E-Commerce Platform - Complete Documentation

Project Overview

This is a full-stack e-commerce application for selling clothing. The platform features user authentication, product management, a shopping cart system with a loyalty points program, and integrated Stripe payment processing.

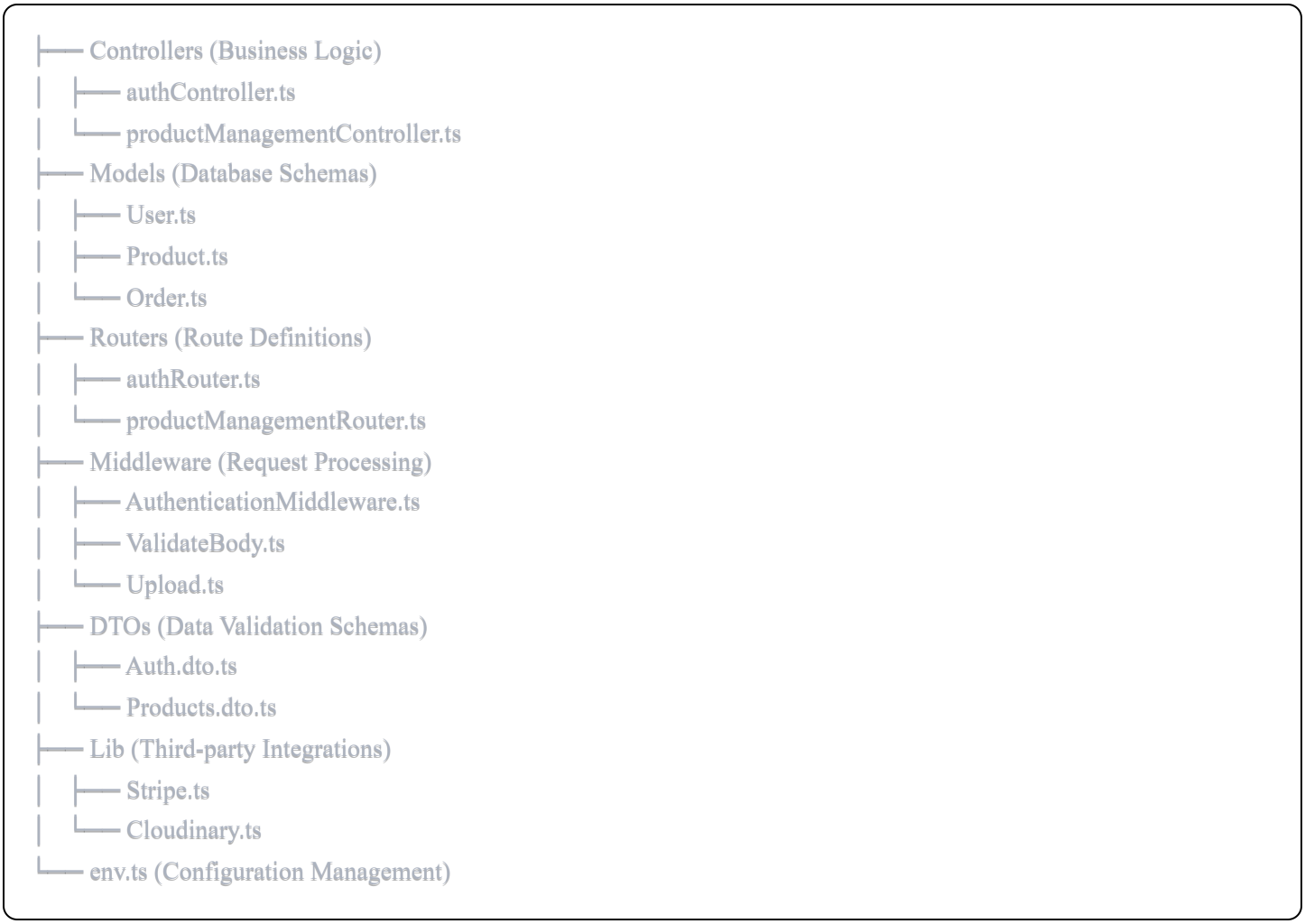
Tech Stack:

