Trinity Store – Comprehensive Technical Architecture Documentation

---

1. System Architecture & Component Design

The application implements a **Multi-Tier Architecture** designed for scalability, security, and maintainability.

A. Client Tier (Frontend – React SPA)

Built as a **Single Page Application (SPA)**, the client tier handles the user interface, state management, and interaction logic.

Key Features:

- **Framework:** React 18+ using Functional Components and Hooks (`useState`, `useEffect`, `useContext`).
- **Navigation:** `react-router-dom` enables seamless client-side routing without full page reloads.
- **State Management:** Local component state for UI-specific data, with specialized hooks for data fetching.
- **API Orchestration:** Centralized `Axios` instance with interceptors (`services/api.js`) automatically attaching JWT tokens to outgoing requests.
- **Visualization:** `Recharts` provides responsive charting for admin analytics.

Benefits of SPA:

- Smooth, app-like navigation.
- Fast interactions with backend APIs.

- Reusable React components for maintainable code.
- Supports dynamic cart updates and product browsing.

B. Application Tier (Backend – Node.js/Express REST API)

The backend orchestrates business logic, security, and database interactions.

Key Components:

- **Entry Point:** `server.js` initializes environment and database connections.
- **App Configuration:** `app.js` sets up middleware (CORS, Helmet, Body-Parser) and route entry points.
- **MVC Pattern:**
    - **Routes:** Define endpoints and apply role-based access (e.g., `adminOnly`).
    - **Controllers:** Handle pure business logic (e.g., revenue calculation, API sync).
    - **Models:** Define Mongoose schemas with validation and middleware (pre/post-save hooks).
- **Global Error Handling:** Centralized middleware ensures consistent API responses.

Additional Notes:

- Optimized for high concurrency with Node.js non-blocking I/O.
- Security headers (Helmet) and input sanitization protect the backend from common attacks.

C. Data Tier (Database – MongoDB)

MongoDB is chosen for **flexible, polymorphic data storage**.

Key Features:

- **Engine:** MongoDB (NoSQL) stores complex product data (nutritional info, ingredients).
- **Object Modeling:** Mongoose provides schema validation, middleware, and population.
- **Indexing:** Critical fields like `email`, `barcode`, and `product name` are indexed for fast queries.
- **Scalability:** Supports horizontal scaling and sharding for large datasets.

---

2. Technological Choices: Rationale & Benefits

Category	Technology	Technical Rationale
Language	JavaScript (ES6+)	Enables shared logic between frontend and backend (isomorphic JS).
Build Tool	Vite	Native ES Modules enable fast cold starts and hot module replacement.
Backend Engine	Node.js (V8)	Event-driven, non-blocking I/O handles high concurrency efficiently.
Authentication	JWT (Stateless)	Eliminates server-side sessions, improving scalability across load-balanced instances.
Security Headers	Helmet.js	Automatically sets secure HTTP headers (CSP, X-Frame-Options, HSTS) for protection.
Sanitization	Custom NoSQL Sanitizer	Express 5 compatible, prevents `$operator` injection.
Documentation	Swagger UI	Live interactive API specification reduces frontend-backend friction.

---

3. Granular Data Flows

A. Stateless Authentication Flow

1. **Initiation:** User submits `email` and `password` via `POST /api/auth/login`.
2. **Identification:** Backend queries MongoDB and compares hashes using `bcrypt`.
3. **JWT Generation:** On success, backend signs a token containing `userId` and `role`.
4. **Frontend Storage:** Token is stored in `localStorage`.
5. **Consumption:** Axios interceptor injects token into `Authorization` header for subsequent requests.

B. Barcode Data Enrichment Flow (External API)

1. **Input:** Admin clicks “Fetch Details” with a 13-digit EAN/UPC barcode.
2. **Execution:** Backend fetches data asynchronously from Open Food Facts API.
3. **Resilience:** `AbortController` cancels requests exceeding 5 seconds.
4. **Extraction:** Relevant fields (e.g., `offData.nutriments.energy-kcal_100g`) are extracted.
5. **Sanitization:** Data is mapped safely to the internal Product schema (fallback: `null` or `0`).
6. **Error Handling:** Failed fetches log errors and return messages to the admin interface.
7. **Logging:** All API interactions are logged for auditing purposes.

C. Transactional Order Flow

1. **Validation:** User clicks “Place Order”; backend verifies JWT and checks item availability.
2. **Atomicity:** Stock decrement operations occur atomically.
3. **Consistency:** `Invoice` and `InvoiceItem` records are created in a single workflow.
4. **Resolution:** Frontend clears the cart and shows confirmation page.
5. **Logging:** All orders and stock changes are logged for traceability.
6. **Scalability:** Optimistic concurrency prevents overselling in high-traffic scenarios.

---

4. Security & Best Practices

- **Identity Layer:** Role-Based Access Control (RBAC) for `customer` and `admin`.
- **Input Validation:** Custom sanitization prevents NoSQL injection.
- **Transport Layer:** CORS and Helmet ensure safe communication.
- **Error Handling:** Centralized error middleware maintains API consistency.
- **Logging & Monitoring:** Critical flows are logged for audit and operational insights.