IEventory

IEventory

Store Management System

Internship Manager-prepared by: Aman Baye and Betelhem Desalegn

Submitted To: Samuel Anteneh
Date: August 2025

TABLE OF CONTENTS

1. Project Overview

Project Name: IEventory – Store & Inventory Management System.

Purpose: IEventory is built to help organizations manage their store resources more efficiently. It allows employees to borrow and return equipment, track consumables, log damaged items, assign delivery staff, and conduct audits.With multi-store support, it enables seamless material sharing between company stores.

Target Audience:
- ✓ Company store managers
- ✓ Employees requesting/borrowing equipment
- ✓ Delivery staff

Key Features:
- ✓ Borrowing and returning equipment
- ✓ Consumables tracking (stock-in/stock-out)
- ✓ Reporting and handling damaged items
- ✓ Assigning and managing delivery staff
- ✓ Audit and reporting system
- ✓ Multi-store support for resource sharing

2. Technology Stack

Frontend: React.js, Tailwind CSS
Backend: Express.js (Node.js framework)
Database: PostgreSQL
Other Tools & Services: Sequelize ORM, Passport.js, REST APIs, Git &GitHub, Postman

Layer	Technology	Purpose
Frontend	React.js, Tailwind CSS	User Interface
Backend	Node.js, Express.js	Business Logic & APIs
Database	PostgreSQL	Persistent Storage
Tools	Sequelize, Passport.js, Postman, GitHub	ORM, Auth, Testing, Version Control

3. System Architecture of IEventory

The architecture of **IEventory** follows a three-tier model, ensuring clear separation of concerns between the presentation layer (Frontend), the application logic (Backend), and persistent storage (Database). This design provides scalability, maintainability, and security for the system.

1. Frontend Layer (React.js)

Technology: React.js
Purpose: Acts as the user interface for all types of users (Admin, Employee, Delivery Staff).
Responsibilities:

- ✓ Offers role-based dashboards and intuitive UI components.
- ✓ Collects user input (e.g., borrow requests, return confirmations, damage reports).
- ✓ Displays real-time inventory, item availability, and delivery status.
- ✓ Manages application state with Redux (e.g., user session, permissions, theme, UI preferences).
- ✓ Fetches and synchronizes data from the backend using React Query, with caching and auto-refresh.
- ✓ Communicates with the backend through HTTPS API requests (REST or GraphQL).

Advantages:

- ✓ **Reusable UI components** → faster development and consistent design.
  Consistent data across dashboards using centralized state (Redux).
- ✓ **Improved performance** with React Query (cached results reduce repeated API calls).
- ✓ **Fast navigation** with Single Page Application (SPA) model.
- ✓ Future-ready → can integrate with mobile apps or other platforms easily.

2. Backend Layer (Express.js)

Technology: Node.js with Express.js
Purpose: The application logic layer that processes requests from the frontend, enforces business rules, and communicates with the database.

Responsibilities:

- ✓ Provides **role-based authentication and authorization** (Admin, Employee, Delivery Staff).
- ✓ Handles **business processes** like borrowing, returning, transferring, and auditing items.
- ✓ Offers **REST API endpoints** (e.g., `/api/items`, `/api/users`, `/api/deliveries`).
- ✓ Manages **error handling** and ensures reliable system responses.
- ✓ Secures data using **JWT tokens, HTTPS, and input validation**.

Advantages:

- ✓ **Handles multiple requests efficiently** with Node.js' event-driven model.
- ✓ Flexible with **middleware support** (logging, validation, error handling).
- ✓ **Easily scalable** to add new features such as notifications or advanced reporting.

3. Database Layer (PostgreSQL)

Technology: PostgreSQL (Relational Database)
Purpose: Acts as the persistent storage system for all inventory and user-related data.

Responsibilities:

- ✓ Stores structured data: users, items, transactions, deliveries, damages, audits.
- ✓ Enforces relationships between entities (foreign keys, constraints).
- ✓ Ensures data integrity and consistency across stores.
- ✓ Provides reporting and query support for audits and analytics.

Advantages:

- ✓ ACID compliance → reliable transactions.
- ✓ Advanced support for indexing, JSON data, and triggers.
- ✓ Suitable for multi-store inventory synchronization.

4. Communication Flow

1. **User Interaction:** A user logs in and makes a request through the React.js frontend.

2. **API Call (Managed by React Query)**: The frontend sends the request (e.g., borrow item) via HTTPS to the backend API. React Query caches responses and may serve cached data instantly while fetching fresh data in the background.

3. **Backend Processing:** The Express.js backend verifies authentication, checks business rules, and interacts with the database.

4. **Database Operations**: PostgreSQL executes queries (e.g., reduce stock, create transaction log).