- **Frontend:** React with TypeScript
- **Backend:** Node.js with Express and TypeScript
- **Database:** MongoDB
- **Image Storage:** Cloudinary
- **Authentication:** JWT tokens and Google OAuth2
- **Payments:** Stripe (test mode)
- **Input Validation:** Zod
- **File Upload:** Multer

Architecture & Component Structure

Backend Architecture

The backend follows a layered architecture pattern:



Data Models

User Model: Stores user credentials, loyalty points, Stripe customer ID, and Google OAuth subscription ID. Supports both password-based and Google OAuth authentication.

Product Model: Contains product details including name, description, price, image URL from Cloudinary, and current stock quantity.

Order Model: Records transaction details including products ordered, total price, payment session ID, order status (pending/canceled/completed), loyalty points used, and timestamps.

State Management Flow

Authentication State

- Users authenticate via login/register or Google OAuth
- JWT tokens are stored in secure HTTP-only cookies
- `AuthenticationMiddleware` validates tokens on protected routes
- Admin status is checked for administrative endpoints

Cart & Checkout State

- Cart items are managed on the frontend before submission
- Backend validates cart items against current inventory
- Loyalty points are calculated and applied during checkout
- Order records are created in pending state before payment confirmation

Order Processing State

- Orders transition through states: pending → completed/canceled
- Stripe webhooks trigger state transitions
- MongoDB transactions ensure atomic stock updates
- User loyalty points are updated after successful payment

API Endpoints

Authentication Routes (`/api/auth`)

POST /login

- Body: `{ email: string, password: string }`
- Authenticates user and returns JWT token in cookie
- Validates credentials against hashed password

POST /register

- Body: `{ email: string, password: string, username: string }`
- Creates new user account with hashed password
- Initializes Stripe customer for the user
- Returns JWT token

POST /google

- Body: `{ code: string }`
- Handles Google OAuth flow using authorization code
- Creates new user or logs in existing user
- Stores Google subscription ID for account linking

GET /me

- Protected endpoint
- Returns current authenticated user's information
- Data is sanitized to exclude sensitive fields

POST /logout

- Clears JWT token cookie

POST /change-password

- Protected endpoint
- Body: `{ oldPassword: string, newPassword: string }`
- Requires old password verification
- Hashes and stores new password

Product Management Routes (`/api/products`)

POST /add

- Protected (admin only)
- Multipart form with image file
- Creates product with Cloudinary image upload
- Returns created product object

PUT /edit

- Protected (admin only)
- Updates product details and optionally replaces image
- Body includes `productId` and updated fields

DELETE /delete

- Protected (admin only)
- Removes product from inventory

POST /checkout

- Protected endpoint
- Body: `{ cart: Array<{productId, quantity}>, points: number }`

- Validates cart items and stock availability
- Creates Stripe checkout session
- Records order in pending status

GET /all

- Public endpoint
- Returns all available products

GET /orders

- Protected endpoint
- Returns all orders for authenticated user

POST /stripe/webhook

- Handles Stripe webhook events
- Processes checkout completion and expiration

Key Features & Implementation Details

1. Dual Authentication System

The application supports two authentication methods:

Traditional Authentication: Users create accounts with email and password. Passwords are hashed using bcrypt with a salt round of 10 before storage. This ensures passwords remain secure even if the database is compromised.

Google OAuth2: Users can sign in using Google accounts without creating a separate password. The Google authorization code is exchanged for an ID token, which is verified against Google's servers. The Google subscription ID is stored to link future sign-ins to the same account.

2. Loyalty Points System

Users earn loyalty points equal to 10× their purchase amount in dollars (e.g., a $50 purchase = 500 points). Each point represents $0.10 in discounts on future purchases.

