0: Users can sign in via Google. The Passport.js strategy handles the handshake with Google servers, creating a local user record if one doesn't exist.

1.2 Admin Flow



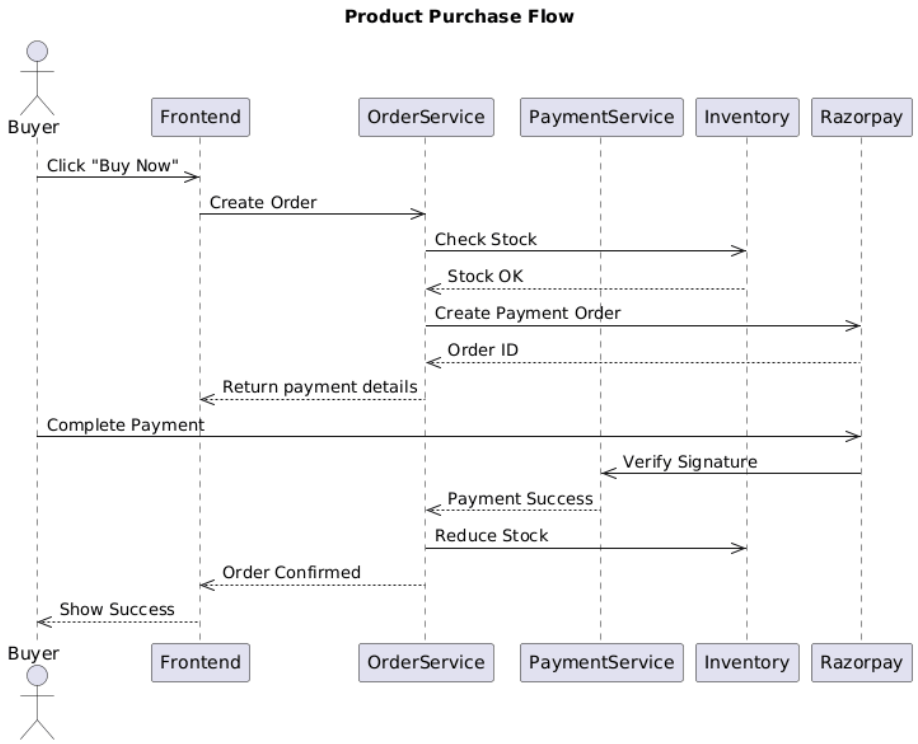
1.3 Product Purchase Flow

Purchase Lifecycle: The purchase flow is designed to ensure strict consistency between Inventory and Payments:

Inventory Locking: When an order is initiated, the OrderController first verifies stock availability.

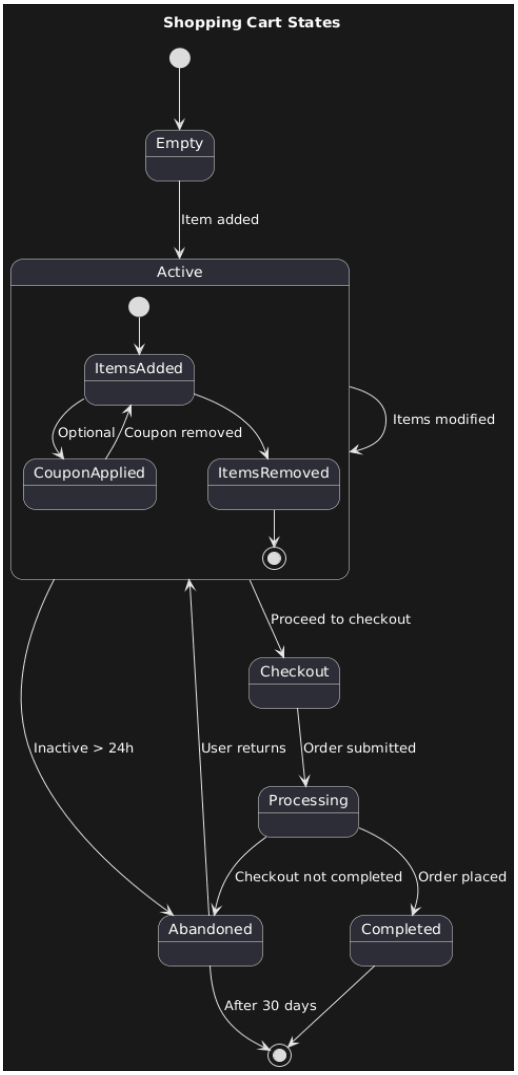
Payment Initiation: A generic Razorpay order is created. Crucially, after the user pays, the backend verifies the HMAC-SHA256