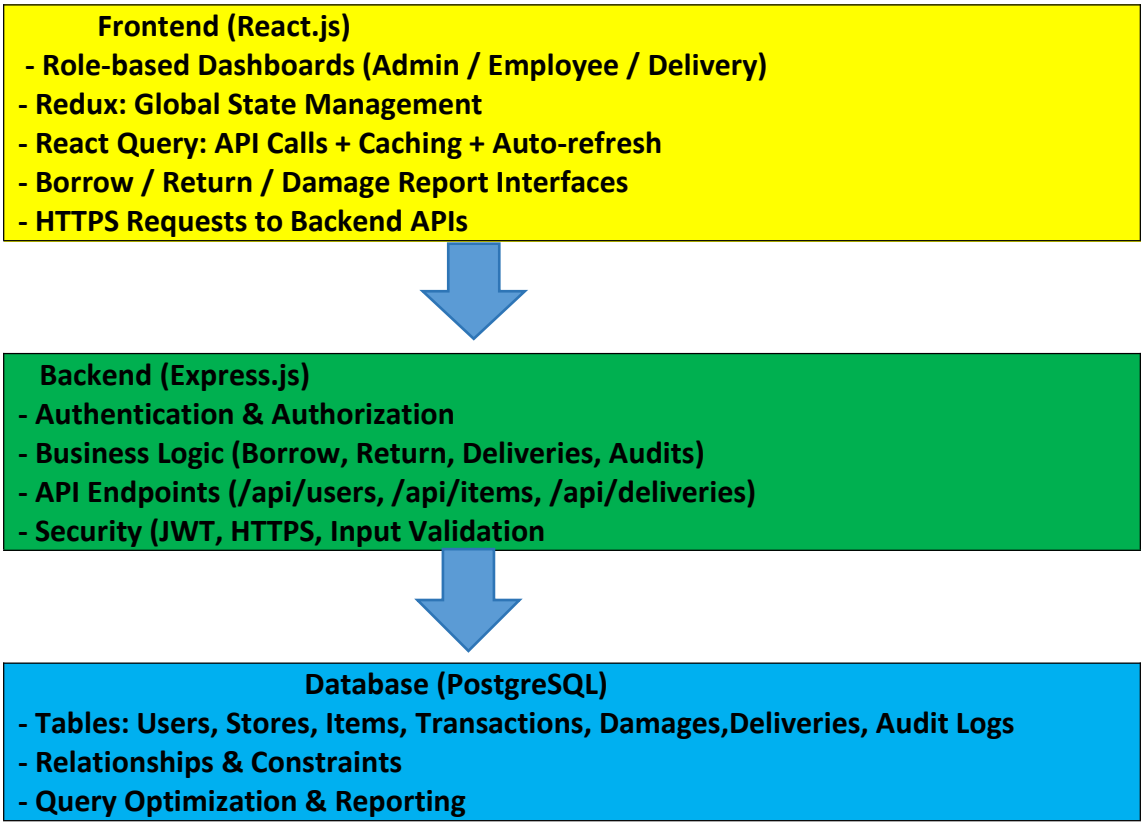
5. **Response Delivery & State Update:**
   - ✓ The backend sends a response (success/error + data) back to the frontend.
   - ✓ React Query updates its cache, and Redux updates global application state if necessary.

6. **UI Update**: The React.js frontend re-renders with the latest data in real-time.

Over all

User logs in → React frontend sends API request → Backend verifies auth → Database updates → Response →UI update

High-Level Architecture Diagram



Benefits of This Architecture

1) **Scalability**: Each layer can be scaled independently (e.g., load balancer for backend, replication for database).

2) **Maintainability**: Separation of concerns allows developers to work on different layers without breaking others.

3) **Security**: Sensitive data is kept at the backend and database layers; only authorized APIs are exposed.

4) **Flexibility**: Easily extendable for mobile apps, additional store locations, or cloud deployment.

5) **Performance Optimization (React Query)**: Cached responses reduce API calls and improve response times, while Redux provides a centralized state store.

4. Database Schema of IEventory

The IEventory system uses a relational database model (**PostgreSQL**) to store and manage data consistently across multiple stores. The schema is designed with normalization principles to avoid redundancy and ensure data integrity.

Entities and Tables

1. Users Table

Purpose: Stores information about all users (Admins, Employees, Delivery Staff).
Attributes:

- ✓ **user_id** (PK) – Unique identifier for each user.
- ✓ **name** – Full name of the user.
- ✓ **email** – Unique email for login.
- ✓ **password_hash** – Hashed password for authentication.
- ✓ **role** – Role (Admin, Employee, Delivery Staff).
- ✓ **store_id** (FK → Stores.store_id) – Store the user belongs to.
- ✓ **refresh_token** – Stores refresh token for session management.
- ✓ **password_reset_token** – Token for password reset functionality.
- ✓ **password_reset_expiry** – Expiry timestamp for reset token.
- ✓ **created_at, updated_at** – Timestamps for record tracking.

