

```
1 # Project Handover Report - Clothing Store E-commerce Platform
2
3 ## 1. Technical Architecture Overview
4
5 ### Technology Stack
6 **Frontend:** React 19 with React Router 7,
  Progressive Web App (PWA) with service workers for
  offline capability, localStorage for cart
  persistence, and Context API for state management (
  Authentication and Shopping Cart).
7
8 **Backend:** Node.js with Express 5, TypeScript for
  type safety, Passport.js for Google OAuth 2.0
  authentication, Express Session with MongoDB
  session store for session management.
9
10 **Database:** MongoDB with Mongoose ODM, containing
  collections for Users, Products, Categories, Sizes
  , Stock (inventory), Orders, and saved payment
  Cards.
11
12 **Deployment:** Currently configured for local
  development (Frontend: localhost:3001, Backend:
  localhost:3000). Production deployment not yet
  configured.
13
14 ### System Architecture & Data Flow
15 The application follows a standard client-server
  architecture with the following data flow:
16
17 1. **User Authentication:** User initiates Google
  OAuth login → Backend redirects to Google → Google
  callback returns to backend → Session created in
  MongoDB → Frontend receives session token → User
  state stored in React Context
18 2. **Product Browsing:** Frontend requests products
  by category → Backend queries MongoDB with
```

18 category filter → Products with populated category references returned → Images served from static folder

19 3. ****Shopping Cart:**** User adds items → Cart stored in browser localStorage → On checkout, cart data sent to backend → Backend validates stock availability → Stock quantities decremented → Order created in MongoDB

20 4. ****Order Management:**** Orders linked to users via userId reference → Admin panel fetches all orders with populated user and product data → Status updates trigger database updates

21

22 **### Key Design Decisions**

23

24 ****Google OAuth over custom authentication:****
Reduces security risks by delegating authentication to Google, eliminates need for password management, and provides trusted user verification. This was chosen to accelerate development and improve security posture.

25

26 ****MongoDB over SQL:**** Selected for flexible schema design allowing easy addition of product attributes, natural JSON structure matching frontend/backend data flow, and simpler deployment without complex migrations. The product catalog's semi-structured nature fits NoSQL better than rigid relational tables.

27

28 ****PWA Implementation:**** Enables offline browsing of cached products, provides app-like experience with install prompts, and improves loading performance through service worker caching. This enhances user experience especially on mobile devices and slow networks.

29

30 ****Context API over Redux:**** Sufficient for current

30 application scale, reduces boilerplate code, and provides simpler learning curve. The application has only two global states (auth and cart), making Redux's complexity unnecessary.

31

32 ****Separate Size and Stock Models:**** Allows flexible inventory management where same product can have different stock levels per size and color combination. This supports realistic e-commerce scenarios where availability varies by variant.

33

34 **## 2. Technical Debt & Known Limitations**

35

36 **### Current Problems**

37

38 ****Security Issues:****

39 - Payment card data stored in plain text including full card numbers (CVV stored temporarily). This violates PCI DSS compliance and exposes sensitive data. Real applications must use payment processors like Stripe.

40 - Session secret hardcoded in .env file should be rotated regularly and stored in secure key management system.

41 - No rate limiting on API endpoints allows potential DDoS attacks.

42 - CORS configured for specific localhost URLs only ; needs environment-based configuration for production.

43

44 ****Code Quality Issues:****

45 - Mixed TypeScript (.ts) and JavaScript (.js) files create inconsistent type safety. Controllers are in JavaScript while routes/models are TypeScript.

46 - Inconsistent error handling - some routes have try-catch blocks, others don't. Missing centralized error handling middleware.

47 - No input validation on most endpoints (missing

47 express-validator or joi). User input directly
passed to database queries.

48 - Test files present but not implemented (App.test.
js contains only placeholder test).

49

50 ****Performance Limitations:****

51 - All product images loaded on initial page load
without lazy loading or pagination.

52 - No caching strategy for frequently accessed data
(categories, sizes).

53 - Database queries not optimized - missing indexes
on frequently queried fields like productId, userId
.

54 - Stock updates not transaction-safe; concurrent
orders could oversell inventory.

55

56 ****Incomplete Features:****

57 - Admin panel has "Edit" and "Delete" buttons on
products but handlers not implemented.

58 - Order tracking system non-functional - status
updates don't trigger notifications.

59 - Email confirmation for orders not implemented.

60 - Product search/filter functionality missing (only
category filtering exists).

61 - User profile editing not possible (name, avatar
cannot be changed).

62 - No password reset flow (though using OAuth
reduces this need).

63

64 ****Deployment Gaps:****

65 - No environment configuration for production
builds.

66 - Database seeding script (seed.js) contains
hardcoded data with no proper production data
strategy.

67 - Static image files served from local filesystem
with hardcoded localhost URLs in database.

68 - No CI/CD pipeline or deployment documentation.

69 - Missing health check endpoints for monitoring.

