Grocery POS System - Comprehensive Project Outline

Project Name: GroceryPOS Pro
Version: 1.0 (Beta Phase Planning)
Date: January 2026
Business Model: Yearly Subscription Service
Target Users: Retail grocery store operators and staff

Table of Contents

---

1. Executive Overview

Project Objective

Develop a comprehensive Point of Sale (POS) system specifically designed for grocery retailers, offering billing automation, inventory management, sales analytics, and subscription-based management through a modern, intuitive web interface.

Key Features

- **Real-time Billing:** Fast checkout with barcode scanning support
- **Inventory Management:** Live stock tracking with low-stock alerts
- **Sales Analytics:** Comprehensive reporting and business intelligence
- **Multi-store Scalability:** Support for multiple store locations (future phase)
- **Subscription Management:** Yearly subscription with customer authentication
- **User Management:** Role-based access control (Admin, Cashier, Manager)

Target Market

- Small to medium-sized grocery stores
- Multi-store retail chains
- Quick-service retailers with inventory needs

2. Technology Stack

Recommended Technology Stack

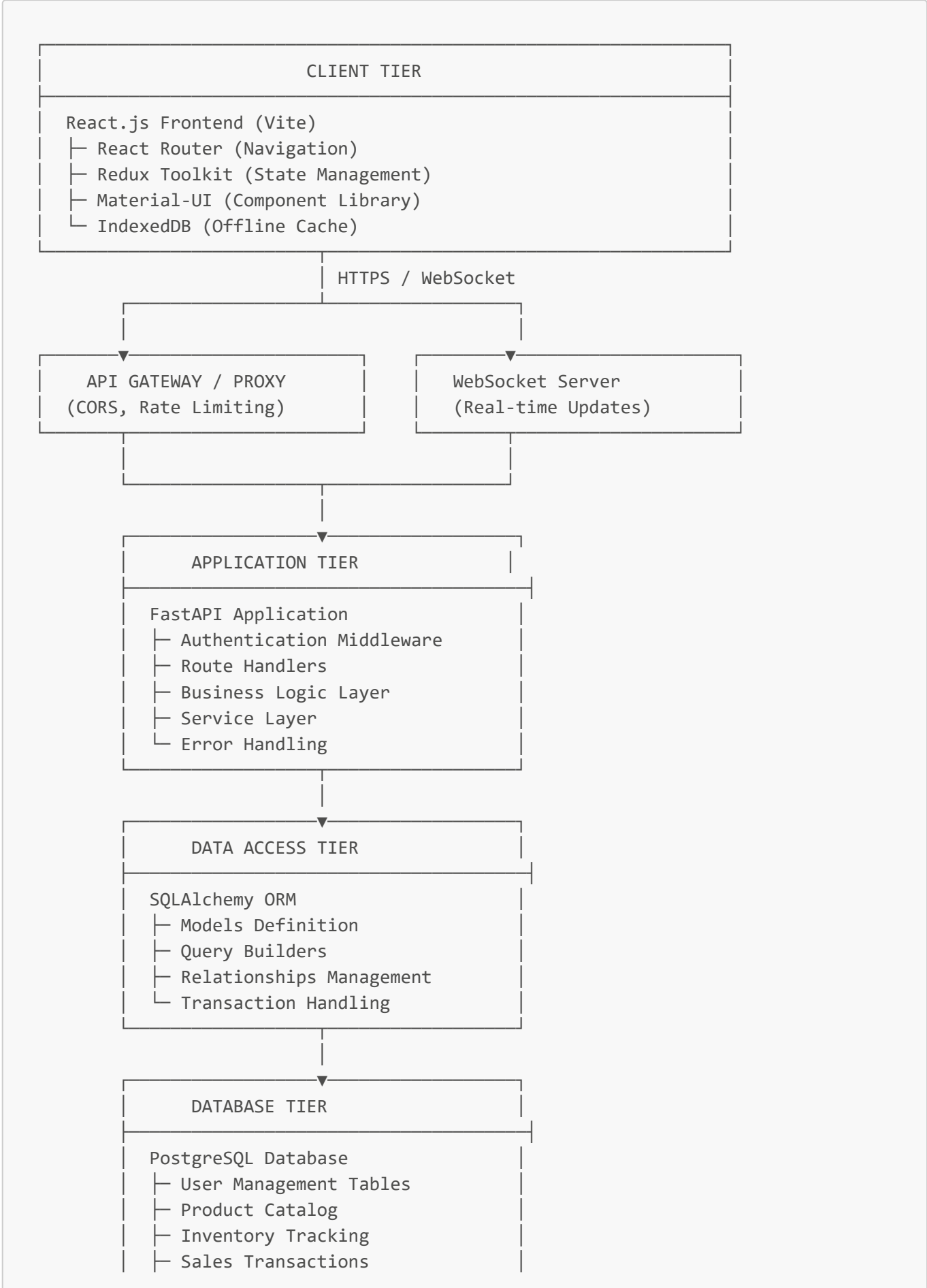
Layer	Technology	Rationale
Frontend	React.js 18+ with Vite	Fast build times, modern SPA capabilities, excellent barcode handling
Frontend UI	Material-UI (MUI) 6.x	Professional retail interface, accessibility compliance, component library
Client Storage	IndexedDB + LocalStorage	Offline capability, quick access to frequently used data
Barcode Scanning	Dynamsoft Barcode Reader / Quagga.js	Accurate detection, camera support, multiple barcode formats
Backend	FastAPI (Python 3.11+)	High performance, async capabilities, excellent for real-time operations
Authentication	JWT (JSON Web Tokens)	Stateless auth, security, scalability
ORM	SQLAlchemy with Alembic	Robust migrations, relationships, query optimization
Database	PostgreSQL 15+	ACID compliance, reliability, support for complex queries
API Documentation	FastAPI Swagger + ReDoc	Auto-generated, interactive documentation
Deployment (Local)	Docker + Docker Compose	Containerization for consistency across systems
Deployment (Cloud)	AWS (EC2 + RDS) / Render / Railway	Scalable, reliable cloud infrastructure
State Management	Redux Toolkit + RTK Query	Predictable state, server sync, caching
Real-time Updates	WebSocket (FastAPI)	Live inventory updates, notification system
Testing	Pytest, Jest, React Testing Library	Comprehensive test coverage

