ChatGPT

# Offline-First Inventory System Implementation Plan (Small Retail, Nigeria)

## 1. Functional Requirements

- **Offline-First CRUD Operations:** The system will support Create, Read, Update, Delete for all core entities – **Products**, **Customers**, **Suppliers**, **Warehouses**, **Purchase Orders**, and **Sales Orders** – entirely offline. Users can add or modify records while offline; data is stored locally and syncs to a central server when internet is available [1] [2]. This ensures uninterrupted operation despite Nigeria's unreliable connectivity.

- **Real-Time Product Availability:** Users can query current stock levels and locations of products in real time. The app will maintain a local inventory count for each product (by warehouse) that updates immediately on each sale, purchase, or transfer. Even offline, the local PWA can instantly reflect updated availability. When re-connected, these changes sync bi-directionally with the server so all devices eventually see up-to-date stock [2]. Conflicts (e.g. two offline users selling the last item) will be handled gracefully via conflict resolution (described later).

- **Partial Text Search:** Implement client-side **full-text search** on product names, SKUs, and descriptions for quick lookup. We will use **Lunr.js** to index product data in the browser, enabling fast substring matches and fuzzy search offline. For example, a search for "Paracet" can find "Paracetamol" (using wildcards/fuzzy queries) [3] [4]. The search index updates whenever product data changes. This allows cashier staff to quickly find items by partial name or code without internet.

- **AI-Powered Similar Product Suggestions:** When viewing a product, the system will suggest similar or alternative products. This will be achieved using lightweight AI/ML techniques *offline*. For example, we can compute product similarity based on category, keywords, or historical purchase patterns. A possible approach is to assign vector embeddings to product descriptions and perform local nearest-neighbor search, or use a simple **content-based recommendation** (e.g. "customers who bought X also bought Y" using offline purchase history). These suggestions help recommend substitutes if an item is out of stock. (In Phase 2, we might start with rule-based suggestions – e.g. same category or manufacturer – then later incorporate ML for more advanced recommendations.)

- **User-Friendly UI (English):** The interface will be simple, optimized for non-technical users, and primarily in English (with clear labels for products, stock, etc.). Design will focus on clarity: large text for critical info (product names, quantities), minimal nested menus, and responsive layout for both low-cost laptops and Android tablets. The UI will include forms for data entry (with validation) and search filters (e.g. by category, warehouse). Though multi-language isn't in scope now, the design keeps text separate to allow future localization.

- **Future Integration of Scanning:** While Phase 1 won't have RFID/barcode scanning, the system is designed with placeholders for these. For example, the product schema includes fields for **Barcode/UPC**, and the UI allows input via keyboard or future scanner input. In a later phase, we

will enable a USB or Bluetooth barcode scanner to input product codes into search or forms (triggering the same lookup logic). RFID integration (for bulk scanning or real-time tracking) is planned; the architecture will accommodate it by supporting unique tag IDs and an interface to capture inventory movements via RFID readers [5] [6].

## 2. Non-Functional Requirements

- **Local-First Data & Sync:** The system uses a **Local Database (PouchDB)** on each device to store all data for offline use. PouchDB is a JavaScript NoSQL DB that runs in-browser (via IndexedDB) and allows full CRUD operations locally [1]. Whenever network is available, PouchDB will automatically synchronize with a central **Apache CouchDB** server (cloud or on-prem) using CouchDB's replication protocol [2]. Sync is **bi-directional and incremental**, meaning changes made offline propagate to the server (and then to other clients), and vice-versa [2]. This ensures eventual consistency across the 5–10 devices even if they rarely are online at the same time.

- **Conflict Resolution:** In a distributed offline system, conflicts can occur if two users edit the same record offline. CouchDB/PouchDB handle this by assigning revision numbers and detecting conflicts on sync. By default, CouchDB will choose a "winning" revision via deterministic algorithm and flag the other as a conflict [7] [8]. Our application will implement **automated conflict resolution** rules where possible. For example, for inventory count conflicts we might take the lowest quantity (assuming both sales happened) or sum quantities if appropriate. For other records, last-write-wins might be acceptable, or we prompt an admin to reconcile. PouchDB gives us programmatic control to resolve conflicts in code if needed [9] [7]. We will log conflicts in an Audit Log for review. The goal is a seamless user experience: most conflicts resolved under the hood, with sensible defaults (e.g. merge line-items in an order if two offline edits add different items).

- **Performance on Low Resources:** The stack is chosen for *lightweight performance*. The frontend is a React Progressive Web App which runs in a browser and can be cached for offline use. We avoid heavy libraries beyond our needs. PouchDB operations on local IndexedDB are fast (for 5-10k records scale) and queries are indexed (via PouchDB's find or map/reduce). The memory footprint is modest and adjustable (we'll purge old sync data as needed). The system will be tested on typical low-end Windows laptops (e.g. 2-4GB RAM, dual-core CPUs) and Android tablets to ensure UI responsiveness (<1 second for common actions) and acceptable load times. Data sync is done in the background with minimal impact on UI threads. We'll also implement pagination or virtual scrolling in product lists to handle large catalogs smoothly.

- **Compatibility:** The solution will run on **Windows and Linux** (and potentially Android tablets via browser or an embedded WebView). As a PWA, it works in Chrome, Firefox, Edge, etc., so OS-specific issues are minimal. We will ensure the app can be installed as a desktop app (PWA install or possibly wrapped in Electron if needed for better OS integration). No special hardware or OS beyond standard configurations is required.

- **Offline PWA Capability:** We will leverage service workers to cache the application shell and assets so that the app loads even with **no internet**. Using the Create React App PWA template, the service worker will precache static files (HTML, JS, CSS) and even certain API calls if needed [10] [11]. This makes the app a true **offline-first Progressive Web App**: users can navigate to the app URL and it will open (from cache) regardless of connectivity. The service worker also allows background sync or push notifications in future (for example, to trigger a sync or alert when back online). We will register the service worker for offline use as recommended [11].

- **Security & Access Control:** The system will implement **role-based access control (RBAC)** to restrict features by user role (e.g. a cashier vs. a manager). This is critical in a multi-user environment to prevent unauthorized actions (like a cashier editing product prices or viewing sensitive reports). We will have a login system (CouchDB supports users or we use a simple JWT-based auth). Each User is assigned one or more Roles (like *Admin*, *Manager*, *Sales Clerk*, *Pharmacist*). The UI will show/hide menu options based on role, and critical mutations will be double-checked against the user's role. Data security is addressed at rest and in transit: local PouchDB data can be encrypted if needed (PouchDB has plugins for encrypted storage [12]), and sync to CouchDB will be over HTTPS with credentials. CouchDB will be configured with an admin user and CORS for our app domain. Although offline-first implies data resides on devices, we'll instruct users on device security (OS login, etc.) and consider adding an app PIN or encryption for sensitive data (e.g. customer contacts).

- **Reliability and Fault Tolerance:** The system is designed to be resilient to connectivity dropouts and power loss. All writes go to local storage first – so even if the internet or power goes out, no data is lost; it will sync when possible. CouchDB on the server can run in clustered mode for high availability (multiple nodes) [13]. If the central server is unreachable for an extended period, each client still has a full copy of data to continue operations (essentially multi-master replication). We will also implement periodic local backups (e.g. allow exporting the local DB to a file) as a safety measure. The sync process will use **live replication** with retry, so it continuously attempts to push/pull updates when online (managed by PouchDB's sync API with `{live: true, retry: true}`).

## 3. Database Design

We design a **normalized relational schema (3NF)** for clarity and integrity, and map it to CouchDB documents for implementation. Normalizing avoids redundancy and anomalies [14] [15]. Each entity corresponds to a document type in CouchDB (with type tags) or a table in an ER model. Key entities and relationships:

- **Product** – Represents an item for sale. Fields: `ProductID` (PK), `Name`, `Description`, `SKU` (unique, indexed), `Barcode` (indexed, for future scanning), `CategoryID` (FK to Category), `SupplierID` (FK to Supplier), `Price` (selling price), `Unit` (e.g. box, each), `ReorderPoint`, etc. **Relationships:** Many Products belong to one Category; many Products can be supplied by one Supplier (in this design we store one primary supplier per product, but a separate SupplierProduct map could allow many-to-many if needed). Indexes on SKU and Barcode allow fast lookups by code [16].

- **Category** – Hierarchical classification for products. Fields: `CategoryID` (PK), `Name`, `ParentCategoryID` (self-FK for subcategories), etc. Each Product links to a Category. This allows grouping and filtering (e.g. list all products in "Beverages"). A Category can have many Products (1–* relationship).

- **Supplier** – Vendors who supply products. Fields: `SupplierID` (PK), `Name`, `ContactInfo` (phone, email, address), `PaymentTerms`, etc. One Supplier supplies many Products. We track suppliers to manage purchase orders and reordering.

- **Customer** – Retail customers. Fields: `CustomerID` (PK), `Name`, `Phone`, `Email` (indexed, unique), `LoyaltyProgramID` (FK, if customer is enrolled in a loyalty program/tier), `Points`

(current loyalty points balance), etc. A Customer can place multiple Sales Orders (1–* relationship).