Point Application Process:

- Users can specify points to redeem during checkout
- Backend validates that requested points don't exceed available balance

- Points are converted to a Stripe coupon for discount application
- After successful payment, points are deducted from user account
- New points are added equal to the order total

Validation Logic: The backend ensures points used are the minimum of (1) requested points, (2) available loyalty points, and (3) order total × 10. This prevents users from overspending points or using points across multiple simultaneous orders.

3. Inventory Management with Stripe Integration

Stock management is tightly integrated with Stripe payment processing to prevent overselling.

Checkout Process:

- Frontend validates cart against current inventory
- Backend re-validates inventory before creating Stripe session
- Order record is created in "pending" status
- User is directed to Stripe checkout

Payment Confirmation:

- Stripe webhook notifies backend of successful payment
- Backend retrieves line items from Stripe session
- Inventory stock is decremented using MongoDB transactions
- Order status transitions to "completed"

4. Concurrent Order & Race Condition Handling

Challenge: Multiple Orders with Insufficient Stock

When two orders are placed and there isn't enough inventory for both, the second order must be rejected while the first succeeds.

Solution Implemented:

- MongoDB transactions ensure atomic stock operations
- The stock update query uses a condition: `stock: { $gte: quantity }`
- Only if the condition is met does the stock decrement
- If `modifiedCount === 0`, the condition wasn't satisfied
- Transaction is rolled back and payment is refunded

- Customer receives an order cancellation message explaining the reason

This guarantees that once stock is decremented, it's immediately unavailable to other orders, preventing overselling even with simultaneous checkouts.

5. Loyalty Points Double-Spending Prevention

Challenge: User Attempts to Use Same Points in Multiple Orders

Users could theoretically submit multiple checkout requests with the same loyalty points before the first order completes, potentially using points twice.

Solution Implemented:

- Points are only deducted from the user account during webhook processing, not during checkout request
- During webhook handling, the backend checks current user balance: `if (user.loyaltyPoints < points)`
- If current balance is insufficient (points were already used), the entire order is refunded
- Customer is alerted that the order was canceled due to insufficient loyalty points
- The delay between checkout and webhook completion naturally prevents double-spending

6. Input Validation & Injection Prevention

Challenge: User Sends Malicious Cart Data from Frontend

Since frontend code can be modified by users, the backend cannot trust cart data directly.

Solution Implemented:

- Zod schemas validate all incoming data before processing
- Cart items are validated with `ProductInCart` schema requiring valid `productId` and positive `quantity`
- Product IDs are verified as valid MongoDB ObjectIds using `isObjectIdOrHexString()`
- Frontend cart is completely rebuilt from database queries, not from user-submitted data
- Cart sent by user is only used to determine what to query; actual product details come from fresh database lookups
- Stock, price, and product existence are re-verified at checkout time

This multi-layer validation ensures that even if a user submits a malicious cart, the backend uses only verified information from the database.

Security Implementation

Authentication Security

JWT tokens are stored in HTTP-only cookies, preventing access from JavaScript and protecting against XSS attacks. The `secure` flag ensures cookies are only sent over HTTPS in production. The `sameSite` policy is set to "none" in production (with secure flag) and "lax" in development to prevent CSRF attacks.

Admin status is checked on protected routes to ensure only administrators can manage products. User IDs extracted from tokens are validated as proper MongoDB ObjectIds before database queries.

Password Security

Passwords are hashed with bcrypt using a salt round of 10, which provides strong protection against brute-force attacks. When users change passwords, the old password is verified before accepting the new one.

API Security

All protected endpoints require valid JWT authentication. Input validation occurs on every endpoint using Zod schemas before business logic executes. CORS is configured to only accept requests from trusted domains (localhost and production frontend URL).

Data Exposure Prevention

User objects are filtered through the `SafeUser` schema before being sent to frontend, excluding sensitive fields like hashed passwords and Stripe IDs. Order data is only accessible to the user who created it or to admins.