Development Tools

- **Version Control:** Git + GitHub
- **CI/CD:** GitHub Actions
- **Code Quality:** ESLint, Prettier, Black, mypy
- **API Testing:** Postman / Insomnia
- **Monitoring:** Sentry, LogRocket (frontend), structured logging (backend)

3. System Architecture

High-Level Architecture Diagram



```
│   └─ Subscription Management          │
└──────────────────────────────────────┘
```

Data Flow Architecture

```
Barcode Scan → Frontend Capture → IndexedDB Cache → API Call
                                                      │
                                    ┌─────────────────┘
                                    │
                    FastAPI Backend → Business Logic → Database
                                          │
                                          ├─ Price Lookup
                                          ├─ Inventory Update
                                          ├─ Tax Calculation
                                          └─ Sales Recording
                                          │
                                    Response → Frontend
                                          │
                    Display → Update Cache → UI Update
```

---

4. Database Schema

Core Entities and Relationships

```
-- Users & Authentication
users
├─ id (PK)
├─ store_id (FK)
├─ email (UNIQUE)
├─ password_hash
├─ first_name
├─ last_name
├─ role (admin, manager, cashier)
├─ is_active
├─ created_at
└─ updated_at

-- Stores & Subscriptions
stores
├─ id (PK)
├─ store_name
├─ store_code (UNIQUE)
├─ address
├─ city
├─ contact_phone
├─ subscription_tier (basic, professional, enterprise)
├─ subscription_start
├─ subscription_end
```

```
├─ max_products
├─ max_users
├─ is_active
└─ created_at

-- Products & Inventory
products
├─ id (PK)
├─ store_id (FK)
├─ sku (UNIQUE per store)
├─ barcode (UNIQUE)
├─ product_name
├─ description
├─ category_id (FK)
├─ unit_price
├─ cost_price
├─ tax_percentage
├─ is_active
├─ created_at
└─ updated_at

inventory
├─ id (PK)
├─ product_id (FK)
├─ store_id (FK)
├─ quantity_on_hand
├─ quantity_reserved
├─ reorder_point
├─ reorder_quantity
├─ last_counted
├─ updated_at

-- Categories
categories
├─ id (PK)
├─ store_id (FK)
├─ category_name
├─ category_code
├─ description
└─ created_at

-- Transactions & Sales
transactions
├─ id (PK)
├─ store_id (FK)
├─ cashier_id (FK - references users)
├─ transaction_number (UNIQUE)
├─ transaction_date
├─ subtotal
├─ tax_amount
├─ discount_amount
├─ total_amount
├─ payment_method (cash, card, digital)
├─ payment_status (completed, pending, cancelled)
```

```
├─ notes
└─ created_at

-- Transaction Items
transaction_items
├─ id (PK)
├─ transaction_id (FK)
├─ product_id (FK)
├─ quantity_sold
├─ unit_price
├─ tax_percentage
├─ line_total
├─ discount_applied
└─ created_at

-- Subscriptions
subscriptions
├─ id (PK)
├─ store_id (FK)
├─ tier_name
├─ max_daily_transactions
├─ max_products
├─ max_users
├─ monthly_price
├─ annual_price
├─ features (JSON)
└─ created_at

-- Audit & Logs
audit_logs
├─ id (PK)
├─ user_id (FK)
├─ store_id (FK)
├─ action (CREATE, UPDATE, DELETE, LOGIN)
├─ entity_type
├─ entity_id
├─ changes (JSONB - before/after values)
├─ timestamp
```