- **SalesOrder** – Represents a customer sale/transaction (could be a cash sale or invoice). Fields: `SalesOrderID` (PK), `OrderDate` (indexed for reporting), `CustomerID` (FK, can be null for walk-in cash sales), `TotalAmount`, `PaymentStatus` (Paid, Pending), etc. **Relationships:** One SalesOrder has many **SalesOrderItem** entries (details of each product sold). SalesOrder is linked to Customer (if known) and generates InventoryMovements (stock decrements).

- **SalesOrderItem** – Join table (detail line items for SalesOrder). Fields: composite PK or separate `ItemID`, plus `SalesOrderID` (FK to SalesOrder), `ProductID` (FK to Product), `Quantity`, `UnitPrice`, `Discount` (if any), `BatchNo` (if batch tracking enabled, the lot number sold), `ExpiryDate` (optional, if we record the expiry of the batch sold). There is a one-to-many from SalesOrder to SalesOrderItem, and many-to-one from SalesOrderItem to Product (each line references one Product). This structure allows an order to contain multiple products. We index `ProductID` in this table for faster product-wise sales queries.

- **PurchaseOrder** – Represents an order placed on a Supplier to restock. Fields: `PO_ID` (PK), `OrderDate`, `SupplierID` (FK), `Status` (e.g. Pending, Received), `TotalCost`, etc. Relationship: one PurchaseOrder to many **PurchaseOrderItem** records.

- **PurchaseOrderItem** – Join table for PurchaseOrder lines. Fields: `PO_ID` (FK), `ProductID` (FK), `QuantityOrdered`, `CostPrice`. When goods are received, each PurchaseOrderItem will generate an InventoryMovement (stock increase).

- **InventoryMovement** – Records any change in inventory (stock in or out) for audit and multi-warehouse tracking. Fields: `MovementID` (PK), `ProductID` (FK), `WarehouseID` (FK), `QuantityChange` (positive for stock in, negative for stock out), `Type` (enum: *SALE*, *PURCHASE*, *ADJUSTMENT*, *TRANSFER*), `RefOrderID` (links to a SalesOrder or PurchaseOrder ID if applicable), `Timestamp`. Every stock change creates an InventoryMovement so we have a ledger of inventory. For example, a SalesOrder of 2 units creates a Movement with QuantityChange = -2. A PurchaseOrder receipt of 50 units creates a +50 movement. **Multi-warehouse:** If transferring stock between warehouses, we can record two Movements: one negative in the source warehouse, one positive in the target warehouse (both tagged with a common transfer reference). A Warehouse has many InventoryMovements; a Product has many Movements; and each Movement links to one Product and one Warehouse. We will index `ProductID, WarehouseID` in this table for quickly computing stock per warehouse.

- **Warehouse** – Physical inventory locations. Fields: `WarehouseID` (PK), `Name` (e.g. “Main Store” or “Backroom Storage”), `Location` (address or description). A Warehouse holds many products (through InventoryMovements or a stock table). Initially for a single pharmacy, there may be just “Main Store” and perhaps a “Back Storage” as two warehouses. As business grows, multiple warehouses or stores can be added.

- **Stock (Denormalized):** *Derived data:* We may include a **CurrentStock** table or view (ProductID, WarehouseID, CurrentQuantity) for fast lookup of stock on hand, instead of calculating from movements each time. This can be maintained by summing InventoryMovements or updated in real-time via triggers in the app. For now, the PouchDB app can maintain a `currentQuantity` field per Product per Warehouse in memory or as part of the product document for quick availability queries, and reconcile with movements periodically.

- **LoyaltyProgram** – Represents a customer loyalty or membership program (e.g. tiers like Silver, Gold or a point scheme). Fields: `ProgramID` (PK), `Name` (e.g. "Gold Membership"), `Description`, `RewardRules` (e.g. points per purchase). A Customer can belong to one LoyaltyProgram (so one-to-many: one program, many customers). Loyalty rules (like earn 1 point per ₦100 spent) will be encoded in logic or in this table. This helps implement loyalty points and rewards.

- **EmailTemplate** – Stores templates for emails (for integration with email marketing or transactional emails). Fields: `TemplateID` (PK), `Name` (e.g. "Order Confirmation"), `Subject`, `Body` (with placeholders). We can have templates for things like low-stock alerts, weekly promotional emails, etc., which can be filled with data and sent when online. Integration with an SMTP server or API will be added when we implement email features.

- **User** – System user account for staff. Fields: `UserID` (PK), `Name`, `Email` (for login), `PasswordHash`, etc. Also includes `Role` info (though roles likely in a separate table for many-to-many). This table will manage authentication and authorization. Each user will be assigned one or more roles that define their permissions.

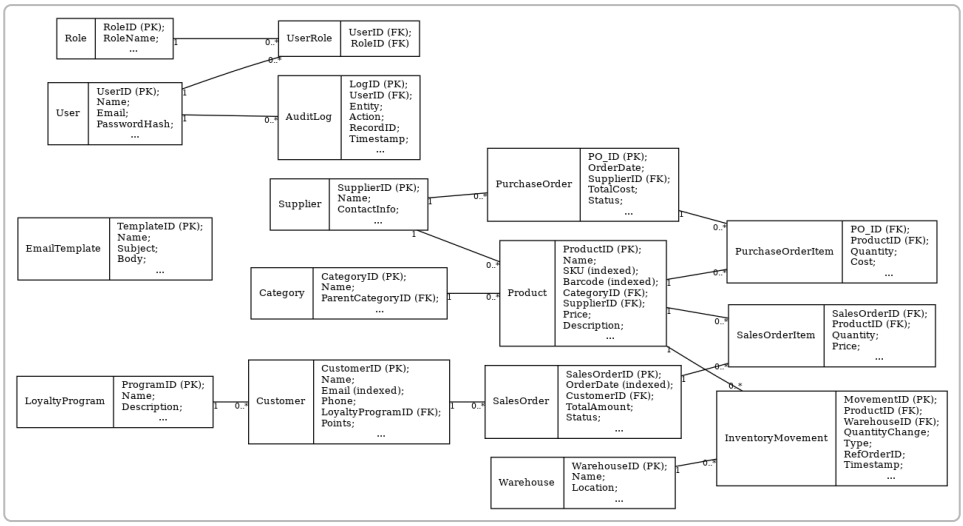
- **Role** – Defines a user role (e.g. "Admin", "Manager", "Cashier"). Fields: `RoleID` (PK), `RoleName`, `Permissions` (this could be a list of allowed actions or a reference to a permissions matrix). We will have a set of predefined roles to start.

- **UserRole** – Join table mapping users to roles (since one user can have multiple roles). Fields: `UserID` (FK), `RoleID` (FK). If we restrict one role per user, we can simplify and put a RoleID in the User table, but a join table gives flexibility (e.g. a user can be both a Manager and a Pharmacist).

- **AuditLog** – Records critical actions for accountability (especially for pharmacy where sensitive inventory like controlled substances may need tracking). Fields: `LogID` (PK), `UserID` (FK of the user who performed action), `Entity` (e.g. "Product" or "InventoryMovement"), `Action` (e.g. INSERT, UPDATE, DELETE), `RecordID` (the primary key of the record affected), `Timestamp`, `OldValue` (optional snapshot before change), `NewValue`. The AuditLog will be appended to on every significant change (especially admin-level changes like price updates, stock adjustments, user role changes, etc.). This provides a trail for security and compliance.

**Entity Relationships:** The schema enforces one-to-many and many-to-many relations via foreign keys and join tables as described. For example, **Product–Category** is one-to-many (one category, many products) [14], **SalesOrder–SalesOrderItem–Product** is a many-to-many between Product and SalesOrder resolved by the SalesOrderItem join. **User–Role** is many-to-many via UserRole. All many-to-many relationships have their own table so that the design is in 3NF (no multi-valued fields).

**ER Diagram:** Below is an ER diagram illustrating the main entities and relationships (PK = Primary Key, FK = Foreign Key):




*Entity-Relationship Diagram for the Inventory System (key tables, PKs, and FKs). One-to-many relations are indicated by "1" and "0..*". Many-to-many relations use join tables (e.g., SalesOrderItem, UserRole).*

*(Note: The actual CouchDB implementation will store these as JSON documents. We may not enforce foreign keys at the DB level in CouchDB, but our application logic will maintain these relationships. In CouchDB, we will likely use document IDs that incorporate entity and possibly use design documents or Mango queries to index fields like SKU or OrderDate for querying [14] [17] .)*

**Indexes:** We will create indexes on frequently queried fields to optimize performance: - **Products:** index on SKU and Barcode for quick product lookups by code. Also an index on CategoryID to list products by category. - **Customers:** index on Email or Phone to quickly find a customer by contact (useful for loyalty lookup at POS). - **Orders:** index SalesOrder by OrderDate (for date-range sales reports) and by CustomerID (to find a customer's purchase history). Index PurchaseOrder by OrderDate and SupplierID similarly. - **InventoryMovement:** index by ProductID+WarehouseID composite to compute current stock per product per location quickly. - **AuditLog:** index by Entity or UserID to allow filtering the audit trail (e.g., view all actions on Products or all actions by a specific user).

In CouchDB/PouchDB, indexes can be created via **Mango queries or map/reduce views** for these fields [18] . We'll define these so that queries (like "find product where SKU = X") are efficient even on large data sets, which is important for low-resource devices.