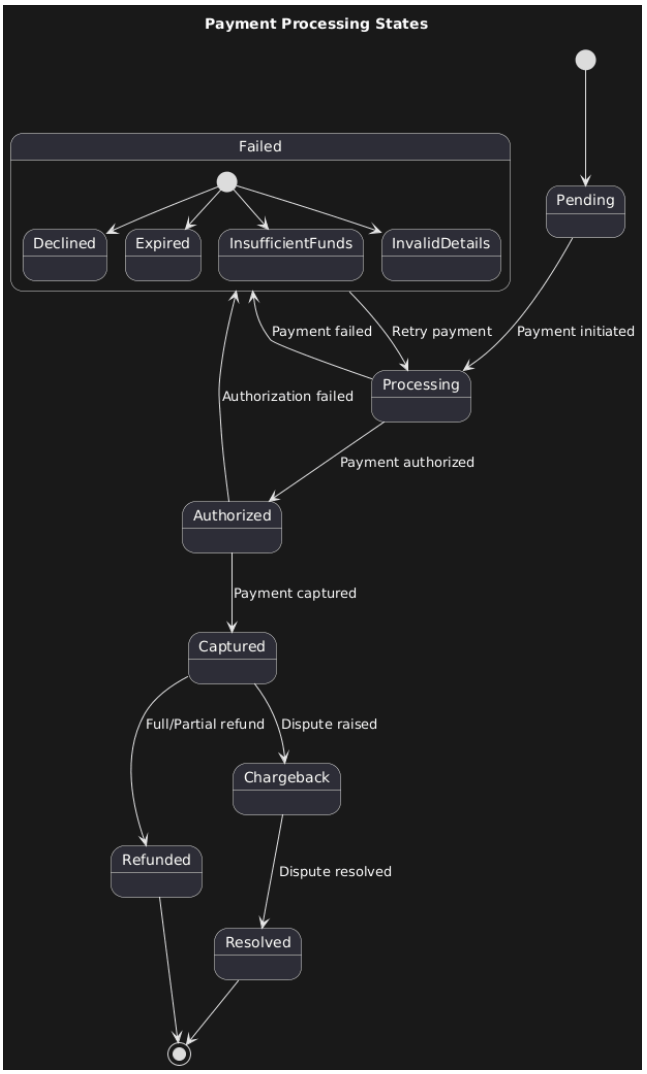
signature returned by Razorpay. Only if this matches is the order marked as "Placed" and stock permanently deducted. This prevents client-side payment tampering.