Database Relationships Diagram

```
stores (1) ──── (M) users
stores (1) ──── (M) products
stores (1) ──── (M) inventory
stores (1) ──── (M) categories
stores (1) ──── (M) transactions
stores (1) ──── (M) subscriptions

products (1) ──── (M) inventory
products (1) ──── (M) transaction_items
categories (1) ──── (M) products
```

```
transactions (1) ——— (M) transaction_items
users (1) ——— (M) transactions (cashier)
users (1) ——— (M) audit_logs

products — barcode index (UNIQUE, for fast scanning)
stores — subscription relationship (for billing)
```

Key Indexes for Performance

```
CREATE INDEX idx_products_barcode ON products(barcode);
CREATE INDEX idx_products_store_sku ON products(store_id, sku);
CREATE INDEX idx_inventory_product ON inventory(product_id);
CREATE INDEX idx_transactions_store_date ON transactions(store_id,
transaction_date);
CREATE INDEX idx_transaction_items_transaction ON
transaction_items(transaction_id);
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_audit_logs_user_timestamp ON audit_logs(user_id, timestamp);
```

---

5. Frontend Architecture

Project Structure

```
frontend/
├── public/
│   └── assets/
├── src/
│   ├── components/
│   │   ├── common/
│   │   │   ├── Header.jsx
│   │   │   ├── Sidebar.jsx
│   │   │   ├── Footer.jsx
│   │   │   └── LoadingSpinner.jsx
│   │   ├── auth/
│   │   │   ├── LoginForm.jsx
│   │   │   ├── SignupForm.jsx
│   │   │   └── ProtectedRoute.jsx
│   │   ├── billing/
│   │   │   ├── BillingCart.jsx
│   │   │   ├── PaymentProcessor.jsx
│   │   │   └── BarcodeScanner.jsx
│   │   ├── products/
│   │   │   ├── ProductList.jsx
│   │   │   ├── ProductForm.jsx
│   │   │   └── ProductSearch.jsx
│   │   ├── inventory/
│   │   │   ├── InventoryTable.jsx
```

```
│   │   │   ├── StockAdjustment.jsx
│   │   │   └── LowStockAlerts.jsx
│   │   ├── reports/
│   │   │   ├── SalesChart.jsx
│   │   │   ├── ReportFilters.jsx
│   │   │   └── ExportOptions.jsx
│   │   └── settings/
│   │       ├── StoreSettings.jsx
│   │       ├── SubscriptionManagement.jsx
│   │       └── UserManagement.jsx
│   ├── pages/
│   │   ├── LandingPage.jsx
│   │   ├── SignupPage.jsx
│   │   ├── LoginPage.jsx
│   │   ├── BillingPage.jsx
│   │   ├── ProductsPage.jsx
│   │   ├── InventoryPage.jsx
│   │   ├── ReportsPage.jsx
│   │   └── SettingsPage.jsx
│   ├── store/
│   │   ├── authSlice.js
│   │   ├── productsSlice.js
│   │   ├── inventorySlice.js
│   │   ├── transactionSlice.js
│   │   ├── settingsSlice.js
│   │   └── store.js
│   ├── services/
│   │   ├── api.js
│   │   ├── authService.js
│   │   ├── productService.js
│   │   ├── transactionService.js
│   │   └── reportService.js
│   ├── hooks/
│   │   ├── useAuth.js
│   │   ├── useFetch.js
│   │   ├── useLocalStorage.js
│   │   └── useBarcodeScanner.js
│   ├── utils/
│   │   ├── formatters.js
│   │   ├── validators.js
│   │   ├── priceCalculator.js
│   │   └── constants.js
│   ├── styles/
│   │   ├── theme.js
│   │   ├── globals.css
│   │   └── variables.css
│   ├── App.jsx
│   └── main.jsx
├── vite.config.js
├── package.json
└── README.md
```

State Management Structure (Redux)

```
store/
├── auth/
│   ├── state: { user, token, isLoading, error, isAuthenticated }
│   ├── actions: login, logout, signup, refreshToken
│   └── selectors: selectUser, selectToken, selectIsAuthenticated
├── products/
│   ├── state: { items, loading, error, filters, searchQuery, pagination }
│   ├── actions: fetchProducts, createProduct, updateProduct, deleteProduct
│   └── selectors: selectAllProducts, selectProductById, selectFilteredProducts
├── transactions/
│   ├── state: { cart, subtotal, taxAmount, total, items, status }
│   ├── actions: addItem, removeItem, updateQuantity, calculateTotals, clearCart
│   └── selectors: selectCartTotal, selectCartItems, selectTax
├── inventory/
│   ├── state: { stock, alerts, lastUpdate }
│   ├── actions: fetchInventory, updateStock, setAlerts
│   └── selectors: selectProductStock, selectLowStockItems
└── settings/
    ├── state: { storeInfo, subscription, users, preferences }
    ├── actions: updateStoreInfo, manageUsers, updatePreferences
    └── selectors: selectStoreInfo, selectSubscriptionTier
```