**Normalization and Denormalization:** The schema is largely normalized to 3NF to ensure data integrity (e.g. supplier info in one place, avoid duplicate entries) [14] . This prevents anomalies – e.g., if a supplier changes address, updating one record suffices. However, for performance and simplicity in a distributed setting, we will use **selective denormalization** [19] [20] : - Certain redundant fields may be kept in documents. For example, we might store `ProductName` and `CategoryName` within SalesOrderItem (at creation time) for quick display of order details without needing to join back to the product table. This way, reporting on a SalesOrder can show item names even if the product record is not immediately fetched. This is a deliberate duplication for ease of use. - We will store aggregate values like `TotalAmount` in SalesOrder (even though it can be derived by summing items) – this speeds up order listings and financial reports without having to sum each time. Similarly, `TotalCost` in PurchaseOrder. - The **Current Stock** per product per warehouse is essentially a denormalized aggregate of InventoryMovements. As mentioned, we may maintain a running quantity in each Product document (perhaps as a dictionary of WarehouseID → Quantity) updated on each movement. This avoids summing all movements for every stock query and provides instant stock availability checks [21] . Periodic reconciliation can ensure it matches the movement ledger.

Using denormalization in these targeted ways will **boost read/query performance for reporting** (which is important for dashboards and low-latency queries on slow hardware) [15], while still preserving normalized data for core relations. The trade-off is some extra logic to update the duplicated fields on each relevant change, but that is manageable in our controlled app code.

## 4. Advanced Features and Competitor Benchmarking

Our system is designed to include advanced inventory and retail features commonly found in leading solutions (like Zoho Inventory, Odoo, Vend, etc.), ensuring we remain competitive:

- **Multi-Warehouse & Location Tracking:** The application supports multiple warehouses (stock locations) from day one. Users can define multiple warehouse records (e.g., main store, secondary storage, future branch locations). All inventory transactions tie to a specific warehouse. Stock levels are tracked per warehouse, and **transfer orders** can be executed to move stock between warehouses (recorded as paired inventory movements). This is analogous to Zoho Inventory's multi-warehouse management feature which lets businesses manage stock across locations and track inter-warehouse transfers [16]. Our system will provide a UI to select the source and destination warehouse for a transfer and adjust inventory accordingly. Real-time tracking across warehouses will give a centralized view of total stock with breakdown by location (centralized control) [22] [23]. This helps the business optimize inventory distribution and prevent stockouts in one location while another has excess.

- **Batch/Lot Tracking with Expiry & Recall:** Especially important for a pharmacy, the system will include **batch number** and **expiry date** tracking for products. We will extend the Product model to optionally require batch tracking (yes/no). For batch-tracked products (e.g., medicines), every Purchase Order receipt will record batch IDs and expiry dates for the stock added. The SalesOrderItems will capture which batch was sold (via a dropdown or scan if using barcode labels on batches). This allows traceability: If a certain batch is recalled or expired, we can quickly query which customers bought that batch, or how many units remain. We will implement warnings for soon-to-expire products – e.g., highlight items within 30 days of expiry so staff can remove or discount them. This feature aligns with competitor offerings: Zoho Inventory supports serial/batch tracking to watch expiration dates [16], and many WMS systems provide FEFO (First-Expire-First-Out) stock rotation [24]. Our system will similarly support FEFO: when selling, prompt the user to sell the oldest batch first. In the event of a recall, an admin can query all SalesOrders containing the affected batch and all Inventory on hand of that batch. These capabilities are crucial for pharmaceuticals to ensure compliance and safety.

- **RFID and Barcode Integration (Future Phase):** While initially we rely on manual input for product codes, we will integrate **barcode scanning** in Phase 2 or 3. The POS interface will have a scan input field – using a USB scanner will input the barcode which the app catches to find the product (the SKU/barcode index makes this instantaneous). We will also generate barcode labels for products that lack them (using commodity barcode generation libraries) so even local suppliers' goods can be tagged. In a future phase, we aim to integrate **RFID** for faster audits and real-time tracking. For example, using RFID readers at the door can log inventory movements automatically, and handheld RFID scanners can speed up stock counts [5] [6]. This is forward-looking, but the design is RFID-ready: each product/batch could have an RFID tag ID stored, and InventoryMovements could be created by reading tags (with appropriate software interface). Integrating these technologies will improve accuracy and reduce manual errors in stock management [25].

- **Demand Forecasting & Reorder Automation:** To avoid stockouts and overstock, the system will include basic **demand forecasting** and automatic reorder point calculation. We will analyze historical sales data per product to identify patterns (daily/weekly sales rates, seasonal trends) [26] [27]. Initially, this could be a simple moving average or last-month vs same-month-last-year comparison to predict upcoming demand [27]. Using that, the system can compute a **recommended reorder point (ROP)** and **safety stock** for each product. The classic formula we'll implement: **Reorder Point = Lead Time Demand + Safety Stock** [28]. Safety stock will be calculated based on variability in demand and supplier lead times (e.g., using the formula: *(Max daily usage × Max lead time) – (Avg daily usage × Avg lead time)* as suggested by inventory management best practices [29] [30]). Users can input lead time for each product's supplier, and the system will suggest a reorder point. When stock falls below the ROP, an **alert/trigger** will notify the manager to reorder (and in future we can auto-generate a PurchaseOrder draft). This approach is similar to Zoho Inventory's "Reorder point" feature which notifies when stock is low [31]. We will provide a dashboard view listing items at or below their reorder level so the manager can quickly create POs. Over time, more advanced forecasting (like exponential smoothing or AI-based demand prediction) can be integrated, possibly leveraging TensorFlow.js offline or cloud ML when online.

- **Analytics Dashboards & KPIs:** We plan to include an **Analytics Dashboard** for the owner/manager that works offline (using local data) and gives insight into key performance indicators:

- Current stock valuation (quantity * cost) per category or overall.
- Fastest-selling products (by units/week).
- Slow-moving or dead stock (no sales in last N months).
- Sales trends over time (daily/weekly sales chart).
- Gross revenue, profit margins, etc.
- Supplier performance (e.g., number of late deliveries).
- Customer metrics (total sales per customer, top 10 customers).

These will be presented with charts and tables. We can use a JS chart library (like Chart.js or Recharts) to render graphs offline. Data will come from local PouchDB queries (which is feasible since data volumes are not huge for a small shop). For example, to get monthly sales, the system can query all SalesOrders and group by month. If performance becomes an issue, we might maintain some pre-aggregated stats (denormalization for reporting). The aim is to provide similar analytical power to what cloud systems do – e.g., Zoho has reporting and analytics built-in [31]. Our offline approach will mimic this so that even without internet, management can get actionable insights. When internet is available, we could also sync data to an external BI tool or Google Sheets for further analysis, but that's optional.

- **CRM Features & Loyalty Program:** Beyond inventory, we incorporate basic **CRM functionality** to nurture customer relationships:
- The system stores customer contact info and purchase history. Staff can search a customer to see past orders (useful for prescriptions refills or tailoring promotions).
- **Loyalty Program:** We will implement a points system where customers earn points on purchases (configurable rate, e.g. 1 point per ₦100). Points are stored in the Customer record and updated on each SalesOrder. The cashier interface can show available points and allow redemption (e.g. use points as discount). Different loyalty tiers (if any) can be managed via the LoyaltyProgram entity (e.g. Gold members earn 2x points).
- **Email/SMS integration:** Using the EmailTemplate data, when online the system can send out emails – e.g., a thank-you email after a purchase, or a promotion to all customers in the loyalty program. We will integrate with an email API (like Mailchimp, SendGrid or a simple SMTP server) in a later phase. For example, we could sync customer emails and purchase data to Mailchimp

for targeted campaigns, similar to Zoho Inventory's integration with Mailchimp for customer campaigns [32]. This keeps customers engaged with promotions ("Get 10% off on your next purchase!") which is valuable in retail.

- In addition, the system's **customer module** could be extended to support SMS alerts (if a customer wants an SMS when an out-of-stock item is back in stock, etc.). Given many Nigerian customers rely on SMS, integration with an SMS gateway (like Twilio or local providers) could be considered in the roadmap.

- **Role-Based Access Control:** Borrowing from enterprise inventory systems, we'll enforce roles and permissions. For instance:

- **Admin:** full access to everything (all CRUD, settings, user management).
- **Inventory Manager:** can create POs, adjust inventory, view reports, but maybe cannot edit users.
- **Sales Clerk/Cashier:** can create SalesOrders (process sales) and view products, but cannot edit product info or see cost prices/profit.
- **Pharmacist:** (if applicable) can view and dispense prescriptions, might have similar rights to cashier with some medical info access.

These roles and their permissions will be configurable, but we'll define sensible defaults. The UI will disable or hide unauthorized actions. On the data side, if a user without permission somehow initiates a restricted action, the system will block it (since it's offline, enforcement is mostly in the UI and later during sync we could have the server validate important constraints). For now, we trust the client enforcement as all users are internal and the threat model is low, but as we add remote API access, we'll tighten security checks on the server side too.

- **Audit Logs & Accountability:** Every critical change will generate an entry in the AuditLog with timestamp and user. For example, if a manager edits the price of a product or deletes an inventory movement (adjusting stock), the system logs who did it and when, with before/after values. This is important for trust and regulatory compliance (especially in pharmacy). The logs can be reviewed by admins – we'll have an "Audit Trail" screen where you can filter by date, user, or entity to inspect changes. For instance, if stock counts are off, an admin can check recent InventoryMovements and see if any manual adjustment was logged and by whom. The audit log will sync to the server as well (like any other data), ensuring a central consolidated log. This level of audit is often found in enterprise systems (ERPs) but is a key differentiator for our system at a small-business level.