2. Stores Table

Purpose: Represents physical or virtual storage locations.
Attributes:

- ✓ **store_id** (PK) – Unique identifier.
- ✓ **store_name** – Name of the store.
- ✓ **location** – Physical or logical location details.
- ✓ **created_at, updated_at** – Timestamps.

3. Categories Table

Purpose: Defines item categories for better organization.
Attributes:

- ✓ **category_id** (PK) – Unique identifier.
- ✓ **name** – Category name (e.g., Equipment, Consumable).
- ✓ **description** – Optional description of the category.

4. **Items Table**
Purpose: Stores detailed information about inventory items.
Attributes:

- ✓ **item_id (PK)** – Unique identifier.
- ✓ **name** – Item name.
- ✓ **category**_id (FK → Categories.category_id) – Item category.
- ✓ **amount** – Current stock amount (renamed from quantity).
- ✓ **store_id** (FK → Stores.store_id) – Store where the item is located.
- ✓ **low_stock_threshold** – Stock amount that triggers alert.
- ✓ **supplier_id** (FK → Suppliers.supplier_id, optional) – Linked supplier.
- ✓ **image** – Optional image of the item.
- ✓ **model** – Optional model or version of the item.
- ✓ **serial_number** – Optional unique serial number.
- ✓ **date_of_purchase** – Purchase date of the item.
- ✓ **manufacturer** – Optional manufacturer of the item.
- ✓ **status** – Current status (e.g., available, damaged, reserved).
- ✓ **created_at, updated_at** – Timestamps for record creation and last update.

5. Transactions Table

Purpose: Tracks borrowing, returning, and transfers.
Attributes:

- ✓ **transaction_id** (PK) – Unique transaction identifier.
- ✓ **user_id** (FK → Users.user_id) – Who performed the transaction.
- ✓ **item_id** (FK → Items.item_id) – Item involved in transaction.
- ✓ **transaction_type** – Borrow, Return, or Transfer.
- ✓ **from_store_id** (FK → Stores.store_id, optional) – For transfers.
- ✓ **to_store_id** (FK → Stores.store_id, optional) – For transfers.
- ✓ **amount**– Number of items involved.
- ✓ **due_date** – Optional due date for borrowed items.
- ✓ **status** – Pending, Completed, Overdue, etc.
- ✓ **created_at, updated_at** – Timestamps.

6. Damages Table

Purpose: Records items reported as damaged.
Attributes:

- ✓ **damage_id** (PK) – Unique identifier.
- ✓ **item_id** (FK → Items.item_id) – Damaged item.
- ✓ **reported_by** (FK → Users.user_id) – User reporting the damage.
- ✓ **description** – Details of the damage.
- ✓ **status** – Pending, Fixed, or Discarded.
- ✓ **resolved_by** (FK → Users.user_id, optional) – Admin resolving the damage.
- ✓ **resolution_date** – Timestamp when resolved.
- ✓ **date_reported** – When the damage was reported.

7. Deliveries Table

Purpose: Handles item transfers and assigns delivery staff.
Attributes:

- ✓ **delivery_id** (PK) – Unique identifier.
- ✓ **transaction_id** (FK → Transactions.transaction_id) – Linked transaction.
- ✓ **assigned_to** (FK → Users.user_id) – Delivery staff.
- ✓ **status** – Pending, In-Progress, Completed.
- ✓ **pickup_time, delivery_time** – Timestamps for delivery tracking.

8. Audit Logs Table

Purpose: Keeps track of critical system events for accountability.
Attributes:

- ✓ **audit_id** (PK) – Unique identifier.
- ✓ **user_id** (FK → Users.user_id) – Who performed the action.
- ✓ **action_type** – Action description (Add, Update, Delete, Borrow, Return, etc.).
- ✓ **target_table** – Name of table affected.
- ✓ **target_id** – Record ID affected.
- ✓ **old_value, new_value** – Optional, for update tracking.
- ✓ **timestamp** – When the action occurred.

9. Suppliers Table

Purpose: Manages supplier information.
Attributes:

- ✓ **supplier_id** (PK) – Unique identifier.
- ✓ **name** – Supplier name.
- ✓ **contact** – Contact number.
- ✓ **email –** Email address.
- ✓ **created_at, updated_at** – Timestamps.

10. Maintenance Logs Table

Purpose: Tracks scheduled maintenance for items/equipment.
Attributes:

- ✓ **maintenance_id** (PK) – Unique identifier.
- ✓ **item_id** (FK → Items.item_id) – Item being maintained.
- ✓ **scheduled_date** – Scheduled maintenance date.
- ✓ **completed_date** – Date maintenance completed.
- ✓ **status** – Pending, Completed.
- ✓ **notes** – Optional notes.