Payment Processing with Stripe

Checkout Session Creation

When a user initiates checkout, the backend creates a Stripe checkout session with line items representing cart products. The session includes customer ID for linking to the user, metadata with user ID and loyalty points used, and success/cancel URLs. If loyalty points are applied, a Stripe coupon is created with the discount amount and added to the session.

Webhook Event Processing

Stripe webhooks notify the backend of payment events. For `checkout.session.completed` events, the backend retrieves the session, verifies the user exists, and checks if the order was already processed to prevent duplicate handling.

The webhook handler then verifies that stock is still available for all items using MongoDB transactions. If any item has insufficient stock at the time of webhook processing, the payment is refunded and the order is canceled. This catches race conditions where inventory was sold out between checkout and payment completion.

If all checks pass, product stock is decremented, loyalty points are adjusted (deducted if used, new points added from purchase total), and the order status is set to "completed".

For `checkout.session.expired` events, pending orders are automatically canceled with a message indicating the checkout expired.

File Upload & Image Management

Product images are uploaded via Multer, which stores files in memory. The backend then uploads images to Cloudinary using the streaming upload API. Cloudinary handles image optimization, CDN distribution, and persistent storage. Only the secure URL returned by Cloudinary is stored in the database, keeping the database lightweight.

Error Handling & Edge Cases

The application handles various error scenarios:

Validation Errors: Return 400 status with descriptive error messages. Frontend can display specific issues.

Authentication Errors: Return 401 status when tokens are invalid, expired, or missing. Middleware prevents unauthorized access to protected endpoints.

Not Found Errors: Return 404 status when referenced products or orders don't exist in database.

Concurrent Payment Issues: Return 400 status when checkout fails due to insufficient stock. Includes which items have problems so user can adjust cart.

Stripe Errors: Webhook failures log errors and return 500 status. Stripe will retry webhook delivery.

Database Errors: Transaction rollbacks ensure partial updates don't corrupt data. Refunds are issued if transaction cannot complete.

Environment Configuration

Environment variables are validated at startup using Zod schemas. Required variables include database connection URI, JWT secret (minimum 30 characters for security), Stripe keys, Google OAuth credentials, Cloudinary credentials, frontend URL, and Node environment. The application exits immediately if any required variable is missing or invalid, preventing runtime errors from misconfiguration.

Development Best Practices Implemented

Type Safety: Full TypeScript usage provides compile-time type checking and IDE autocompletion, reducing

runtime errors.

Request Type Definitions: Express Request types include custom user property through TypeScript declaration merging, enabling `req.user` with full type safety.

Schema-Driven Validation: Zod schemas serve as both validation logic and TypeScript type sources via `z.infer`, ensuring types and validation always match.

Middleware Composition: Authentication middleware accepts a parameter to conditionally enforce admin checks, reducing code duplication.

Error Logging: Console logging throughout the application aids debugging in development and production monitoring.

Atomic Transactions: MongoDB sessions ensure multi-step operations like stock updates either complete fully or roll back entirely, preventing data inconsistency.

Frontend Architecture

The frontend is built with React and TypeScript, using Redux for state management and React Router for navigation. The application follows a modular component structure with centralized state management and persistent storage.

Technology Stack

React & TypeScript: Provides type safety and component-based architecture. React hooks manage component lifecycle and side effects.

Redux & Redux Toolkit: Centralizes application state including authentication, cart, wishlist, products, orders, and search. Toolkit simplifies reducer creation with slices.

React Router: Manages client-side routing with protected routes for authenticated users only.

Google OAuth: Enables Google authentication through `@react-oauth/google` library.

Axios API Client: Handles all backend communication with automatic cookie inclusion for JWT tokens.

Local Storage: Persists cart and wishlist data across browser sessions with type-safe parsing.

Application Initialization Flow

When the application loads, it executes three parallel initialization tasks:

1. Cart & Wishlist Hydration: The app retrieves saved cart items and wishlist from local storage. The `safeParse` utility safely deserializes JSON with fallback to empty arrays if parsing fails or stored data is invalid.