- **API for Integrations:** The system will expose **RESTful APIs** to allow future integrations with other software. For example, if the business later launches an e-commerce website, the website could pull product availability via our API, or push online orders into the system. We plan to build a simple Node.js (Express) backend that interfaces with the CouchDB data to serve such APIs with proper authentication. Alternatively, since CouchDB itself has a REST API for all data [13], we could enable direct access to it from trusted integrations using API keys. However, wrapping with our own API layer allows adding business logic and security. We'll ensure common integration needs are covered:

- **Accounting software:** Export or sync sales and purchase data to an accounting system (QuickBooks, Xero, etc.). For instance, provide endpoints to fetch sales orders or perhaps automate daily summary exports. Zoho Inventory integrates with QuickBooks and Zoho Books [33]; we aim to allow similar data exchange so the retailer's accountant can get necessary info.

- **E-commerce:** If needed, expose product catalog and stock levels via API (with appropriate read-only keys) so an online shop could query if an item is in stock. Also accept incoming orders from e-commerce to decrement inventory.
- **Reporting tools:** Provide an endpoint or direct database access for external BI tools if the business wants to run more complex analytics in Excel/PowerBI, etc.

In summary, our advanced features cover the breadth of **inventory management** best practices: multi-location control, traceability (batch/RFID), smart reordering, analytical insights, customer engagement, and secure controls. We benchmarked against top solutions like Zoho Inventory – which offers multi-warehouse, batch tracking, barcode scanning, reordering alerts, integration to CRM/Email, etc. [16] [31] – and ensured our system incorporates those capabilities in an **offline-first** manner. This gives the small business modern, competitive tools while operating in a low-connectivity environment.

# 5. Technology Stack Justification

Our chosen stack balances modern web development with offline capability and simplicity:

- **Front-End: React** (with TypeScript) is used to build a rich client-side application. We will scaffold with `create-react-app` using the PWA template (`--template cra-template-pwa`) to get offline support out-of-box [34]. React is well-suited because it's component-based, allowing us to create reusable UI components (forms, tables, charts). It has a robust ecosystem (for state management, UI libraries) and our team is familiar with it. The PWA approach means the app can be installed on devices and work offline seamlessly. CRA's PWA template sets up the service worker and manifest for us [10] [11], which accelerates development. React will handle the UI/UX, form validations, and offline caching of interface. We'll likely use a UI toolkit like **Material-UI** or **Ant Design** for a clean responsive design, unless we opt for custom lightweight CSS to reduce footprint.

- **Local Database: PouchDB** (JavaScript NoSQL) on the client provides the core offline storage and sync capabilities. We choose PouchDB because it's specifically designed for offline-first apps: it runs in the browser, storing data in IndexedDB, and can synchronize with a CouchDB server effortlessly [1] [35]. The API is friendly (similar to a NoSQL document store) and supports powerful queries and map/reduce. PouchDB is proven in production for syncing across clients even with intermittent connectivity. With PouchDB, we can implement our local CRUD logic easily (e.g. `db.put()` to add a document, `db.find()` for queries). It also has plugins for things like full-text search and encryption if needed. The **legacy and community** support for PouchDB is strong (even in 2025, it's actively used, and its replication protocol is the foundation of many local-first systems [36]). This ensures reliability.

- **Server Database: Apache CouchDB** complements PouchDB as the server-side database. CouchDB speaks the same replication protocol, making it the natural choice to sync with Pouch. We chose CouchDB because it's a schema-less document database that supports multi-master sync, conflict detection, and RESTful access out of the box [18] [37]. It's light enough to run on a small VM or even a local server in the shop if needed. CouchDB's ability to run standalone or as a clustered service gives us flexibility for scaling. Additionally, CouchDB uses an **HTTP/JSON API** for all operations [13], which simplifies integration and debugging (one can `GET` and `PUT` data directly via HTTP). We can host CouchDB in a Docker container (see below) to simplify deployment. CouchDB's **MVCC architecture** ensures that writes don't lock the database [18], which is great for our scenario of concurrent sync – it will queue up revisions and handle merges. It also means even if multiple devices are syncing, CouchDB can handle it without data

corruption. Given our data volume and user count are small, a single CouchDB instance (or a small cluster) is more than sufficient performance-wise.

- **Search Library: Lunr.js** is chosen for implementing client-side search. Lunr is a lightweight JavaScript full-text search engine that can index JSON data and run queries all within the browser. We prefer Lunr because it requires no server, and thus works offline, aligning with our offline-first goal. Lunr supports features like tokenization, field-specific search, wildcards, and fuzzy search [3] [4], which means we can get a Google-like search experience for products. We will index product name, category, brand, etc., so that a user typing part of a name instantly gets matches ranked by relevance. Alternative search solutions like ElasticSearch are too heavy and online-dependent for our use; Lunr gives us just enough power with minimal overhead (~50KB lib). Also, Lunr can be updated incrementally when new products are added. (In the future, if we have performance issues with very large catalogs, we might consider **ElasticLunr** or **FlexSearch** which have similar concepts, but for our scale Lunr is ideal.)

- **AI/ML Tools:** For the "similar product" suggestions, if we decide to incorporate a machine learning approach, we can use **TensorFlow.js** or simple JS libraries for KNN. For example, TensorFlow.js could be used offline to run a small pre-trained model that embeds product text into vectors and finds nearest neighbors. This would be entirely in-browser. Alternatively, a library like **natural** (NaturalNode) or **compromise** for NLP could help with basic keyword matching. However, to keep resource usage low, we will likely start with non-ML algorithms as discussed (category-based or attribute similarity). If needed, the heavy lifting can be done during sync (e.g., server can compute similar items and tag them in the data).

- **Backend Server (for API & Logic):** We will use **Node.js** with Express (or Koa) to build any needed backend services. This backend is not strictly required for core operation (since Pouch/Couch can sync peer-to-peer), but it becomes useful for:

- Hosting the static PWA (if not using GH Pages).
- Providing a secured API layer to the data for external integrations or centralized logic (like sending emails, complex conflict resolution, etc.).
- Running background jobs (e.g., daily data backups, generating forecast data).

Node.js is chosen due to its JavaScript commonality (we can share code between client and server if needed, e.g. form validation logic or data models). Also Node has great libraries for sending emails, interacting with SMS APIs, etc. We can containerize this Node server as well.

- **Hosting & Deployment:** For simplicity and cost, the front-end PWA can be deployed as a **static website** (since after build it's just HTML/CSS/JS). For example, we could use **GitHub Pages** to host the app, which is free and can serve our PWA (though GH Pages doesn't support server-side code, it's fine for the client app). If we need dynamic hosting (for the Node API), we might use a small VM or a service like Heroku/DigitalOcean. The CouchDB server could run on a cloud VM (with a public IP or behind a VPN for the store). However, considering unreliable internet, we might also deploy CouchDB on a local machine in the store for LAN syncing among devices, and then have that machine sync to a cloud CouchDB when internet is available (to have an off-site backup). This architecture gives a local "hub" for offline operation across multiple devices on the same LAN (since PouchDB can sync over local network). It's a bit more complex, so initially we may just use one central CouchDB in the cloud and each client syncs whenever it can reach it (the trade-off being if internet is down, clients don't see each other's changes until back online, which we accept in early phase).

- **Docker:** We heavily utilize Docker for ease of setup and portability:

- **CouchDB in Docker:** We will use the official CouchDB Docker image [38] . This allows running CouchDB with a single command, isolating it from host configuration. For example, during development we can start CouchDB via Docker on localhost so developers don't need to install it manually. In production, the server deployment will likely also use Docker (and Docker Compose for orchestrating multiple services). We'll provide a **Docker Compose** file that can spin up CouchDB (and potentially the Node API and maybe a reverse proxy like Nginx if needed). This also makes scaling to a cluster easier – CouchDB can be clustered by spinning up multiple containers (though for a small business, a single node with backups might suffice initially).
- **Optional SQLite alternative:** The question mentioned SQLite + SQLSync as an alternative. SQLSync is an emerging solution (Rust-based) that wraps SQLite for offline collaboration [39] . While our main plan is CouchDB/PouchDB (mature and high-level), we acknowledge this alternative. SQLite could be embedded for local storage and then use a custom sync service to a central Postgres or MySQL. SQLSync specifically offers a way to sync SQLite across clients, aiming for similar offline-first behavior in a relational model [39] . The technology is still new, but in the future if a fully relational offline solution is desired, we could consider migrating to it. For now, sticking to Couch/Pouch is lower risk and faster to implement, given existing tools and our familiarity. Docker can also run a Postgres or other DB if needed for alternate setups, but that adds complexity.

- We will also Dockerize our Node server (if any) to ensure consistency across development and production. Developers can run `docker-compose up` to start the whole stack locally (CouchDB, perhaps a mock server, etc.), making onboarding easier.

- **State Management:** On the front-end, for managing application state (like current user, cached lists, etc.), we will likely use a library like **Redux** or the newer **React Context/Redux Toolkit** if the app state becomes complex. This isn't strictly stack, but part of tech choices. We want to ensure the app works offline with perhaps large lists in memory, so careful state management helps.