11. Notifications Table

Purpose: Manages alerts and notifications.
Attributes:

- ✓ **notification_id** (PK) – Unique identifier.
- ✓ **user_id (FK → Users.user_id) –** Recipient of the notification.
- ✓ **type** – Notification type (email, dashboard alert).
- ✓ **message** – Content of the notification.
- ✓ **status** – Sent, Pending.
- ✓ **timestamp** – When created.

Relationships Between Entities

1. **Users ↔ Stores:** One store has many users.

   Stores.store_id → Users.store_id (1:M)

2. **Stores ↔ Items:** One store has many items.

   Stores.store_id → Items.store_id (1:M)

3. **Users ↔ Transactions ↔ Items:** Users perform many transactions; each transaction involves one item.

   Users.user_id → Transactions.user_id
   Items.item_id → Transactions.item_id

4. **Users ↔ Damages ↔ Items:** Users report damaged items.

   Users.user_id → Damages.reported_by
   Items.item_id → Damages.item_id

5. **Stores ↔ Deliveries ↔ Items ↔ Users:** Deliveries involve items moved between stores and assigned to delivery staff

   Stores.store_id → Transactions.from_store_id / to_store_id
   Users.user_id → Deliveries.assigned_to

6. **Users ↔ Audit Logs:** All actions logged for accountability.

   Users.user_id → Audit_Logs.user_id

7. **Items ↔ Suppliers:** Items linked to a supplier.

   Items.supplier_id → Suppliers.supplier_id

8. **Items ↔ Maintenance Logs:** Items can have multiple maintenance records.

   Items.item_id → Maintenance_Logs.item_id

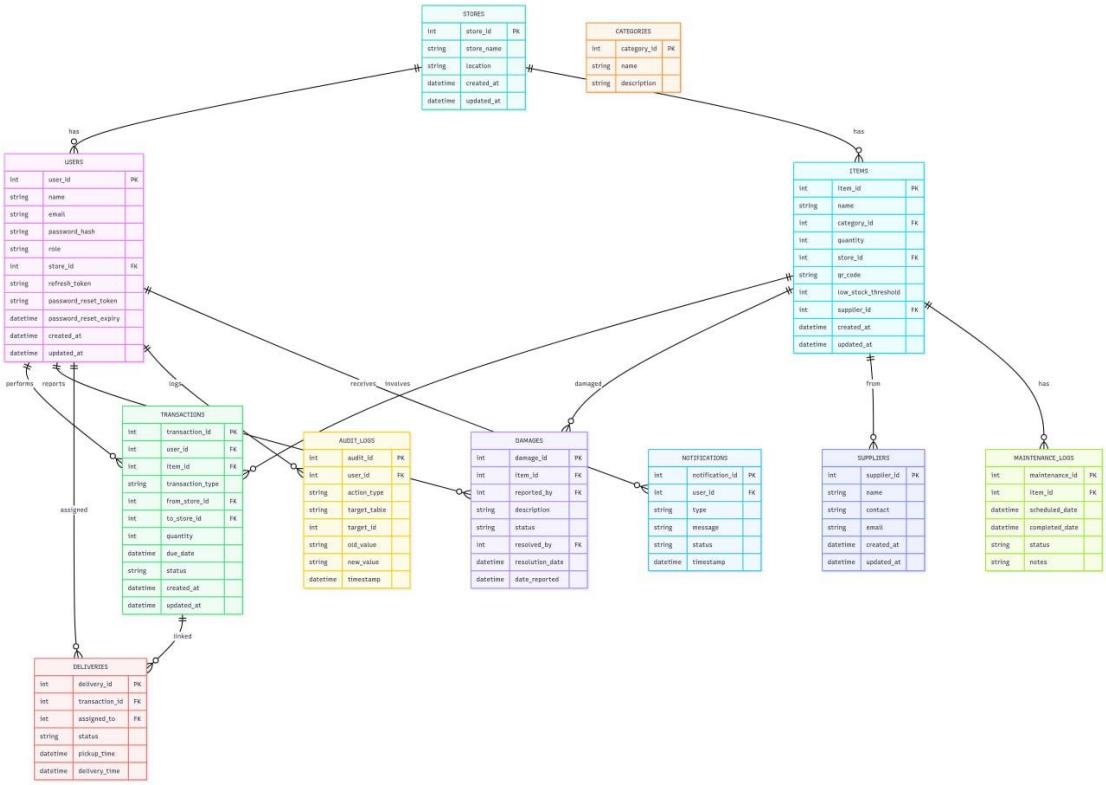
9. **Users ↔ Notifications:** Users can receive multiple notifications.