Key Frontend Libraries

```
{
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.20.0",
    "@mui/material": "^5.14.0",
    "@mui/icons-material": "^5.14.0",
    "@emotion/react": "^11.11.0",
    "@emotion/styled": "^11.11.0",
    "@reduxjs/toolkit": "^1.9.7",
    "react-redux": "^8.1.0",
    "axios": "^1.6.0",
    "@tanstack/react-query": "^5.28.0",
    "react-barcode-scanner": "^4.0.0",
    "quagga": "^0.12.1",
    "chart.js": "^4.4.0",
    "react-chartjs-2": "^5.2.0",
    "date-fns": "^2.30.0",
    "lodash-es": "^4.17.21",
    "yup": "^1.3.0",
    "formik": "^2.4.5",
    "react-hot-toast": "^2.4.1",
    "zustand": "^4.4.0"
  },
```

```
  "devDependencies": {
    "vite": "^5.0.0",
    "@vitejs/plugin-react": "^4.2.0",
    "eslint": "^8.55.0",
    "prettier": "^3.1.0",
    "jest": "^29.7.0",
    "@testing-library/react": "^14.1.0"
  }
}
```

---

6. Backend Architecture

Project Structure

```
backend/
├── app/
│   ├── core/
│   │   ├── config.py          # Environment configuration
│   │   ├── security.py        # JWT, password hashing
│   │   └── constants.py       # App constants
│   ├── db/
│   │   ├── base.py           # SQLAlchemy declarative base
│   │   ├── session.py        # Database session management
│   │   └── models.py         # All SQLAlchemy models
│   ├── api/
│   │   ├── v1/
│   │   │   ├── endpoints/
│   │   │   │   ├── auth.py
│   │   │   │   ├── users.py
│   │   │   │   ├── products.py
│   │   │   │   ├── inventory.py
│   │   │   │   ├── transactions.py
│   │   │   │   ├── reports.py
│   │   │   │   └── settings.py
│   │   │   ├── dependencies.py
│   │   │   └── router.py
│   │   └── v2/  # Future API version
│   ├── middleware/
│   │   ├── error_handler.py
│   │   ├── logging.py
│   │   ├── cors.py
│   │   └── rate_limit.py
│   ├── services/
│   │   ├── auth_service.py
│   │   ├── product_service.py
│   │   ├── transaction_service.py
│   │   ├── inventory_service.py
│   │   ├── report_service.py
│   │   └── email_service.py
│   ├── schemas/
```

```
│   │   ├── user.py
│   │   ├── product.py
│   │   ├── transaction.py
│   │   ├── inventory.py
│   │   └── store.py
│   ├── utils/
│   │   ├── logger.py
│   │   ├── validators.py
│   │   ├── formatters.py
│   │   └── exceptions.py
│   └── main.py          # FastAPI app initialization
├── migrations/          # Alembic migrations
│   ├── env.py
│   ├── script.py.mako
│   └── versions/
├── tests/
│   ├── conftest.py
│   ├── test_auth.py
│   ├── test_products.py
│   ├── test_transactions.py
│   └── test_inventory.py
├── requirements.txt
├── docker-compose.yml
├── Dockerfile
└── README.md
```

FastAPI Main Application Structure

```python
# app/main.py
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.api.v1.router import api_router
from app.core.config import settings
from app.middleware.error_handler import setup_error_handlers
from app.middleware.logging import setup_logging

def create_app() -> FastAPI:
    app = FastAPI(
        title="GroceryPOS Pro API",
        description="Comprehensive POS System for Grocery Retailers",
        version="1.0.0"
    )

    # CORS Configuration
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.CORS_ORIGINS,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )
```

```
    # Error Handling
    setup_error_handlers(app)

    # Logging
    setup_logging(app)

    # Routes
    app.include_router(api_router, prefix="/api/v1")

    return app

app = create_app()
```

Key Backend Dependencies

```
fastapi==0.104.1
uvicorn==0.24.0
sqlalchemy==2.0.23
alembic==1.13.1
psycopg2-binary==2.9.9
python-jose==3.3.0
passlib==1.7.4
python-multipart==0.0.6
pydantic==2.5.0
pydantic-settings==2.1.0
pydantic[email]==2.5.0
pytest==7.4.3
pytest-asyncio==0.21.1
httpx==0.25.2
python-dotenv==1.0.0
aioredis==2.0.1  # For caching
websockets==12.0  # For real-time updates
```