- **Other Libraries:**

- For forms: perhaps Formik or React Hook Form to handle validation elegantly.
- For date handling: Luxon or date-fns (to manage things like expiry dates, order dates).
- For authentication UI: maybe Firebase UI if we had online auth, but since likely local auth, we'll do our own.
- For charts: Chart.js or Recharts for dashboards.
- For barcode generation: jsBarcode or similar to print labels.

In summary, the chosen stack – **React PWA + PouchDB/CouchDB + Lunr + Node/Docker** – is well-aligned with an offline-first requirement. It leverages web technologies that can run on any device, ensures data sync and conflict handling via CouchDB's proven multi-master replication [37] , and remains lightweight enough for a small deployment. We avoid heavy enterprise software (no large SQL servers or heavy middleware needed on-premise – CouchDB is relatively low-maintenance). This stack gives us a modern app that works like a cloud SaaS but doesn't break when the internet does.

## 6. Development Workflow and Tools

To implement this system efficiently, we will use a modern development workflow with emphasis on prototyping, collaboration, and continuous delivery:

- **Project Management:** We'll follow an agile approach (likely Kanban or short sprints) given the small team. We will maintain a Trello or GitHub Projects board to track tasks: functional requirements breakdown, bug fixes, testing tasks, etc. Clear priorities will be set (Phase 1 features first, etc.).

- **Version Control:** The codebase will be managed on **GitHub** (private repo if needed). We will use Git for all code, with a branching strategy like *feature branches* merging into a *dev* branch, and stable releases merged into *main* branch. Code reviews will be done via pull requests on GitHub to ensure code quality and share knowledge among team members.

- **IDE & Coding Aids:** Developers will primarily use **Visual Studio Code** as the IDE, which is well-suited for our stack. We'll set up recommended extensions in the repo (like ESLint, Prettier, EditorConfig) so that code style remains consistent. VS Code also helps with debugging React and Node apps with its built-in debugger. We might include **settings.json** in the repo to standardize formatting on save, etc.

- **Wireframing/Design:** Before heavy coding of the UI, we'll create wireframes for key screens (product list, order entry, reports dashboard, etc.). We can use **Pencil Project** (an open-source wireframing tool) for quick mockups of forms and layouts. This helps in getting feedback from end-users early about the UI/UX. Alternatively, a web-based tool like **Figma** or **Webflow** could be used: Webflow in particular can create high-fidelity prototypes that stakeholders can interact with. However, given offline focus and simplicity, Pencil or even hand-drawn sketches scanned and shared might suffice initially. The main point is to nail down a simple, intuitive UI (especially for the POS screen where speed is crucial).

- **Continuous Integration (CI):** We will set up **GitHub Actions** for CI. A typical workflow is triggered on every push or PR:

- Install dependencies (cache them for speed).
- Run automated tests (unit tests for JS functions, possibly integration tests).
- Lint the code (using ESLint) and run type checks (TypeScript) to catch errors.
- Build the React app (ensure it compiles without error).

This ensures we catch issues early. If any step fails, the CI will flag it. This way, we maintain a healthy codebase that's always in a deployable state.

- **Continuous Deployment (CD):** For deployment, we can automate it with Actions as well. For instance, when code is merged into `main` (or a tagged release), an action can:
- Build the static files (`npm run build` for React).
- Deploy to GitHub Pages (using `peaceiris/actions-gh-pages` or similar GitHub Action to push the `build` folder to gh-pages branch) if we host frontend on GH Pages.
- Similarly, if we have a backend, we could build a Docker image and push to a registry or deploy to a server via SSH.

For example, a **GitHub Actions YAML** for deploying might look like:

```
name: Build and Deploy PWA
on:
  push:
    branches: [ main ]
jobs:
  build-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - run: npm ci
      - run: npm run build
      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./build
```

This would automatically push the latest build to GitHub Pages, making it available to users (assuming custom domain or direct GH Pages URL).

- **Development Server & Hot Reload:** During development, we'll use the React dev server (`npm start`), which provides live reloading when files change. This speeds up the dev cycle. For the backend (CouchDB), in dev each dev can either connect to a shared CouchDB dev instance or run one locally via Docker (e.g. `docker run -p 5984:5984 -e COUCHDB_USER=admin -e COUCHDB_PASSWORD=pass couchdb:3`). We will include a **docker-compose.yml** in the repo for dev that looks like:

```
version: '3'
services:
  couchdb:
    image: couchdb:3
    ports:
      - "5984:5984"
    environment:
      - COUCHDB_USER=admin
      - COUCHDB_PASSWORD=password
    volumes:
      - ./couchdata:/opt/couchdb/data
    # Optionally, mount a local.ini to configure CORS for dev
```

This lets a dev spin up CouchDB with one command. We will enable CORS in CouchDB for `http://localhost:3000` so that the React dev server can talk to it (CouchDB can be configured via `local.ini` or an admin API call to allow origin).

- **Prototyping & Testing Environment:** To demo the app quickly or test on actual devices, **ngrok** is a great tool. We can run the React dev server or a local build on a machine and use ngrok to

expose it over the internet temporarily (ngrok gives a public URL that tunnels to localhost). This is useful to show the client (pharmacy owner) a working prototype on their tablet without deploying to a server. We'll use ngrok in internal testing too, for example to test syncing: run CouchDB locally, expose it via ngrok, point two clients (maybe one in a VM or phone) to that URL and simulate multi-user sync in a realistic scenario.

- **Quality Assurance:** We will implement testing at multiple levels:

- **Unit Tests:** Use Jest (which comes with CRA) for testing pure functions (e.g. a function that calculates reorder point, or one that allocates batch inventory FIFO). These will be run in CI. For example, test that `calcReorderPoint(avgDaily, leadTime, safetyStock)` returns expected values under various scenarios.
- **Integration Tests:** Possibly using React Testing Library to test that components interact with the PouchDB store correctly. We might set up an in-memory PouchDB instance for tests or use a special test database and then verify that, say, adding a product via the form results in that product appearing in the list (simulate user clicks, etc.). These ensure our UI and data layer work together.
- **End-to-End Tests:** In later phases, we can use Cypress or Playwright to simulate a user's workflow in a headless browser. For example, a test that goes through the process of creating a sales order and checks that stock decreased and an audit log entry was made. We can also simulate going offline in such tests by disabling network and ensuring the app still works (this can be tricky in automated tests, but Cypress does allow stubbing network calls).
- **Performance Testing:** We will do manual performance testing on target devices. E.g., load the app with 5,000 products and simulate a search query, measure the time. Or try bulk importing 1,000 inventory movements and see sync time. If needed, we might write scripts to generate dummy data and measure sync throughput. CouchDB can handle thousands of docs easily, but we want to ensure the UI remains snappy.

- **User Acceptance Testing (UAT):** We will involve a couple of end users (perhaps the store staff) to try the system in a controlled environment. They will run through typical tasks (add products, create sales, simulate network outages) and give feedback on usability and any bugs. Because the environment is unique (low connectivity), having them test in a realistic setting is important. We'll incorporate their feedback especially on UI clarity and offline behavior (e.g. if sync delays cause any confusion, we might add a small "sync" status indicator icon, etc.).

- **Documentation:** We will document setup and usage clearly. A **README** in the repo will explain how to run the app in dev, how to deploy, environment variables (like CouchDB URL, credentials). We'll also prepare an **user guide** (in Markdown or PDF) for the business users that covers basic operations (with screenshots): how to install the PWA on their device, how to perform key tasks, and troubleshooting tips (e.g., what to do if a device hasn't synced in a while). Keeping documentation up-to-date will be part of our workflow (possibly using GitHub Wiki or a docs folder versioned with code).

- **Folder Structure:** We will organize the codebase for clarity:

- **frontend/** (or just root if it's only front-end):
    - **public/** – static files (including `index.html`, the manifest.json for PWA, icons).
    - **src/** – React source code:
    - **components/** – Reusable UI components (buttons, form inputs, modals).
    - **pages/** – Page-level components (ProductListPage, OrderPage, DashboardPage, etc.).

    - **db/** – PouchDB related code (e.g. `db.js` that initializes PouchDB, sets up sync with CouchDB and perhaps contains functions like `syncNow()` or conflict handlers).
    - **state/** – Redux slices or context providers if using global state management.
    - **services/** – Utility modules, e.g. `searchService.js` for Lunr indexing and search, `forecast.js` for demand calculation, etc.
    - **styles/** – CSS or SASS files if separate from components.
    - **App.js / index.js** – Main app bootstrap.
    - **serviceWorkerRegistration.js** – (from CRA PWA template) to register the SW.
    - **setupTests.js** – (for configuring Jest).
    - **package.json**, etc.
- **backend/** (if implementing a Node backend):
    - **index.js** (entry to start Express server)
    - **routes/** – define API endpoints
    - **models/** – define data models or CouchDB access logic (could use nano or any CouchDB client in Node)
    - **services/** – e.g. emailService to send emails via SMTP
    - Perhaps **scripts/** – like a script to trigger daily forecast calculation.
    - **Dockerfile** for backend.
- **infrastructure/**:
    - **docker-compose.yml** (for dev or prod)
    - **couchdb/** – maybe config files like `local.ini` if we need to customize Couch (for enabling CORS or setting cluster).
    - **nginx/** – if we add a reverse proxy for serving the PWA and API on same domain, its config would go here.
- **.github/workflows/** – CI/CD pipeline YAML files.
- **docs/** – any documentation or design diagrams, or maybe the ER diagram image.