Users.user_id → Notifications.user_id

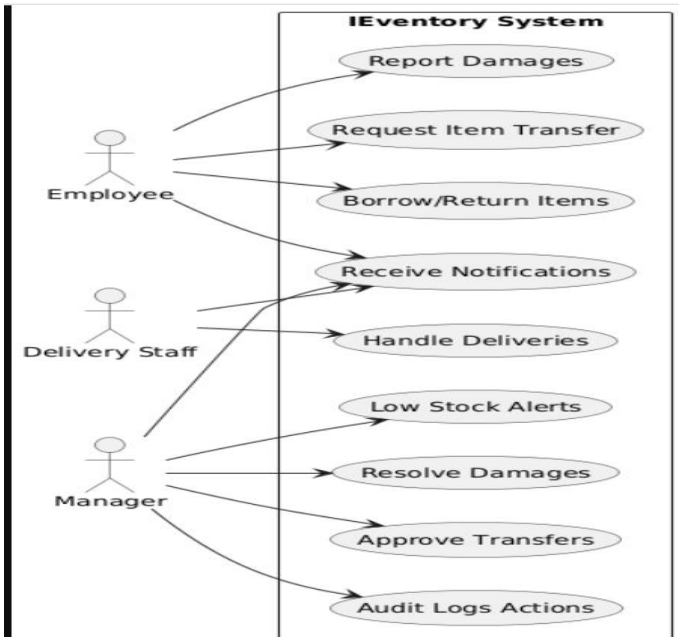
10. **Deliveries → Transactions**： 1:1 or 1:many, depending on whether multiple deliveries can exist for one transfer

    Deliveries → Transactions (Transaction_type = 'Transfer')