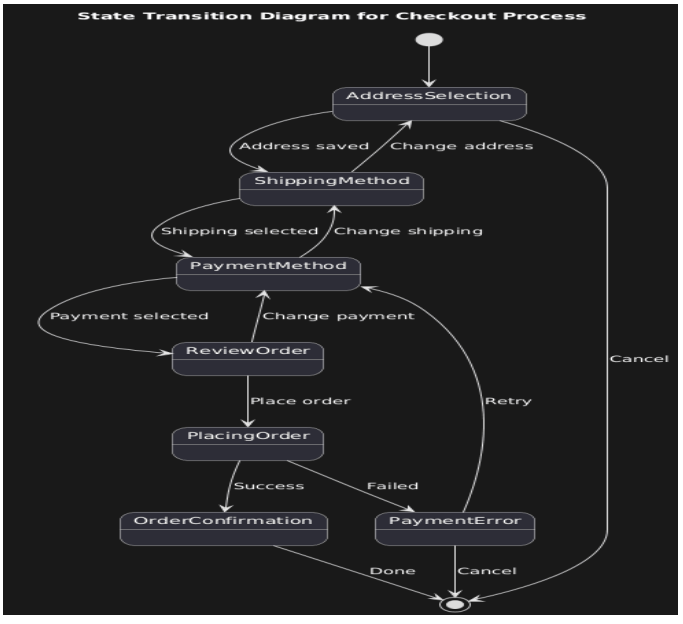


1.3 Shopping Cart

1.4 Payment

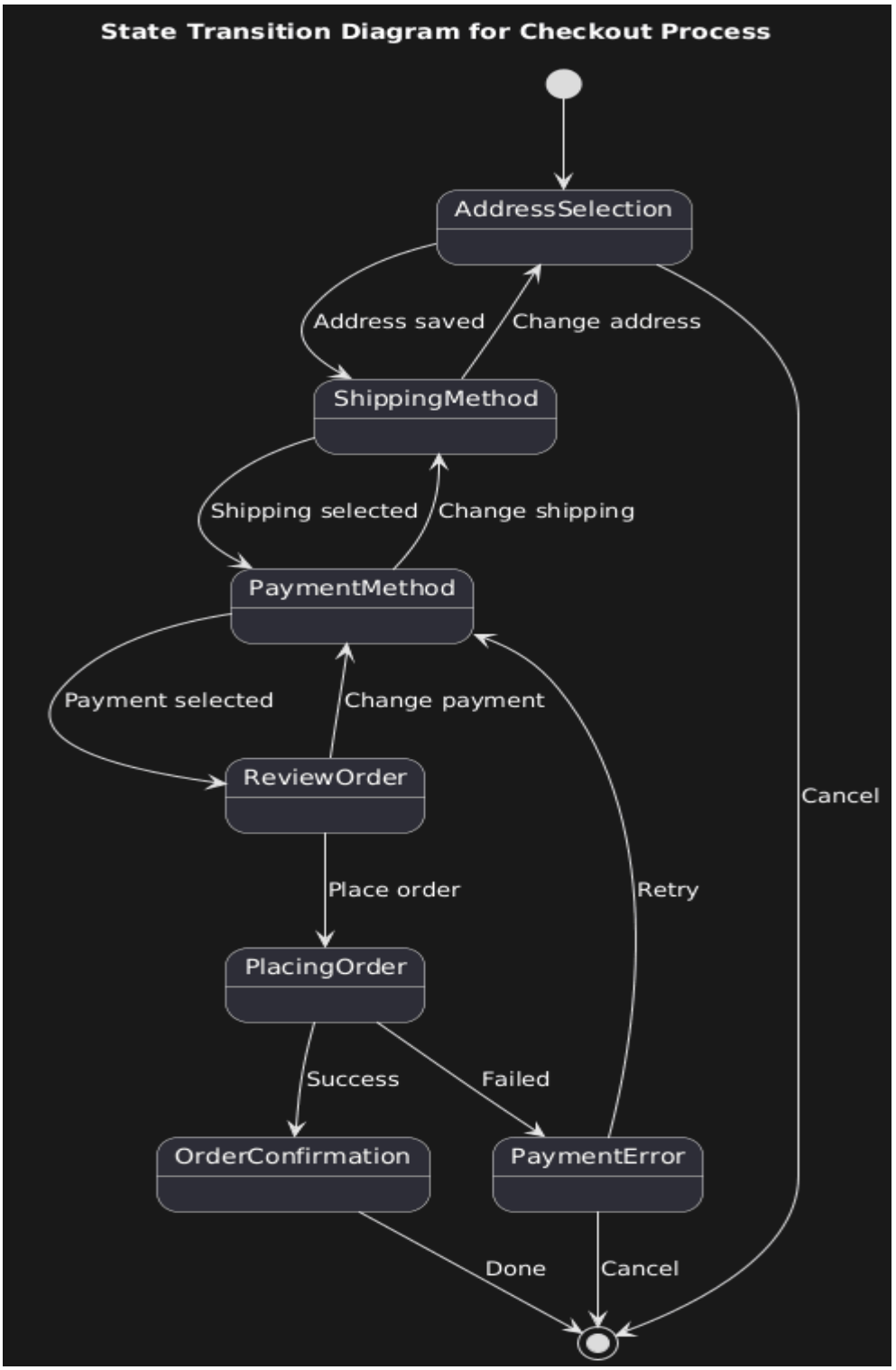




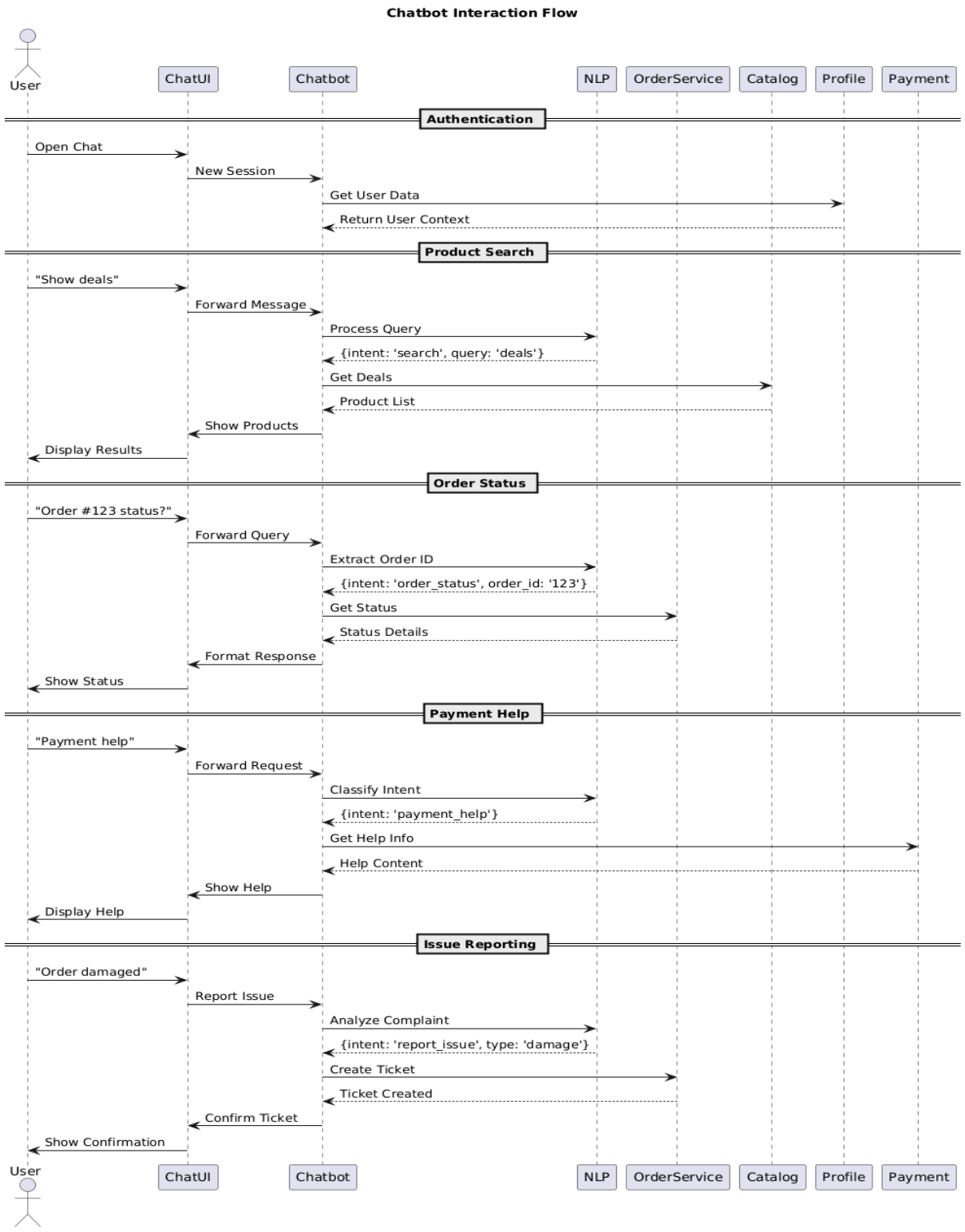


1.5 Search flow

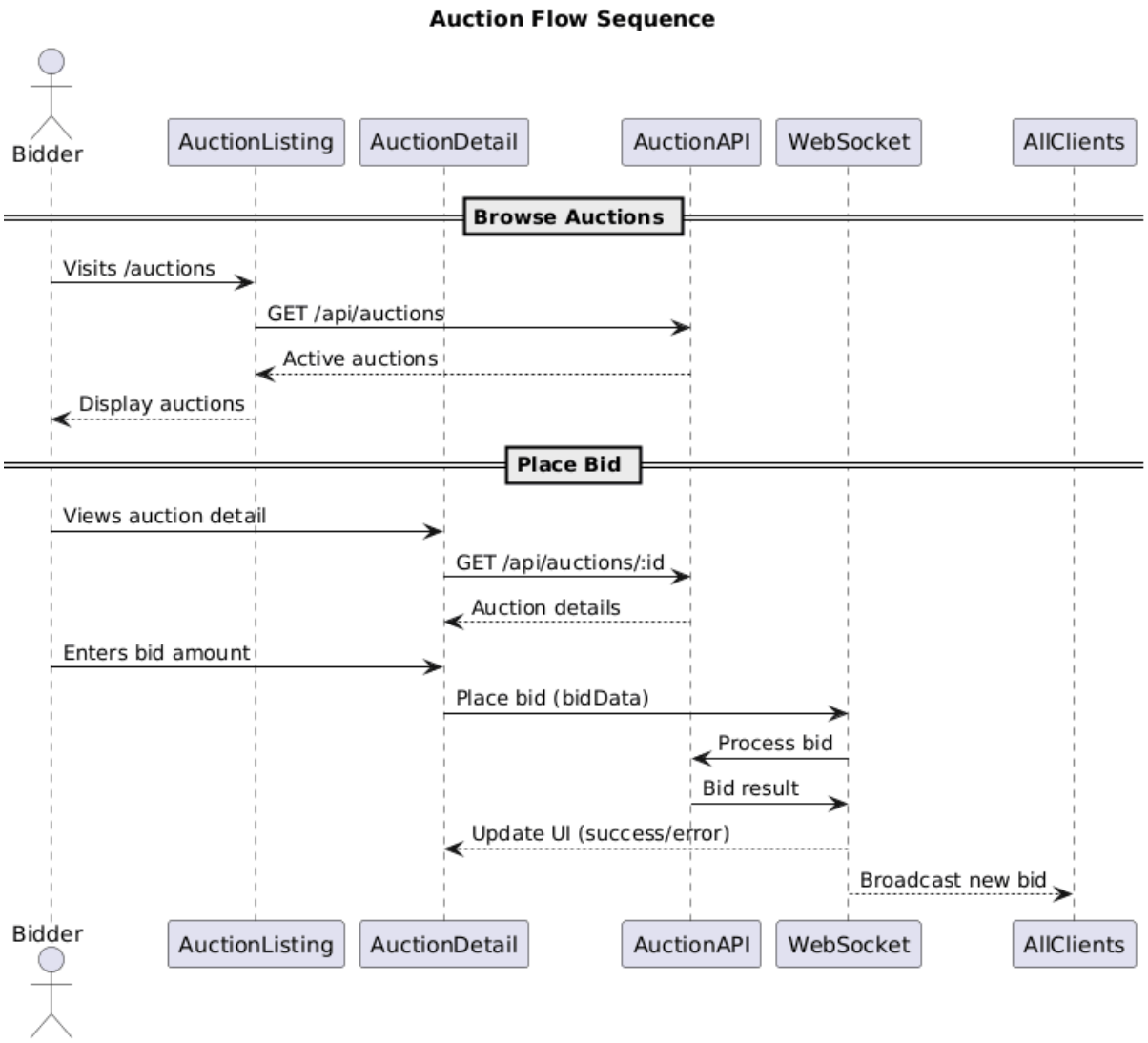
1.6 checkout flow

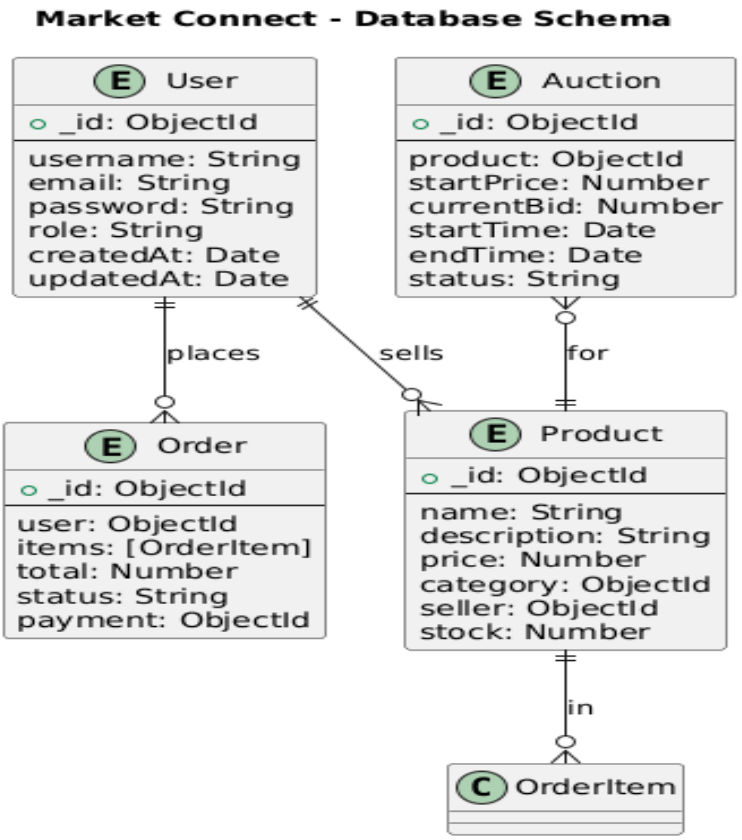


1.7 Chatbot


Chatbot Interaction Flow
User
ChatUI
Chatbot
NLP
OrderService
Catalog
Profile
Payment
Authentication
Open Chat
New Session
Get User Data
Return User Context
Product Search
"Show deals"
Forward Message
Process Query
{intent: 'search', query: 'deals'}
Get Deals
Product List
Show Products
Display Results
Order Status
"Order #123 status?"
Forward Query
Extract Order ID
{intent: 'order_status', order_id: '123'}
Get Status
Status Details
Format Response
Show Status
Payment Help
"Payment help"
Forward Request
Classify Intent
{intent: 'payment_help'}
Get Help Info
Help Content
Show Help
Display Help
Issue Reporting
"Order damaged"
Report Issue
Analyze Complaint
{intent: 'report_issue', type: 'damage'}
Create Ticket
Ticket Created
Confirm Ticket
Show Confirmation
User
ChatUI
Chatbot
NLP
OrderService
Catalog
Profile
Payment