70

71 ### Why These Are Problematic

72

73 The plain text card storage is the most critical issue - a data breach would expose complete payment information causing legal liability, financial losses, and destroyed customer trust. The lack of input validation creates SQL/NoSQL injection vulnerabilities allowing attackers to manipulate or delete data. Missing transaction handling means race conditions during checkout can cause inventory inconsistencies, overselling products. The hardcoded localhost image URLs will break entirely in production, preventing any product images from displaying.

74

75 ### Intentional Prioritization Decisions

76

77 ****Payment processing:**** Full Stripe/PayPal integration deprioritized to focus on core e-commerce flow. Current implementation serves as UI prototype but must not go to production.

78

79 ****Email notifications:**** Skipped to meet deadline - requires email service integration (SendGrid, AWS SES) which adds infrastructure complexity.

80

81 ****Advanced product filtering:**** Basic category filtering implemented; comprehensive search with size/color/price filters deemed non-critical for MVP.

82

83 ****Automated testing:**** Test infrastructure present but not implemented due to time constraints. Manual testing performed instead.

84

85 ****Image upload system:**** Products use external

85 URLs; admin upload interface postponed. Current approach sufficient for demo but limits content management.

86

87 ## 3. Future Development Recommendations

88

89 ### Must Implement (Security & Critical Functionality)

90

91 **Integrate proper payment processor:** Replace card storage with Stripe Elements or PayPal SDK. Store only payment method tokens, never raw card data. This is non-negotiable for production launch.

92

93 **Add comprehensive input validation:** Implement express-validator on all POST/PATCH endpoints. Validate data types, required fields, string lengths, and sanitize inputs. Prevents injection attacks and data corruption.

94

95 **Implement proper authentication middleware:** Add JWT refresh tokens alongside sessions. Create proper admin authorization middleware checking isAdmin flag before sensitive operations. Currently admin routes rely on trust.

96

97 **Add database transactions for orders:** Wrap order creation and stock updates in MongoDB transactions. Ensures atomicity - either complete order succeeds or entire operation rolls back, preventing inventory inconsistencies.

98

99 **Configure production environment:** Create separate .env files for development/staging/production. Use environment variables for all URLs, implement proper CORS configuration, add rate limiting middleware, and set up health check

```
99 endpoints.
100
101 ### Should Implement (Enhanced Functionality)
102
103 **Order email notifications:** Integrate SendGrid/
    AWS SES to send confirmation emails with order
    details and tracking information. Improves
    customer experience and reduces support inquiries.
104
105 **Product search and advanced filtering:** Add
    full-text search on product names/descriptions
    using MongoDB Atlas Search or Elasticsearch.
    Implement filters for price range, sizes, colors,
    and sorting options.
106
107 **Image management system:** Build admin interface
    for uploading product images to cloud storage (
    AWS S3, Cloudinary). Store CDN URLs in database
    instead of localhost paths.
108
109 **Inventory alerts:** Add low-stock notifications
    for admin dashboard. Implement automatic reorder
    suggestions when inventory falls below threshold.
110
111 **Order tracking:** Create detailed order status
    workflow (pending → processing → shipped →
    delivered) with automatic status updates and
    customer-facing tracking page.
112
113 **Testing suite:** Implement unit tests for
    controllers, integration tests for API routes, and
    E2E tests for critical flows (checkout,
    authentication). Target 70%+ code coverage.
114
115 ### Should Improve (Code Quality & Performance)
116
117 **Complete TypeScript migration:** Convert all
    JavaScript controllers to TypeScript. Add proper
```

117 type definitions for request/response objects.
Enable strict mode in tsconfig.json for better
type safety.

118

119 ****Add database indexes:**** Create indexes on `userId`
, `productId`, `categoryId`, and `order status` fields.
Implement compound indexes for frequently combined
queries (`productId + sizeId + color` for stock
lookups).

120

121 ****Implement caching layer:**** Add Redis for caching
product lists, categories, and sizes. Set
appropriate TTL values. Reduces database load and
improves response times significantly.

122

123 ****Centralized error handling:**** Create global
error handler middleware catching all errors.
Implement proper error classes (`ValidationError`,
`AuthError`, `NotFoundError`). Return consistent error
response format.

124

125 ****API versioning strategy:**** Current routes use ``/api/v1/``
prefix but no versioning strategy defined
. Document approach for introducing breaking
changes without disrupting existing clients.

126

127 ****Refactor large components:**** Break down
`ProductsPage`, `Checkout`, and `AdminPage` components
into smaller, reusable pieces. Extract custom
hooks for common logic (`useAuth`, `useCart` already
exist but could be expanded).

128

129 ****Optimize bundle size:**** Implement code splitting
on routes. Lazy load admin components only when
needed. Use `React.lazy()` and `Suspense` for
performance improvements.

130

131 ****Documentation:**** Add JSDoc comments to complex

131 functions. Create API documentation using Swagger/
OpenAPI. Document deployment procedures and
environment setup in detail.

132

133 ---

134

135 ****Recommended Priority Order:**** Security fixes →
Production deployment → Testing → Performance
optimization → Feature enhancements

136

137 ****Estimated Timeline:**** Critical security fixes (2
weeks) → Production deployment setup (1 week) →
Core functionality completion (3 weeks) →
Performance optimization (2 weeks) → Advanced
features (4+ weeks)