This structure keeps front-end and back-end separate, making it easier to deploy or scale them independently.

- **Collaboration and Communication:** We'll have regular check-ins (could be daily standups or at least weekly meetings) to discuss progress and blockers. Given potentially different locations, using Slack/Teams for quick communication and screen-sharing for demos will be done. We will also maintain a changelog and use semantic versioning for releases (e.g. v0.1, v0.2 during development, v1.0 for the first production release).

By using these tools and practices (VS Code for development, GitHub for CI/CD, Docker for environment parity, etc.), we aim to ensure a **smooth development process**, quick prototyping for stakeholder feedback (via ngrok or GH Pages deployments), and a robust testing regimen so that by the time we deploy, the system is reliable. Automating the build/test/deploy as much as possible frees us to focus on coding features and fixing issues early, rather than firefighting deployments later.

# 7. Proof-of-Concept Roadmap & Milestones

We propose a phased implementation over several months, with clear milestones and deliverables at each phase to track progress:

- **Phase 1: Schema Design & Offline CRUD POC** (Estimated 4–6 weeks)
  *Scope:* Set up the fundamental data structures and get a basic app working offline.
  **Tasks & Milestone Deliverables:**

- **Database Schema & Setup:** Finalize the ER schema and document structure. Set up CouchDB (Docker) and PouchDB integration. Verify that the sync works on a simple example (e.g., add a doc on one device, see it appear on another after sync). This includes configuring CouchDB admin user and enabling CORS for development. *Deliverable:* A short document defining the schema (likely an output of section 3's work) and a running CouchDB instance accessible to the team.
- **React App Initialization:** Initialize CRA with PWA template. Set up routing (e.g., using React Router) for main pages (Products, Orders, etc.). Implement PouchDB in the app ( `db.js` ) and on app load, establish sync with the remote CouchDB (with proper URL/credentials). Include basic error handling for sync (e.g., log if sync fails due to no connection, and keep retrying).
- **CRUD for Core Entities:** Start with **Products** and **Customers** as they are simplest. Build a form to add/edit a product (fields: name, SKU, etc.) and store it in PouchDB. Build a product list page to display stored products (data coming from local DB). Ensure that these operations work offline (test by disconnecting network and adding products, then reconnect and see them sync to server). Implement validation (e.g., prevent duplicate SKU). Similarly, a form and list for Customers. For now, UI can be minimal (unstyled or basic HTML table).
- **Sales Transaction Basics:** Create a very basic SalesOrder entry screen: ability to add line items (select a product from a dropdown and quantity). For Phase 1, it could be simplified (maybe not handling inventory deduction yet, just recording the order header and items). The focus is on proving multiple related docs can be created and synced.

- **Offline Capability Demo:** Install the PWA on a test device and show that it works without internet (thanks to service worker caching and PouchDB). For example, turn off wifi, add a new customer, turn on wifi, and show it syncs to server (perhaps by checking CouchDB Fauxton or syncing another device). Also demonstrate app refresh offline (that the service worker serves it).
  **Milestone Checkpoint:** By end of Phase 1, we expect a **working prototype** where users can manage products and customers offline, and data syncs when online. This will be demonstrated to stakeholders to gather feedback on data fields and basic UI. We'll also likely have some initial test results (e.g., how conflict might be handled if two devices added the same product with different details – we can demonstrate Pouch's default conflict handling and note the need for custom handling if any).

- **Phase 2: Search, Suggestions & Inventory Features** (Estimated 4 weeks)
  *Scope:* Enhance the system with search functionality, product recommendations, and implement inventory tracking (stock in/out, batch handling).
  **Tasks & Deliverables:**

- **Implement Lunr.js Search:** Integrate Lunr. On the product list page (and any place where product search is needed, e.g. adding items to an order), add a search bar. Index product data (maybe on app startup or whenever products change). Support partial matches and maybe fuzzy search. Deliverable is the ability to type part of a name or SKU and get a filtered list of products instantly. We'll test search speed with ~1000 products to ensure it's fast on a typical device.
- **"Similar Product" Suggestions:** On the product detail page (or as a dropdown when a product is out-of-stock in an order), show a list of similar products. Implement a simple algorithm (like same Category or similar name). If we have time and it's lightweight, incorporate a more advanced approach (e.g. vectorize names by word frequency and compute cosine similarity). The deliverable is that for a given product X, the UI shows e.g. "You may also consider: [Product Y], [Product Z]" which are alternatives (perhaps same category or if we had user purchase patterns, those frequently bought together). We'll verify that for a test category (say analgesics), products do suggest each other.

- **Inventory Movements & Stock Management:** Now integrate the inventory dimension:
    - Create a **Stock page or module** that shows current inventory per product per warehouse. This might involve calculating from movements (for now, since we may not have had movements yet, implement the logic).
    - Implement **Purchase Order receiving**: e.g., add a form to create a new PurchaseOrder with items (supplier, date, items with quantity). On marking it as "Received", the system should create corresponding InventoryMovement records (one per item, +Quantity). Update the product's stock accordingly.
    - Implement **Sales Order stock deduction**: when finalizing a SalesOrder, create InventoryMovements for each item (-Quantity). If using batch tracking, prompt for batch selection for each item sold (from available batches in inventory). Ensure that stock levels (maybe a field in Product or a separate stock collection) decrement. Possibly prevent selling if stock isn't available (or at least warn).
    - Add a simple **Adjust Stock** feature: allow a manager to manually create an InventoryMovement (e.g., for corrections or write-offs). For instance, if a bottle broke, they do an adjustment of -1 for that product.
    - At this stage, incorporate **Batch/Lot tracking**: For products marked as batch-tracked, the PurchaseOrder receiving form will include Batch No. and Expiry for each line. Store that in InventoryMovement or a separate Batch table. Ensure that when selling, if multiple batches exist, the sale deducts from the correct batch (probably the one with nearest expiry first by default – FEFO). Keep track of batch balances. This is complex, but we can simplify initial implementation by considering each batch as a sub-entity of the product.
    - *Deliverable:* The system should maintain correct stock counts locally. We'll test a scenario: Add 100 units via PO, sell 2, adjust -1, and verify the system shows 97 remaining. Also ensure this syncs – another device after sync should show the same stock. We'll also test that selling an item with no stock produces an alert and is prevented (or allowed and results in negative stock if we choose that route, but better to prevent oversell).
- **User Roles Implementation:** Introduce the concept of roles in the app (if not already). At least create an Admin user and a Clerk user and demonstrate that certain features (like adjust stock or view cost prices) are only visible to Admin. We might implement a basic login screen at this phase (even if just a simple username/password checked against a user document in PouchDB). If full auth is too much, we can simulate it by having a toggle for role in the UI for testing. But ideally, implement CouchDB authentication: enable CouchDB's `_users` database and use `db.login()` from PouchDB so each user has their own replication feed. This might be stretch for this phase, but it's noted.

- This phase's end deliverable is a **functionally complete inventory management core**: you can receive stock, sell stock, track the inventory levels, search products easily, and get suggestions for similar products. Basically, at this point the system could run in the store for basic daily operations (maybe not with all polish, but functionally).

- **Phase 3: Connectivity & Resilience Testing** (Estimated 2 weeks, overlapping with Phase 2 & 4 as needed)
  *Scope:* Rigorously test and refine the offline-online sync behavior, conflict resolution, and system performance under stress.
  **Tasks:**

- **Offline/Online Sync Tests:** Simulate various network conditions. For example, use browser devtools to throttle network or go offline to see how the app behaves. Test live replication: start two browser sessions as two "users", perform actions offline on both, then bring them online and observe if all changes merge correctly on CouchDB. Specifically test conflict scenarios: e.g.,

User A and User B both edit the same product name offline, or both sell from the same product quantity offline such that stock might go negative. Verify how conflicts are handled: by default one will "win" (latest rev) [7]; ensure the conflict is either resolved by our logic or at least doesn't break the app (no crashing, and the losing conflict could be flagged for manual resolution). We might implement a simple conflict handler in PouchDB that logs conflicts [40] and resolves them by a rule (like keep the one with max timestamp for non-stock, but for stock changes perhaps sum them).

- **Performance Tuning:** Populate the app with sample data to mimic a realistic scenario: e.g., 1000 products, 1000 customers, 5000 past sales orders, etc. Ensure the UI can handle this (list virtualization for large lists might be needed, we can implement lazy-loading or pagination). Measure sync time for, say, 1000 new docs – CouchDB replication might take a few seconds, which is fine. Ensure that writing to local DB is instant (should be, since IndexedDB can handle that volume). Optimize any slow query by adding an index or tweaking code. For search, ensure indexing 1000+ items is done maybe in a web worker thread to not freeze the UI. We'll incorporate such optimizations here.
- **Resource Usage on Low-End Devices:** If possible, test the app on the actual hardware or an emulator (e.g., an old Android tablet or a low-spec Windows laptop). Monitor memory usage (e.g., Chrome task manager) after loading the data. If memory is high, consider strategies (like not keeping too many docs in memory at once). Also test adding a PWA to home screen on Android and ensure it launches properly.
- **User Feedback Incorporation:** By this phase, we likely have feedback from initial users (if we did a pilot). Address any major usability issues (e.g., "it's hard to switch warehouse context" or "the text is too small on tablet"). Also refine the UI styling now that core is working – add loading indicators for sync, make sure forms are mobile-friendly (use proper input types, big buttons, etc.). Make the app as robust and user-friendly as possible.