2. User Authentication Check: An API request is made to `/auth/me` to verify the current user session. If successful, user data is dispatched to Redux auth slice. If unauthorized, the user is marked as not authenticated

and local storage is cleared as a security measure.

3. Products & Orders Fetch: Products are fetched from `/products/all` and dispatched to the products slice. Orders are fetched from `/products/orders` if the user is authenticated. These requests happen simultaneously with the auth check, improving perceived performance.

If either products fetch or auth check fail, appropriate error states are displayed. Loading indicators prevent UI interactions during initialization.

Routing Structure

The application uses a nested routing structure with a main layout component wrapping all routes.

Public Routes:

- `/` - Landing page with initial marketing content
- `/products` - Browse all products with filtering and search
- `/product/:id` - Individual product details page
- `/login` - User login with password or Google OAuth
- `/register` - New user account creation

Protected Routes (require authentication):

- `/wishlist` - View saved favorite products
- `/cart` - View shopping cart and initiate checkout
- `/profile` - User account and order history

Protected routes render a `ProtectedRoute` wrapper component that checks authentication status. If user is not authenticated, they're redirected to login.

State Management Slices

Auth Slice

Manages user authentication state with two properties: the current user object (null when not authenticated) and a loading flag indicating whether auth status is being determined.

Reducers:

- `auth`: Sets user data and marks loading complete, triggered when `/auth/me` returns successfully
- `notAuth`: Clears user data and marks loading complete, triggered when authentication check fails

Usage: Protected route checks `user` property. App loading screen displays while `loading` is true.

Cart Slice

Stores array of cart entries, each containing a product ID and quantity. Automatically synchronizes with local storage on every change.

Reducers:

- `addProduct`: Adds item to cart or updates quantity if already present
- `removeProduct`: Removes item entirely from cart
- `incQuantity`: Increments quantity for existing cart item
- `decQuantity`: Decrements quantity, removing item if quantity reaches zero
- `clearCart`: Empties entire cart after successful checkout
- `setCart`: Initializes cart from saved state with type validation

Validation: The `setCart` reducer includes safety checks ensuring each item has a string product ID, numeric quantity, and quantity greater than zero. This prevents corrupted local storage from breaking the app.

Persistence: Every reducer action updates local storage immediately, ensuring cart persists across browser sessions and page reloads.

Favorites/Wishlist Slice

Stores array of product IDs representing favorited items. Separate from cart to allow wishlist items to be managed independently.

Reducers:

- `setFavorites`: Initializes wishlist from saved state
- `addFavorite`: Adds product ID to wishlist array
- `removeFavorite`: Removes product ID from wishlist array

Persistence: Like cart, wishlist updates are persisted to local storage immediately.

Products Slice

Stores all available products fetched from the backend. This slice is primarily read-only after initial population.

Reducers:

- `setProducts`: Replaces entire products array with fetched data from `/products/all`

Usage: Components filter and search this data locally rather than making repeated backend requests. Products include all necessary information for display and selection.

Orders Slice

Stores all orders for the authenticated user, fetched from `/products/orders`.

Structure: Each order contains product details at time of purchase (preventing price changes from affecting historical orders), order status (pending/completed/canceled), total price, payment URL, and timestamps.

Reducers:

- `setOrders`: Replaces orders array with data fetched from backend

Usage: Profile and order history pages display order data. Pending orders show payment links. Completed orders show product receipts.

Search Slice

Manages the search query string for product filtering. Simple string state that's normalized (trimmed and lowercase) to ensure consistent search behavior.

Reducers:

- `setSearch`: Updates search term and normalizes to lowercase for case-insensitive filtering

Usage: Product list components filter displayed products based on search term. Updates in real-time as user types.