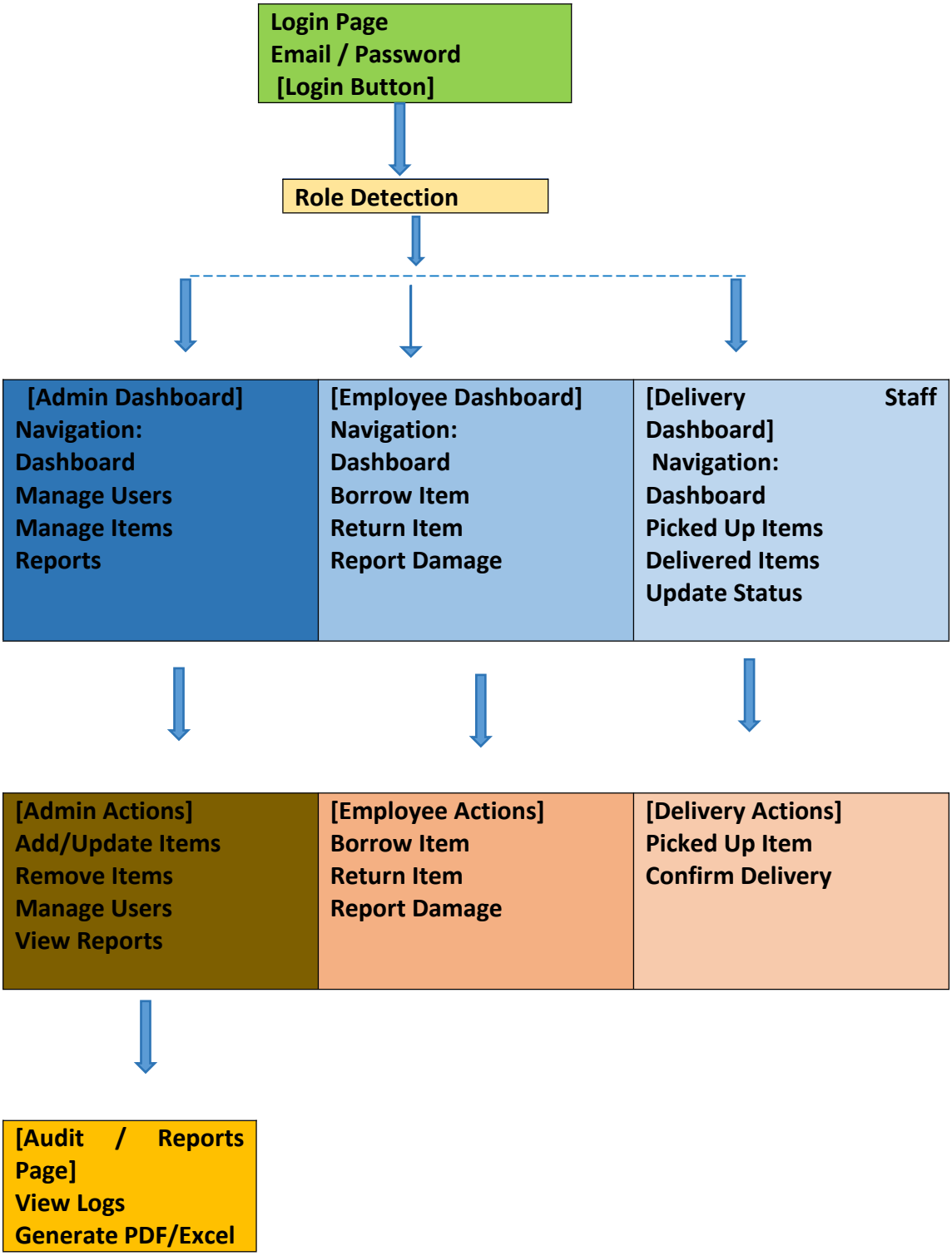
ER Diagram



Use case diagram



IEventory UI Flow Diagram



Flow Explanation

1) **Login Page**: All users authenticate here.
2) **Role Detection**: System detects the user role (Admin / Employee / Delivery Staff).
3) **Dashboards**: Each role has its dedicated dashboard with role-specific navigation.
4) **Actions**:
    - ✓ **Admin**: Manage inventory, users, view reports.
    - ✓ **Employee**: Borrow/Return items, report damaged items.
    - ✓ **Delivery Staff:** Update delivery status, confirm deliveries.
5) **Audit/Reports:** Admin can generate reports or review logs.

5. Implementation of IEventory

1. Folder Structure

```
IEventory/
│
├── backend/                    # Express.js backend
│   ├── controllers/            # Request handlers (items, users, deliveries)
│   ├── models/                 # Database models / schema definitions
│   ├── routes/                 # API route definitions
│   ├── middlewares/            # Authentication, validation, error handling
│   ├── utils/                  # Helper functions (e.g., token generation)
│   ├── config/                 # DB connection, JWT secrets, env setup
│   ├── tests/                  # Backend unit & integration tests
│   └── server.js               # Express server entry point
│
├── frontend/                   # React.js frontend
│   ├── public/                 # Static assets (index.html, images)
│   ├── src/
│   │   ├── components/         # Reusable UI components (buttons, tables, forms)
│   │   ├── pages/              # Pages & dashboards for Admin,Employee,Delivery
│   │   ├── redux/              # Redux slices & store for global state
│   │   ├── api/                # React Query hooks & API calls
│   │   ├── routes/             # React Router configuration
│   │   ├── styles/             # CSS / SCSS files
│   │   └── App.js              # Main app entry point
│   ├── tests/                  # Frontend component tests
│   └── package.json
│
├── database/                   # SQL scripts / migrations
├── .env                        # Environment variables
└── README.md
```

Explanation:

backend/: Handles server-side logic, routes, and database interactions.

front-end/: React UI components, Redux state, React Query hooks, and page routing.

database/: Stores PostgreSQL schema scripts or migrations.

config/: Holds environment-specific configs like JWT secrets and DB credentials.

2. Routing & Navigation

Frontend Routing

Uses React Router for SPA navigation.

Example routes:

- ✓ /login → LoginPage
- ✓ /admin/dashboard → AdminDashboard
- ✓ /employee/dashboard → EmployeeDashboard
- ✓ /delivery/dashboard → DeliveryDashboard
- ✓ /items → ItemsPage
- ✓ /users → UsersPage

Role-based routing ensures unauthorized users cannot access restricted pages.

Navigation Flow: Login → Dashboard → Role-specific actions → Reports/CRUD pages.

Backend Routing

Express.js REST API endpoints:

- ✓ `POST /api/auth/login` → Authenticate user, return JWT
- ✓ `GET/POST/PUT/DELETE /api/users` → Manage users (Admin only)
- ✓ `GET/POST/PUT/DELETE /api/items` → Manage items
- ✓ `POST /api/transactions` → Borrow/return items
- ✓ `POST /api/damages` → Report damaged items
- ✓ `GET/POST /api/deliveries` → Delivery management
- ✓ `GET /api/audit` → View audit logs (Admin only)

React Query fetches data from these endpoints and caches results.

Redux stores global app state (user info, inventory data, borrowed items).

6. Features

1) Authentication & Authorization

Capabilities:

- ✓ Login with email/password
- ✓ JWT tokens for secure backend requests
- ✓ Password reset functionality
- ✓ Session management with refresh tokens
- ✓ Role-based access control: Admin, Employee, Delivery Staff

Functional Details (Role Permissions):

- ✓ **Admin:** Can manage all user accounts, assign roles, reset passwords
- ✓ **Employee:** Can log in, reset their own password, and manage personal sessions
- ✓ **Delivery Staff:** Can log in and manage personal sessions only

2) Inventory Management

Capabilities:

- ✓ Add, update, delete, and view items (CRUD)
- ✓ Track stock levels, categories, and units of measurement
- ✓ Multi-store support: transfer items between stores
- ✓ Low Stock Alerts trigger notifications
- ✓ QR code support for faster scanning
- ✓ Store item details: image, model, serial number, manufacturer, date of purchase, status

Functional Details (Role Permissions):

- ✓ **Admin:** Add/update/delete items, transfer items, manage categories
- ✓ **Employee:** View items, request borrow
- ✓ **Delivery Staff:** View items (cannot modify)
- ✓ **Notes:** Low stock alerts notify Admin and optionally Employees