Core Service Layer Example

```
# app/services/transaction_service.py
class TransactionService:
    def __init__(self, db: Session):
        self.db = db

    async def create_transaction(self, store_id: int, cashier_id: int,
                                 items: List[dict], payment_method: str):
        # Calculate totals
        subtotal = sum(item['quantity'] * item['price'] for item in items)
        tax = subtotal * 0.18  # Tax percentage configurable
        total = subtotal + tax

        # Create transaction record
```

```
        transaction = Transaction(
            store_id=store_id,
            cashier_id=cashier_id,
            subtotal=subtotal,
            tax_amount=tax,
            total_amount=total,
            payment_method=payment_method,
            payment_status="completed"
        )

        # Add transaction items & update inventory
        for item in items:
            transaction_item = TransactionItem(
                product_id=item['product_id'],
                quantity_sold=item['quantity'],
                unit_price=item['price']
            )
            transaction.items.append(transaction_item)

            # Update inventory
            await self.update_inventory(item['product_id'],
                                        -item['quantity'], store_id)

        self.db.add(transaction)
        self.db.commit()
        return transaction

    async def update_inventory(self, product_id: int, qty_change: int, store_id:
int):
        inventory = self.db.query(Inventory).filter(
            Inventory.product_id == product_id,
            Inventory.store_id == store_id
        ).first()

        if inventory:
            inventory.quantity_on_hand += qty_change
            self.db.commit()
```

---

7. Page Specifications

Page 1: Landing Page

Purpose: Public-facing introduction to GroceryPOS Pro
Access: Public (no authentication required)
Key Elements:

- Hero section with value proposition
- Feature highlights (real-time billing, inventory management, analytics)
- Pricing plans comparison
- Customer testimonials

- Call-to-action buttons (Sign Up, View Demo)
- FAQ section

Components:

- Navigation header with Login/Sign Up buttons
- Hero section with background image
- Features grid (6-8 key features)
- Pricing tables (Basic, Professional, Enterprise tiers)
- Testimonials carousel
- Contact/Demo request form
- Footer with links

Technical Requirements:

- Responsive design (mobile-first)
- SEO optimization
- Performance optimized
- No authentication needed
- Public API for pricing and feature data

Page 2: Signup Page

Purpose: User registration for new store owners/managers
Access: Public
Key Elements:

- Store information form
- Owner/Manager information
- Email verification
- Password setup with strength indicator
- Terms and conditions agreement
- Plan selection

Form Fields:

```
Store Details:
├─ Store Name (required)
├─ Store Code/ID (auto-generated or custom)
├─ Contact Phone (required)
├─ Address (required)
├─ City/State (required)
├─ Registration Number (optional)

Owner Details:
├─ First Name (required)
├─ Last Name (required)
├─ Email (required, unique)
├─ Password (required, min 8 chars, 1 uppercase, 1 number, 1 special char)
```

```
├─ Confirm Password (required)

Subscription:
├─ Plan Selection (radio buttons)
└─ Payment Information

Agreements:
├─ I agree to Terms of Service (checkbox)
└─ I agree to Privacy Policy (checkbox)
```

Validation Rules:

- Email format validation
- Password strength requirements
- Phone number format
- Store name uniqueness per region
- Email verification before activation

Backend Workflow:

1. Validate input data
2. Check email uniqueness
3. Hash password
4. Create user record
5. Create store record
6. Send verification email
7. Return JWT token for immediate login
8. Initialize subscription record

API Endpoint:

```
POST /api/v1/auth/signup
Body: {
  store_name, store_code, contact_phone, address, city,
  first_name, last_name, email, password, plan_id
}
Response: {
  user_id, store_id, token, refresh_token, message
}
```

Page 3: Login Page

Purpose: Authenticate existing users
Access: Public
Key Elements:

- Email input
- Password input

- Remember me checkbox
- Forgot password link
- Sign up link
- Social login options (future phase)

Validation:

- Email format
- Password not empty
- Account active check
- Subscription active check

Backend Workflow:

1. Validate credentials
2. Check user exists
3. Verify password hash
4. Check subscription status
5. Generate JWT token
6. Generate refresh token
7. Update last_login timestamp
8. Return tokens

API Endpoint:

```
POST /api/v1/auth/login
Body: { email, password, remember_me }
Response: {
  token, refresh_token, user, store,
  subscription_status, expires_in
}
```

Error Handling:

- Invalid credentials (generic message for security)
- Account not verified
- Subscription expired
- Account locked (after 5 failed attempts)

Page 4: Billing Page (POS App)

Purpose: Main checkout interface for retail transactions
Access: Authenticated (Cashier, Manager, Admin)
Key Elements:

Layout Sections:

1. **Header Bar**

- Current time and date
- User greeting (Hello, [Cashier Name])
- Quick settings menu
- Logout button

2. **Left Sidebar - Product Search/Scanner**
   - Large barcode input field (auto-focused)
   - Manual product search dropdown
   - Product quick search by name/category
   - Scan history (last 10 scanned items)

3. **Main Content - Shopping Cart**
   - Item list with columns: Product Name, Qty, Unit Price, Tax, Line Total
   - Edit quantity controls (±, delete)
   - Notes for each item
   - Quick discount application per item
   - Subtotal display

4. **Right Sidebar - Summary & Payment**
   - Subtotal amount
   - Tax breakdown (itemized by tax rate)
   - Discount section (store coupon, cashier discount)
   - **Grand Total (prominent display)**
   - Payment method selection buttons
   - Complete Transaction button
   - Cancel/Clear Cart button