- *Deliverable:* A **refined, well-tested system** ready for broader use. We should be confident that data won't be lost even if devices go on/off network, and that the app can handle the expected workload. We will produce a **Test Report** summarizing the scenarios tested (including conflict cases and how they are resolved, sync delays handled, etc.). This phase ensures no surprises in production due to connectivity issues.

- **Phase 4: Advanced Features, Integrations & Final Touches** (Estimated 4 weeks)
  *Scope:* Implement remaining advanced features (loyalty, analytics dashboard, notifications, external integration hooks) and prepare for production deployment.
  **Tasks:**

- **Analytics Dashboard:** Build the dashboard page with summary components (charts and tables for KPIs as described in section 4). This likely involves writing some aggregation queries against PouchDB data. We might utilize MapReduce in CouchDB to pre-aggregate sales by month, etc., and then simply fetch that. Alternatively, we compute on the fly for the prototype. We ensure this works offline with data available. We may incorporate a library like D3 or Chart.js for visuals. Make sure to keep it simple for now (maybe a few line charts and a table of top products).
- **Loyalty Program & CRM:** Finalize the loyalty points logic: e.g., each SalesOrder, if a Customer is attached, the system calculates points earned (maybe configurable per loyalty program or a default rate) and updates the Customer's point balance. Also create a simple UI for redeeming points (e.g., in SalesOrder allow applying points as discount). Implement any tier logic: if we have Gold/Silver programs with perks, ensure customers can be assigned and that maybe the receipt or UI indicates their tier. Also, consider an **email integration**: perhaps use a Node script or third-party integration to send a "welcome email" when a new customer is added (this could be triggered when online, by checking a flag and then using an API like Mailchimp or a local SMTP).

At minimum, allow exporting customer emails so the business can do marketing manually if needed.

- **Notifications & Alerts:** Add in-app alerts for certain triggers: e.g., if a product stock falls below reorder point (we can show a notification badge on the Dashboard or an alert on login). If we implement background sync via service worker, we could even send a push notification to the manager's device for low-stock alerts (requires some web push setup; might be deferred). Also, an alert for soon-to-expire batches: the system should list batches expiring within say 30 days so staff can act (this could be on the dashboard or a separate report).
- **Barcode Printing and Scanning Utility:** If not done earlier, provide a way to print barcodes for products (maybe generate a PDF of labels for a chosen set of products). Also test a barcode scanner with the system to ensure the input is captured in the search or sales form correctly (usually scanners just input as keyboard, so it should work if focus is in the search box).
- **External API Endpoints:** Build out a few key REST API endpoints on the backend to demonstrate integration capability:
    - e.g. `GET /api/products` (with optional filter by updated_since) to get product data – for an e-commerce site to consume.
    - `POST /api/order` to insert a new sales order – could be used by an online store checkout to decrement inventory.
    - Secure these with an API key or basic auth for now.
    - Document them (Swagger or README) for future developers.
- **Audit Log UI:** Create an admin-accessible view to see the AuditLog entries. Perhaps allow filtering by date or user. Ensure that every sensitive action we identified is actually logging an entry (review code to add logging on those events if missed). This gives the owner confidence in monitoring the system's use.
- **Final UI/UX Polish:** At this stage, refine the styling and usability details:
    - Apply a consistent theme (colors, typography).
    - Ensure responsive design for tablet vs desktop (use CSS flexbox/grid or a UI framework grid).
    - Add proper form error messages, confirmations (e.g., "Are you sure you want to delete this?" prompts).
    - Localization, if any needed (likely not, English-only is fine for now as requested).
    - Setup app icons and manifest properly so that the installed PWA has a nice icon on device and launches full-screen. Test on Android adding to home screen.
    - Set up a fallback or refresh strategy for the service worker (since CRA's service worker can sometimes cache old versions; we might implement a "New version available, click to update" prompt using serviceWorkerRegistration as per CRA docs).
- **User Training & Docs:** Prepare training materials (how-to guides, perhaps a short video demonstration) for the staff. Possibly run a small training session at the client site or via a call, going through common tasks. Incorporate any last-minute feedback from this.
- **Soft Launch:** Before full production, do a pilot run for a day in the store parallel to their existing system (if any). Use real data and see if any issues arise (especially around speed at checkout, etc.). This is the final validation.

- *Deliverable:* The end of Phase 4 yields **Version 1.0** of the system, feature-complete and polished. We will have deployment artifacts ready: Docker images or URLs for the app, and a CouchDB instance running with initial data loaded. We will also deliver a **Milestone Report** summarizing all implemented features and any pending backlog items.

- **Phase 5: Deployment & Post-Launch Support** (Estimated 2 weeks post-launch for monitoring and quick fixes)
  *Scope:* Deploy the system live in the pharmacy/grocery and closely monitor initial usage, fixing

any issues.
**Tasks:**

- **Production Deployment:** Set up the production environment. Likely:
  - CouchDB running on a cloud server (ensure domain or IP reachable by clients, enable SSL). Alternatively, CouchDB on a local machine at the shop + a replication to cloud for backup.
  - Serve the PWA either via GitHub Pages (if internet is stable enough to load it; after first load it's cached for offline) or host it on the same server (e.g., using Nginx to serve static files and maybe proxy to CouchDB/API). We'll configure the app to point to the correct CouchDB URL (perhaps using an environment variable at build time).
  - Ensure DNS and SSL: e.g., if domain is inventory.myshop.com, point it accordingly. CouchDB may need CORS allowed for that domain.
  - Create initial user accounts (admin etc.) for the staff.
  - Import any existing product list the business has (maybe via CSV upload into the system) so we start with data.
- **Go-Live and Monitoring:** On launch day(s), have a developer on standby (either on-site or remotely) to support. Monitor the CouchDB logs for replication issues or any 500 errors. Also monitor performance on clients – if any device struggles, be ready to suggest using a lighter browser or such.
- **Gather Metrics:** We can implement a simple logging of app usage (maybe not full analytics since offline, but perhaps log events to an "AppUsage" document that syncs when online). This can tell us how often sync is happening, or if any errors occur. Or use Sentry (it can buffer errors and send when online) for runtime error tracking.
- **Post-Launch Review:** After a week of usage, meet with the business to evaluate: Is the app meeting their needs? Are there pain points or any feature gaps? This may generate a list of improvements or Phase 6 features (like more reports, or maybe integration with a receipt printer, etc.).
- **Hand-over:** Train a person in the business to be the system administrator (managing user accounts, basic troubleshooting like resetting a device's data if needed). Provide all documentation and ensure they know how to reach support.

This roadmap provides a structured path. The timeline in summary: - *Month 1:* Core data and offline CRUD prototype. - *Month 2:* Add search, suggestions, inventory transactions (stock in/out). - *Month 3:* Testing and hardening (overlap with Month 2 & 4 tasks). - *Month 4:* Advanced features (dashboard, loyalty, integration) and polish. - *Month 5:* Buffer for any delayed tasks, user testing, and launch prep. - *Month 6:* Deployment and initial support.

Throughout these phases, **testing strategies** will be continuous: - We will write unit tests alongside features (Phase 1 and 2 should include tests for business logic like stock calculations). - Integration tests in Phase 2/3 for critical flows (create order -> stock changes). - UAT in Phase 3/4 with actual users using near-final system. - Performance tests in Phase 3 (simulate low connectivity by toggling network to ensure system behaves). - Security tests in Phase 4 (try to break role permissions, ensure a cashier account can't access admin functions by direct URL, etc.).

Milestones are evaluated with the team and stakeholders to decide if we move to next phase or need additional tweaks. This staged approach mitigates risk by delivering incremental value and allowing adjustments based on feedback (for example, after Phase 1, we might realize some extra fields are needed or UI adjustments, which we can incorporate early).

## 8. Scaling and Migration

While the initial deployment is for a single small business, we plan for the future to ensure the system can scale and adapt as needed:

- **Scaling Up Users and Data:** The design using CouchDB multi-master replication inherently supports adding more users/devices easily – each simply syncs with the server and gets the latest data. For ~5-10 concurrent users we have now, this is fine. If the business grows (say 50 users or multiple store branches), we can **cluster CouchDB** for better throughput and availability. CouchDB 3.x supports clustering/sharding; we can deploy a cluster of 3 nodes so reads/writes are distributed, and one node failure doesn't bring the system down [13]. PouchDB will work with a cluster just as with a single node (through a load-balancer or using CouchDB's built-in clustering port). We may also consider separating databases by store or region if multi-branch (CouchDB sync can be filtered, so each store's devices sync only their subset, while HQ could sync all – this is a way to partition data as scale increases).

- **Microservices & Modularization:** As features expand, we might split the application into microservices for better maintainability. For instance, we could have separate services for **Inventory Management**, **Orders/Checkout**, **CRM/Loyalty**, each with their own database (or their own CouchDB database or SQL database as appropriate) and API. They can communicate via APIs or a message bus if needed. In the current architecture, everything is in one CouchDB; but CouchDB can handle multiple databases – we could put customers and loyalty in a separate DB from products and inventory to reduce replication overhead if needed (e.g., a tablet used only for inventory counts might not need customer data).

- Docker Compose will aid this transition: we can define additional services and link them. For example, an `order-service` (Node.js) that processes online orders separate from the PWA front-end.