3) Borrow & Return

Capabilities:

- ✓ Employees request items to borrow; stock automatically updated on return
- ✓ Optional due dates for borrowed items
- ✓ All actions logged in Transactions table
- ✓ Redux updates borrowed items globally

Functional Details (Role Permissions):

- ✓ **Admin:** Can view all borrow/return transactions, optionally approve requests
- ✓ **Employee:** Can borrow/return items, view own transactions
- ✓ **Delivery Staff:** No borrow/return permissions

4) Damage Reporting

Capabilities:

- ✓ Employees report damaged items; Admin resolves issues
- ✓ Track status: Pending, Fixed, Discarded
- ✓ Optional notifications sent to responsible users

Functional Details (Role Permissions):

- ✓ **Admin:** Resolve damages, update status, view all damage reports
- ✓ **Employee:** Report damage, view status of reported items
- ✓ **Delivery Staff:** View items, but cannot report damage

5) Delivery Management

Capabilities:

- ✓ Delivery staff view assigned deliveries
- ✓ Update delivery status: Picked up, In-Progress, Completed
- ✓ Optional notifications for status updates

Functional Details (Role Permissions):

- ✓ **Admin:** Assign deliveries, track status
- ✓ **Delivery Staff:** Update status of assigned deliveries
- ✓ **Notes:** transaction_id in Deliveries table **must reference a transaction of type "Transfer" only**

6) Audit & Reporting

Capabilities:

- ✓ Admin can view audit logs of critical actions (Add, Update, Delete, Borrow, Return)
- ✓ Generate reports for inventory, transactions, damages, deliveries
- ✓ Logs optionally track old vs new values

Functional Details (Role Permissions):

- ✓ **Admin:** Full access
- ✓ **Employee / Delivery Staff:** No access

7) Search & Filtering

Capabilities:

- ✓ Search items, transactions, deliveries
- ✓ Filter by category, status, store, date range
- ✓ Advanced multi-criteria, full-text search

Functional Details (Role Permissions):

- ✓ **Admin & Employee:** Can search and filter
- ✓ **Delivery Staff:** Can search assigned deliveries only

8) Notifications

Capabilities:

- ✓ Email and dashboard notifications for low stock, overdue returns, delivery updates, damage reports

Functional Details (Role Permissions):

- ✓ **Admin:** Receive all notifications
- ✓ **Employee:** Receive notifications related to own borrow/return/damage reports
- ✓ **Delivery Staff:** Receive notifications for assigned deliveries

9) Supplier & Maintenance Management

Capabilities:

- ✓ Track supplier information, link to supplied items
- ✓ Schedule and log maintenance for items/equipment

Functional Details (Role Permissions):

- ✓ **Admin:** Full access
- ✓ **Employee:** View suppliers and maintenance logs
- ✓ **Delivery Staff:** View maintenance logs only

10) Dashboard & Mobile Responsiveness

Capabilities:

- ✓ Dashboard shows quick insights: low stock, pending deliveries, borrowed items
- ✓ Frontend mobile responsive for field use

Functional Details (Role Permissions):

- ✓ **Admin:** Full dashboard
- ✓ **Employee:** Limited dashboard (borrowed items, low stock)
- ✓ **Delivery Staff:** Limited dashboard (assigned deliveries)

Non-Functional Requirements

1. **Performance:**

- ✓ Inventory queries and updates respond within 2 seconds
- ✓ Reports generation ≤ 5 seconds for large datasets

2. **Reliability & Availability:**

- ✓ Support multiple concurrent users without conflicts
- ✓ Daily database backups
- ✓ 99.5% uptime

3. **Security:**

- ✓ Role-based access control
- ✓ Passwords hashed; JWT tokens for API requests
- ✓ Sensitive actions logged in audit logs

4. **Usability:**

- ✓ Intuitive and mobile-friendly interface
- ✓ Low learning curve for all users

5. **Maintainability & Scalability:**

- ✓ Easily add new stores, users, or features
- ✓ Database follows normalization principles

6. **Data Integrity:**

- ✓ Prevent negative stock amounts
- ✓ Ensure proper foreign key relationships
- ✓ Deliveries must only reference transfer transactions

7. **Notifications & Alerts:**

- ✓ Real-time or near real-time notifications
- ✓ Dashboard accurately reflects system state.

Role-Based Access

Feature	Admin	Employee	Delivery Staff
Add Item	✓	✗	✗
Update Item	✓	✗	✗
View Item	✓	✓	✓
Borrow Item	✗	✓	✗
Deliver Item (transfer)	✗	✗	✓

7. Deployment

We considered different hosting options for **frontend, backend, and database**:

1) **Frontend (React):**

- ✓ Hosted on **Vercel** (recommended for seamless Next.js deployments).
- ✓ Alternative: **Netlify** or company's internal server.