Barcode Scanning Flow:

```
User Focus on Scanner Input
    ↓
Scan barcode or manual input
    ↓
Lookup product in IndexedDB (local cache)
    ↓
If found locally: Add to cart instantly
    ↓
If not found: Call API /products/barcode/{code}
    ↓
Add to cart and update IndexedDB
    ↓
Auto-select quantity input for quick adjustment
    ↓
Continue scanning or press Enter to confirm qty
```

Features:

- **Offline Mode:** Cache top 500 products locally
- **Quick Quantity Entry:** Click qty field and type number
- **Return Items:** Special workflow for returns (negative qty, reason)
- **Discounts:**
  - Percentage discount (% button)
  - Fixed amount discount ($ button)
  - Coupon code entry (validation from backend)
- **Payment Methods:**
  - Cash (exact amount, change calculation)
  - Card (simulated, ready for payment gateway integration)
  - Digital Wallet (QR code generation)
  - Cheque (future)
- **Receipt Generation:**
  - Print preview
  - Email receipt option
  - SMS receipt option (opt-in)

Data Flow:

```
# Frontend state on Billing Page
cart = {
  items: [
    { product_id, product_name, quantity, unit_price, tax%, line_total },
    ...
  ],
  subtotal,
  tax_amount,
  discount_amount,
  total_amount,
  payment_method,
  notes: ""
}

# On transaction completion:
POST /api/v1/transactions/create
{
  store_id, cashier_id, items, subtotal,
  tax_amount, discount_amount, total_amount,
  payment_method, payment_status
}
```

API Endpoints:

```
GET /api/v1/products/search?query=string
GET /api/v1/products/barcode/{barcode}
POST /api/v1/transactions/create
GET /api/v1/transactions/{transaction_id}/receipt
POST /api/v1/transactions/{transaction_id}/email-receipt
```

```
GET /api/v1/products/categories
GET /api/v1/coupons/validate?code=string
```

Page 5: Products Page (POS App)

Purpose: Manage product catalog
Access: Authenticated (Manager, Admin)
Key Elements:

Features:

1. **Product List View**
    - Table with columns: SKU, Barcode, Product Name, Category, Unit Price, Cost Price, Tax%, Active Status
    - Sorting by any column
    - Pagination (20, 50, 100 items per page)
    - Search/filter bar
    - Quick edit buttons (Edit, Delete, Bulk Upload)
2. **Product Form (Add/Edit)**
    - SKU (unique per store)
    - Barcode (auto-generate or manual)
    - Product Name
    - Category selection (dropdown)
    - Description
    - Unit Price
    - Cost Price
    - Tax Percentage (predefined options: 0%, 5%, 12%, 18%)
    - Active/Inactive toggle
    - Image upload (future)
    - Bulk operations
3. **Bulk Upload**
    - CSV template download
    - CSV file upload
    - Validation results
    - Confirmation before saving
4. **Filters & Search:**
    - By category
    - By price range
    - By active status
    - Search by SKU or product name

Data Structure:

```
product_schema = {
  "id": int,
  "store_id": int,
  "sku": str,          # Stock Keeping Unit (unique per store)
  "barcode": str,      # Can be EAN-13, UPC, Code128, etc.
  "product_name": str,
  "description": str,
  "category_id": int,
  "unit_price": float,
  "cost_price": float,
  "tax_percentage": float,
  "is_active": bool,
  "created_at": datetime,
  "updated_at": datetime
}
```

API Endpoints:

```
GET /api/v1/products                    # List all
POST /api/v1/products                   # Create
GET /api/v1/products/{id}               # Get single
PUT /api/v1/products/{id}               # Update
DELETE /api/v1/products/{id}            # Delete
GET /api/v1/products/barcode/{code}     # Search by barcode
POST /api/v1/products/bulk-upload       # Bulk import (CSV)
GET /api/v1/categories                  # Get all categories
POST /api/v1/categories                 # Create category
```

Page 6: Inventory Page (POS App)

Purpose: Monitor and manage stock levels
Access: Authenticated (Manager, Admin)
Key Elements:

Features:

1. **Inventory Dashboard**
    - Total SKUs metric
    - Low stock items (count)
    - Out of stock items (count)
    - Inventory value (quantity × cost price)
    - Last inventory count date
2. **Inventory Table**

- Columns: Product Name, SKU, Current Stock, Reorder Point, Reorder Qty, Last Update
- Color coding: Green (adequate), Yellow (low), Red (critical)
- Edit inline for quick adjustments
- Stock movement history per product

3. **Low Stock Alerts**

- List of items below reorder point
- Suggested purchase orders
- Email notification setup
- Alert acknowledgment

4. **Stock Adjustments**

- Manual adjustment form (Quantity, Reason)
- Reasons: Damage, Theft, Expiry, Breakage, Counting Error, Received, Returned
- Bulk adjustment upload
- Adjustment history with audit trail

5. **Reorder Management**

- Configure reorder point and quantity per product
- Auto-generate purchase orders for low stock items
- Export to PDF for supplier