Layout & Component Structure

```
App (Router setup, initialization)
├── MainLayout (Wrapper for authenticated routes)
│   ├── Header (Navigation, search, user menu)
│   └── Outlet (Route-specific content)
│       ├── LandingPage
│       ├── Home (Product list with search)
│       ├── ProductPage (Individual product details)
│       ├── Wishlist (Protected: favorited products)
│       ├── Cart (Protected: shopping cart)
│       └── Profile (Protected: user account & orders)
├── Login (Unprotected)
├── Register (Unprotected)
└── ProtectedRoute (Wrapper for guarded routes)
```

The MainLayout component includes a Header (navigation and search) and an Outlet where route-specific content renders. This structure ensures consistent header display across protected routes.

Data Flow Patterns

Adding Product to Cart:

1. User clicks "Add to Cart" on product component
2. `addProduct` action dispatched with product ID and quantity
3. Redux updates cart state, automatically persists to local storage
4. Component re-renders showing updated cart count in header
5. User can proceed to cart or continue shopping

Checkout Process:

1. User reviews cart and initiates checkout
2. Frontend validates cart against current product data (prices, availability)
3. User selects loyalty points to redeem
4. API request sent to `/products/checkout` with validated cart and points
5. Backend validates again and returns Stripe checkout URL
6. User redirected to Stripe hosted checkout page
7. After payment, Stripe redirects back to frontend
8. Frontend refetches orders on redirect, showing new order in history

Search & Filter:

1. User types in search box
2. `setSearch` action dispatched with search term
3. Product list component subscribes to search slice
4. Component filters products array locally matching search term
5. Filtered products re-render instantly

Local Storage Strategy

Cart and wishlist are persisted to local storage to survive page reloads and browser closures. This provides seamless user experience where items remain available when returning to the site.

Safety Mechanisms:

- `safeParse` utility wraps JSON parsing in try-catch

- Type validation ensures stored data matches expected structure
- Invalid data defaults to empty arrays
- Initial app load clears saved data on authentication failure, preventing stale data from persisting

When Local Storage is Cleared:

- After failed authentication check (on app startup if session expired)
- Explicitly on logout to ensure no sensitive data remains

API Integration

The frontend uses an Axios instance configured to include credentials (cookies) with all requests. This automatically sends JWT tokens stored in HTTP-only cookies, eliminating the need to manually handle token passing.

Key API Calls:

- `GET /auth/me` - Check authentication status on app load
- `POST /auth/login` - User login
- `POST /auth/register` - User registration
- `POST /auth/google` - Google OAuth login
- `POST /auth/logout` - Clear session
- `GET /products/all` - Fetch all available products
- `POST /products/checkout` - Initiate payment with cart and points
- `GET /products/orders` - Fetch user's order history

Errors from `/products/orders` are silently caught since this endpoint fails if not authenticated, which is expected on page load before auth check completes.

Type Safety

TypeScript types are defined for all Redux state shapes and API responses. This prevents typos and provides autocomplete in Redux selectors and component props. The app uses `useSelector` and `useDispatch` with proper typing from the store configuration.

Component Preparation & Optimization

The frontend carefully prepares data before rendering:

Product List: Filters against products array locally, avoiding unnecessary API calls. Search is normalized (lowercase, trimmed) for consistent matching.

Cart Display: Reconstructs total prices and item counts from local cart state and product database, never trusting raw cart data from local storage.

Wishlist: Stores only product IDs to minimize storage, looks up full product data from products array for display.

Orders: Displays as-is from backend since order history includes snapshots of products at purchase time.

Protected Routes: Validates user exists before rendering protected content. Shows loading screen while auth status is being determined.

Summary

This e-commerce platform demonstrates production-grade practices for handling complex real-world scenarios in payment processing, inventory management, and user authentication. The backend prioritizes data consistency through transactions, security through encryption and validation, and error handling for race conditions. The frontend complements this with careful state management, persistent storage with safety mechanisms, and robust type safety throughout. Together, they create a seamless user experience while protecting against common vulnerabilities and data inconsistencies.