2) **Backend (Node.js/Express + APIs):**

- ✓ Deployed on **Render** / **Heroku** / **AWS EC2**.
- ✓ Connected to the frontend through environment variably `NEXT_PUBLIC_API_URL`.

3) **Database (PostgreSQL):**

- ✓ Managed service such as **Supabase**, **AWS RDS**, or **Railway**.
- ✓ Configured using `.env` database credentials.

8. Testing & Security

Testing Methods

1) **Manual Testing**

- ✓ Verify navigation between pages (login, inventory, reports).
- ✓ Check CRUD operations (Add, Edit, Delete, View items).
- ✓ Confirm search and filter functionalities return correct results.
- ✓ Validate role-based access (Admin vs. Staff).

2) **Automated Testing** *(optional, if implemented)*

- ✓ **Unit Tests**: Testing individual React components (e.g., LoginForm, ProductCard).
- ✓ **Integration Tests**: Validate Redux + React Query flows (state update after API call).
- ✓ **API Tests**: Use tools like Postman/Insomnia or Jest + Supertest to test backend endpoints.

Validation

1) **Frontend Validation (React + Form Handling)**

- ✓ Required fields: Item name, quantity, price, etc.
- ✓ Type checks: Quantity must be numeric, Email must match regex pattern.
- ✓ Password requirements: Minimum 8 characters, mix of letters/numbers/symbols.
- ✓ Real-time error messages with `react-hook-form` or custom validation.

2) **Backend Validation**

- ✓ Duplicate entries prevented (unique SKU or product ID).
- ✓ Ensure valid data before inserting/updating in database.
- ✓ Sanitize inputs to prevent SQL Injection & XSS attacks.

Security Measures

1) **Authentication**

- ✓ JWT-based authentication for secure user sessions.
- ✓ Tokens stored in **HTTP-only cookies** or secure storage.
- ✓ Refresh tokens for longer sessions.

2) **Authorization (Role-Based Access)**

- ✓ **Admin**: Full access (manage inventory, users, reports).
- ✓ **Staff/User**: Limited access (view, update stock only).
- ✓ Role enforcement done at both **frontend (React)** and **backend (API)**.

3) **Encryption**

- ✓ Passwords stored using **bcrypt hashing**.
- ✓ HTTPS enforced in production to protect data in transit.

9. Limitations & Future Improvements

Current Limitations

1) **No Offline Mode:** The system requires an internet connection to function; there is no offline-first capability.
2) **Basic UI/UX:** While functional, the interface lacks advanced styling, accessibility compliance (WCAG), and user experience enhancements.
3) **Role Management:** Roles are limited to Admin, Employee, Delivery Staff; there is no granular permission control for more complex organizations.
4) **Performance Constraints:** With very large datasets (thousands of products/transactions), performance may slow due to lack of optimization.
5) **Manual Testing:** Most testing has been manual; automated test coverage is still minimal.
6) **Single Language Support:** Currently, the app only supports English and does not include localization.

Suggested Future Improvements

1) **Offline Support (PWA):** Enable offline mode so users can continue using the system without internet, syncing once back online.
2) **UI/UX Enhancement:** Improve styling, responsive design, and accessibility compliance (WCAG).
3) **Performance Optimization:** Use database indexing, caching, and pagination for faster query response on large datasets.
4) **Automated Testing:** Integrate unit, integration, and end-to-end tests for reliability.
5) **Multi-Language Support:** Add localization for different languages to reach a wider audience.
6) **Mobile App Integration:** Build a companion mobile app for Android/iOS for on-the-go inventory management.
7) **AI/ML Features:** Implement demand forecasting and intelligent stock reordering using AI.

10. Conclusion

What was achieved in the project

- ✓ Successfully developed a **Store Management System (IEventory)** with core features such as authentication, CRUD operations, inventory tracking, and search functionality.
- ✓ Implemented **state management with Redux** and **data synchronization using React Query**, ensuring smooth user experience and real-time updates.
- ✓ Created a **responsive and user-friendly interface** with proper routing and navigation for easy access to all modules.

Impact for the company / team

- ✓ Simplified **inventory tracking and management**, reducing manual errors and improving efficiency.
- ✓ Provided a **centralized system** that can grow with the business needs, replacing outdated/manual processes.
- ✓ Enhanced collaboration among team members by offering a consistent platform with structured workflows.

Next steps if the company continues development

- ✓ Expand features to include **analytics and reporting dashboards** for deeper insights.
- ✓ Integrate **barcode/QR code scanning** for faster stock management.
- ✓ Enhance **security measures** with two-factor authentication and audit logging.
- ✓ Optimize system performance for **scalability**, supporting more users and larger inventories.
- ✓ Automate backups, updates, and notifications to ensure long-term reliability.