Data Structure:

```
inventory_schema = {
  "id": int,
  "product_id": int,
  "store_id": int,
  "quantity_on_hand": int,
  "quantity_reserved": int,      # For orders not yet fulfilled
  "reorder_point": int,          # Threshold for alerts
  "reorder_quantity": int,       # Standard order size
  "last_counted": datetime,
  "updated_at": datetime
}
```

API Endpoints:

```
GET /api/v1/inventory                              # List all
GET /api/v1/inventory/low-stock                    # Get low stock items
GET /api/v1/inventory/{product_id}                 # Get single
PUT /api/v1/inventory/{product_id}/adjust          # Manual adjustment
POST /api/v1/inventory/adjustments                 # Bulk adjustments
POST /api/v1/inventory/generate-purchase-order     # Auto PO generation
GET /api/v1/inventory/history/{product_id}         # Stock movement history
```

Page 7: Sales & Reports Page (POS App)

Purpose: Analyze sales data and generate reports
Access: Authenticated (Manager, Admin)
Key Elements:

Dashboard Section:

1. **Key Metrics Cards**
    - Today's Sales
    - Total Transactions (today)
    - Average Transaction Value
    - Top Product
2. **Charts**
    - Sales trend line chart (last 30 days)
    - Category-wise sales pie chart
    - Hourly sales bar chart
    - Payment method breakdown

Reports Section:

1. **Sales Report**
    - Date range filter
    - Filter by: Category, Product, Payment Method
    - Columns: Date, Transaction#, Items, Amount, Payment Method, Cashier
    - Export to PDF/Excel
2. **Product Performance Report**
    - Product name, units sold, revenue, profit, profit%
    - Sort by revenue or units sold
    - Filter by date range and category
3. **Cashier Performance Report**
    - Cashier name, transactions, total sales, average transaction, % of store sales
    - Useful for performance bonuses
4. **Inventory Turnover**
    - Product name, total sold (period), turnover rate
    - Identifies slow-moving items
5. **Tax Report**
    - Tax collected by rate
    - Useful for tax compliance

6. **Custom Report Builder** (Future)
    - Drag-and-drop metrics selection
    - Save custom report templates

API Endpoints:

```
GET /api/v1/reports/dashboard          # Dashboard metrics
GET /api/v1/reports/sales              # Sales report with filters
GET /api/v1/reports/products           # Product performance
GET /api/v1/reports/cashiers           # Cashier metrics
GET /api/v1/reports/inventory-turnover # Inventory analysis
GET /api/v1/reports/tax                # Tax summary
POST /api/v1/reports/export            # Export report
```

Page 8: Settings Page (Store + Subscription)

Purpose: Configure store settings and subscription management
Access: Authenticated (Admin only)
Key Elements:

Tabs:

1. **Store Settings**
    - Store name
    - Store address
    - Contact phone
    - Tax ID/GST number
    - Currency (INR default)
    - Time zone
    - Business hours
    - Tax rates (configuration for different categories)
    - Default discount policies
2. **Subscription Management**
    - Current plan (display only: Basic/Professional/Enterprise)
    - Plan expiry date
    - Upgrade/Downgrade options
    - Billing history (invoices)
    - Auto-renewal status toggle
    - Payment method on file
3. **User Management**
    - List of store users
    - Columns: Name, Email, Role, Status, Created Date

- Add user button (form modal)
- Edit role/status
- Deactivate/Delete user
- Reset password (send email)
- Activity log per user

4. **Backup & Data**

- Last backup date
- Manual backup trigger
- Backup frequency configuration
- Download historical data (CSV/JSON)

5. **System Preferences**

- Number format (1,000 vs 1.000)
- Date format
- Receipt format/customization
- Email notification preferences
- Offline sync frequency
- Session timeout

6. **API Keys** (For integration partners)

- Generate API key
- View active keys
- Revoke key
- Rate limit configuration

7. **Security**

- Change password
- Two-factor authentication setup
- Login history
- Active sessions management

API Endpoints:

```
GET /api/v1/settings/store
PUT /api/v1/settings/store
GET /api/v1/settings/subscription
PUT /api/v1/settings/subscription/upgrade
POST /api/v1/settings/subscription/cancel
GET /api/v1/settings/users
POST /api/v1/settings/users
PUT /api/v1/settings/users/{user_id}
DELETE /api/v1/settings/users/{user_id}
POST /api/v1/settings/backup
GET /api/v1/settings/audit-logs
```

8. Security Implementation

Authentication Strategy

JWT (JSON Web Tokens)

```
Header: {
  "alg": "HS256",
  "typ": "JWT"
}

Payload: {
  "sub": user_id,
  "store_id": store_id,
  "email": user_email,
  "role": user_role,
  "iat": issued_at,
  "exp": expiration_time,
  "aud": "grocerypos-app"
}

Signature: HMAC_SHA256(base64_header + "." + base64_payload, SECRET_KEY)
```

Token Strategy:

- Access token: 15 minutes validity
- Refresh token: 7 days validity
- Store refresh token in HttpOnly cookie (secure against XSS)
- Refresh token rotation on each use