1.8 Auction flow





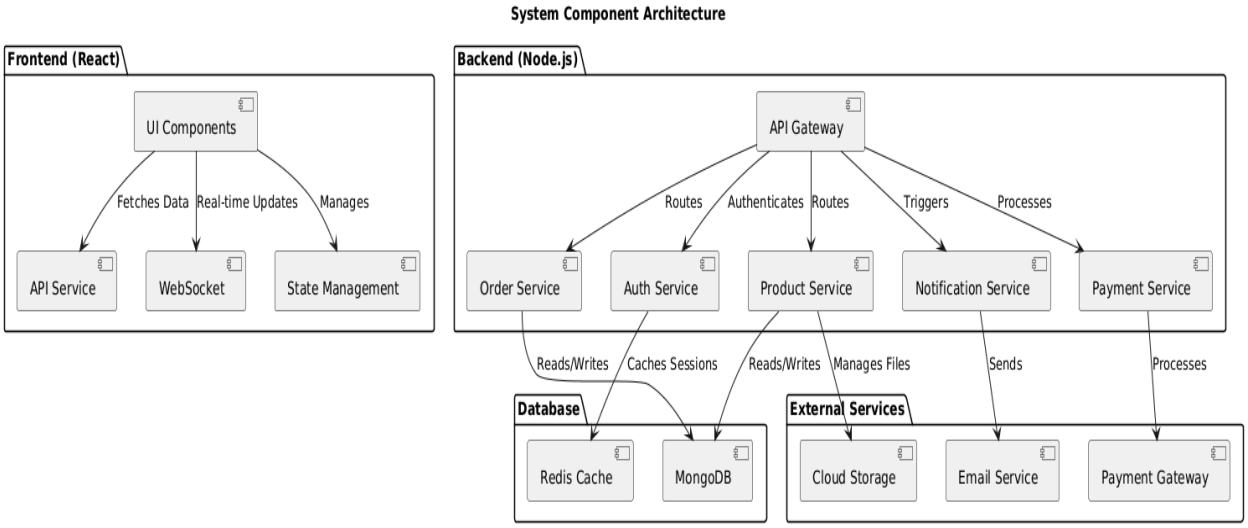
2..Database Schema

Data Model: We utilize MongoDB for its flexible schema design, which is essential for our diverse Product catalog where different categories (Electronics vs. Fashion) require different specification fields.

Auction Handling: Auction details (start time, current bid) are embedded directly within the Product document to allow for atomic updates and high-performance reads during live bidding.

Financials: Orders and Payments are stored with normalized references to Users to maintain an audit trail.

3.system components



Architecture Description: The system follows a Hybrid Monolithic-Microservice Architecture.

Core Backend (Node.js): Acts as the primary API Gateway and handles business logic for Users, Orders, and Auctions. It is structured using a Layered (MVC) pattern where Controllers (OrderService, AuthService) interact with the Database Layer.

AI Microservice (Python): A separate Flask service handles the AI Chatbot. It communicates with the Core Backend via REST APIs to fetch product context and generate intelligent responses using the Groq LLM.

Real-Time Engine: A Socket.io server runs alongside the Node.js backend to manage live Auction Rooms and bid updates.

4.Middleware

✅ Security Middleware

Middleware	Purpose	Implementation	Configuration
`cors`	Handles CORS rules	Allowed origins list	Environment-based
`helmet`	Secures HTTP headers	Security headers	Production only
`rateLimit`	Throttles excessive requests	Redis / Memory store	Configurable limits
`xssClean`	Prevents XSS attacks	Input sanitization	Applied on all requests
`sanitizeInput`	Removes harmful input	Mongo injection & XSS filter	Global middleware

Security middleware protects the application by enforcing safe request handling, preventing attacks like XSS, and controlling access through secure headers and rate limiting.

✅ File Handling Middleware

Middleware	Purpose	File Types	Max Size
`uploadProductImage`	Upload product images	jpg, png, webp	5 MB
`uploadAvatar`	Upload user profile images	jpg, png	2 MB
`validateFile`	Validate file format & size	Type & dimension checks	Custom rules
`cloudinaryUpload`	Upload to Cloudinary	All supported formats	Depends on plan

✅ Error Handling Layer

Middleware	Purpose	Handles	Response Format
`errorHandler`	Global exception handler	All errors	Standardized JSON
`notFound`	Handle 404 routes	Invalid endpoints	JSON `{error: ...}`
`validationError`	Format validation issues	Zod/Joi errors	Structured validation msg
`asyncWrapper`	Handles async errors	Promise rejections	Passes error to handler

The error handling layer captures system and validation errors and returns consistent, structured JSON responses for debugging and stability.