- Domain-driven design principles will guide how we break things out: e.g., **Domain 1: Inventory** (products, warehouses, movements, POs), **Domain 2: Sales** (sales orders, customers, loyalty), **Domain 3: Administration** (users, roles, audit). If the system becomes a product offered to multiple businesses, we'd also consider a multi-tenant architecture (each tenant with their own DB or partition).

- **Migration to Alternative Backend (if needed):** If one day the decision is made to move to a purely relational or another system (perhaps due to integration requirements with other enterprise software), we will plan migration:

- **Data Export/Import:** Because our data is neatly structured, we can write scripts to export from CouchDB to CSV or directly to a SQL database. E.g., use CouchDB's `_all_docs` API to dump JSON, then map to SQL inserts. We might create an ETL process to migrate to Postgres or MySQL if offline capabilities become less crucial and relational consistency is desired. However, note that moving away from CouchDB would lose the seamless offline sync, so more likely we stick with Couch or use Couch as a sync layer to another DB (some projects sync CouchDB to Postgres for reporting [41]). Indeed, we might consider using a tool like **couch2pg** to continuously replicate CouchDB data into Postgres for advanced querying (the Community Health Toolkit uses this pattern [41]).

- **Hybrid Sync (SQLite option):** Another possible migration path, as mentioned, is adopting **SQLite on the client with a sync mechanism**. SQLSync (Rust) is one approach; there's also

**Cloudant** (IBM's hosted CouchDB) or PouchDB's own adapters to SQL if any. If in the future we wanted to leverage a full SQL engine locally, we might embed something like **SQL.js** (SQLite compiled to webAssembly) as the local store. But at present, PouchDB covers our needs and migration is not expected unless requirements change drastically.

- **Clustering and High Availability:** For ensuring the system can handle bigger loads or critical uptime (e.g., if this is expanded to a chain of stores or sold as a product):

- CouchDB clustering, as mentioned, would be configured (we could use Docker Compose to spin up multiple CouchDB nodes and link them in a cluster config, ensuring data is replicated between nodes). This gives fault tolerance and read scaling.
- We'd also use **load balancers** for the PWA/API if needed (for example, host the front-end on a CDN or multiple edge locations for faster access if many users in different regions).

- Caching: If some data (like product list) gets large and read-heavy, we might put a cache layer (like Redis or in-memory cache) for the API. But since the PWA mostly reads from local, caching is inherently handled client-side.

- **Domain Name and HTTPS:** For production, we'll ensure a proper domain setup. Perhaps something like `inventory.mypharmacy.com` for the web app. We'll get an SSL certificate (likely via Let's Encrypt) so that the app and CouchDB connections are secure (CouchDB behind a reverse proxy with SSL). This is important as browsers will require secure context for service workers and some PWA features. We'll plan DNS such that if the app is to be moved or scaled, we can adjust the IPs behind the domain without changing the PWA configuration (the PWA might have the server URL baked in, so might not matter if we keep same domain for CouchDB).

- **Data Backup and Archiving:** As data grows (years of transactions), we should have a strategy:

- **Backups:** Regular CouchDB backups (e.g., daily dump or continuous replication to a backup CouchDB). Because each PouchDB client actually has the full dataset too, they act as implicit backups, but we should not rely on that. We'll have a scheduled backup of the CouchDB container volume (could use a cron job or service to dump to file or replicate to secondary DB).

- **Archiving Old Data:** After X years, performance might suffer or the PWA might slow down with very large local data. We can mitigate by archiving old records: e.g., sales orders older than 5 years might be moved to an "Archive" database that is not replicated to every device (or at least not to the cashier's device). This keeps active data small. If needed, the manager can query the archive through an admin interface or by temporarily syncing that database. This kind of partitioning can be planned as the business grows.

- **Multi-Business Support:** Currently designed for one business, but if we were to provide this as a solution to multiple clients, we'd need multi-tenancy. Easiest would be separate CouchDB databases (or even separate CouchDB instances) per client. The front-end could be reused, just pointing to different DBs based on login or subdomain. This is beyond current scope but worth keeping in mind – e.g., ensure no hard-coding of one database name so that we can deploy another instance for another pharmacy easily.

- **Continued Improvement:** We'll keep the system updated with latest tech (ensuring PouchDB and dependencies are up-to-date for security). Also as web standards improve (e.g., better offline APIs or maybe browser native DBs), we can adopt them.

- **Migration to Newer Tech:** By 2025+, there are also alternative **local-first frameworks** (like Replicache, Firebase (online), or even leveraging service workers as data servers). We stick with Couch/Pouch because it's stable and proven. But we'll monitor if new libraries (maybe SQLSync or others) become stable, which could influence future versions. Our modular design (separating data logic in services) means we could swap out the storage layer if needed with moderate effort (for instance, replace PouchDB with direct IndexedDB calls or another store, and implement a custom sync – not likely needed, but possible).

In conclusion, our deployment will start small but is built on technologies that can **scale vertically and horizontally**: add more users, more data, or even more modules, without fundamental changes. By using Docker and standard protocols, migrating across environments (dev to prod, or to new servers) is straightforward. And by keeping the architecture modular (client-server decoupling and clearly separated concerns), we can evolve parts of the system (like upgrading to a microservices architecture or integrating with new systems) without a complete rewrite.

The result is a **deployment-ready, scalable inventory system**: one that meets the immediate needs of a small Nigerian retail pharmacy for offline operations, while also laying the groundwork for future expansion, integration, and long-term maintainability. Each component from the UI to the database has been chosen with this balance in mind, and our plan includes the necessary steps, code snippets, and tools to implement and deliver this system successfully.

**Sources:**


1. Knowaloud Media – *PouchDB for offline-first web development* (describes PouchDB's offline capability and sync with CouchDB) [1] [35]
2. PouchDB Official Guide – *Conflict Resolution* (explains how CouchDB/PouchDB handle conflicts with MVCC, default winner selection) [7] [8]
3. Lunr.js Documentation – *Searching with wildcards and fuzzy queries* (demonstrates Lunr's support for partial matches, wildcards, etc., enabling client-side search) [3] [4]
4. Zoho Inventory Features – *Inventory management capabilities* (shows that leading inventory systems support multi-warehouse, batch tracking, barcode, reordering alerts) [16] [31]
5. Solid Innovation Blog – *Demand Forecasting Tip* (recommends analyzing historical SKU data with moving averages for forecasting demand) [27]
6. ShipBob Inventory Guide – *Reorder Point Formula* (defines ROP = demand during lead time + safety stock) [28]
7. Solid Innovation Blog – *RFID for Inventory Accuracy* (highlights use of RFID/barcode to improve inventory tracking and audits in multi-warehouse scenario) [5] [6]
8. CodiLime Tech Blog – *Normalization vs Denormalization* (discusses using 3NF for data integrity and denormalization for fast reads in reporting) [14] [19]
9. DEV Community – *CouchDB Offline-First with Docker* (outlines CouchDB features: clustering for availability, multi-master replication for offline sync) [18] [37]
10. Medium (Michael Fatemi) – *CRA PWA Setup* (instructions for enabling service worker in Create React App to get "offline-first" behavior) [34] [11]


---


[1] [35] PouchDB for offline-first web development | by Knowaloud Media | Medium
https://medium.com/@knowaloud/pouchdb-for-offline-first-web-development-1f3cbd691a38

[2] [12] PouchDB: Empowering Offline-First Web and Mobile Applications | by Stephen O. Roy | Medium
https://medium.com/@stephenoroy/pouchdb-empowering-offline-first-web-and-mobile-applications-afe501a02cee

3 4 Searching : Lunr
https://lunrjs.com/guides/searching.html

5 6 22 23 25 26 27 5 Top Tips for Multi-Warehouse Inventory Management | Solid Innovation
https://solid-innovation.com/blogs/news/tips-for-multi-warehouse-inventory-management?srsltid=AfmBOor1yQ41Xfa2RxlHlwuEHaVMCsCEBGVAe-EhRaYOV6y3vyUch2do

7 8 9 40 Conflicts
https://pouchdb.com/guides/conflicts.html

10 11 34 Turn your Create React App into a Progressive Web App in 100 seconds - DEV Community
https://dev.to/myfatemi04/turn-your-create-react-app-into-a-progressive-web-app-in-100-seconds-3c11

13 18 37 38 CouchDB: Offline-first multi-master synchronization using Docker and Docker-compose - DEV Community
https://dev.to/animusna/couchdb-offline-first-with-multi-master-synchronization-using-docker-and-docker-compose-293e

14 15 17 19 20 21 Normalization vs. Denormalization in Databases
https://codilime.com/blog/normalization-vs-denormalization-in-databases/

16 31 32 33 Complete inventory management features | Zoho Inventory
https://www.zoho.com/us/inventory/features/

24 Batch Inventory Management System - The Retail Exec
https://theretailexec.com/logistics/batch-inventory-management-system/

28 29 30 Reorder Point Guide: Formula + How to Calculate 3 ROPs
https://www.shipbob.com/blog/reorder-point-formula/

36 Offline-First with CouchDB and PouchDB in 2025 - Hacker News
https://news.ycombinator.com/item?id=43850550

39 SQLSync - llimllib notes
https://notes.billmill.org/databases/sqlite/SQLSync.html

41 CHT Sync Setup with Docker - Community Health Toolkit
https://docs.communityhealthtoolkit.org/hosting/analytics/setup-docker-compose/