Authorization Strategy

Role-Based Access Control (RBAC)

```
roles:
├─ Admin
│  └─ All permissions (unrestricted)
├─ Manager
│  ├─ View all reports
│  ├─ Manage inventory
│  ├─ Manage products
│  ├─ Manage users (except admin)
│  ├─ Process transactions
│  └─ View settings (read-only)
└─ Cashier
   ├─ Process transactions
   ├─ View own transaction history
   ├─ Search products
   └─ View transaction receipts
```

Permission Decorator (FastAPI)

```
@app.post("/api/v1/products")
@require_auth
@require_role(["admin", "manager"])
async def create_product(product: ProductSchema):
    # Only admin and manager can access
    pass
```

Password Security

Password Requirements:

- Minimum 8 characters
- At least 1 uppercase letter
- At least 1 lowercase letter
- At least 1 number
- At least 1 special character

Hashing Algorithm:

- Use bcrypt with salt rounds = 12
- Never store plaintext passwords
- Never transmit passwords over non-HTTPS

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)
```

Data Protection

Encryption:

- TLS/SSL for all data in transit
- Data at rest: Sensitive fields encrypted (payment info, SSN, etc.)
- Database-level encryption recommended for production

Sensitive Data Fields:

- User passwords (hashed)

- Payment information (PCI DSS compliant - tokenized)
- User phone numbers (optional encryption)
- Store tax ID (encrypted)

API Security

Rate Limiting:

- 100 requests per minute per IP for public endpoints
- 1000 requests per minute per authenticated user
- Stricter limits on auth endpoints (5 requests/minute for login)

CORS Configuration:

```
allow_origins = [
    "http://localhost:3000",        # Dev
    "http://localhost:5173",        # Vite dev
    "https://app.grocerypos.com",   # Production
    "https://admin.grocerypos.com"  # Admin dashboard
]
```

Input Validation:

- Validate all user inputs on backend
- Reject oversized payloads (max 10MB for uploads)
- Sanitize strings to prevent SQL injection
- Validate email, phone, URLs before processing

CSRF Protection:

- Use CSRF tokens for state-changing operations
- Token validated on backend for all POST/PUT/DELETE requests

Logging & Monitoring

Security Audit Trail:

```
# Log all security-relevant events
audit_log_entry = {
  "timestamp": datetime.now(),
  "user_id": user.id,
  "action": "LOGIN_SUCCESS | LOGIN_FAILURE | UNAUTHORIZED_ACCESS",
  "resource": "/api/v1/products",
  "method": "GET",
  "status_code": 200,
  "ip_address": request.client.host,
  "user_agent": request.headers.get("user-agent")
}
```

Alerts:

- Failed login attempts (>5 triggers account lock)
- Unauthorized access attempts
- Large data exports
- Permission changes
- Subscription changes

Compliance

Data Privacy:

- GDPR compliance (if EU users)
- Data retention policies (delete old data after 2 years)
- User data export functionality
- Right to be forgotten implementation

PCI DSS (if handling cards):

- Never store full card numbers
- Use tokenization service (Stripe, Square, etc.)
- Encrypt card data in transit
- Regular security assessments

---

9. Barcode Scanning Integration

Implementation Strategy

Frontend Approach:

1. **Dual Input Method:**
   - Camera-based scanning (using `quagga.js` or `react-barcode-scanner`)
   - Keyboard input (for USB barcode scanners)
2. **Quagga.js Configuration:**

```
// For high-speed barcode scanning (USB scanner emulation)
useEffect(() => {
  const handleBarcodeInput = (event) => {
    if (event.code.startsWith('Digit') || event.key.match(/[0-9]/)) {
      barcodeBuffer += event.key;
    }

    // When barcode complete (usually ends with Enter key)
    if (event.key === 'Enter') {
      processBarcode(barcodeBuffer);
      barcodeBuffer = '';
    }
  };
```

```
  window.addEventListener('keydown', handleBarcodeInput);
  return () => window.removeEventListener('keydown', handleBarcodeInput);
}, []);
```

3. **Mobile Camera Scanning (Future):**

```
// Using react-barcode-scanner for mobile
import { useCamera } from '@react-barcode-scanner/core';

const BarcodeScanner = () => {
  const { enableCamera, scan } = useCamera();

  useEffect(() => {
    enableCamera();
    const unsub = scan((data) => {
      processBarcode(data);
    });
    return () => unsub();
  }, []);
};
```

Barcode Processing Flow

```
1. Barcode Received
   ├─ Check length and format (EAN-13, UPC, etc.)
   ├─ If valid: Proceed to lookup
   └─ If invalid: Show error, clear input

2. Product Lookup
   ├─ Query IndexedDB cache first (fastest - <5ms)
   ├─ If found: Add to cart
   ├─ If not found: Call API /products/barcode/{code}
   └─ On API response: Update cache and add to cart

3. Add to Cart
   ├─ Check if product already in cart
   ├─ If yes: Increment quantity
   ├─ If no: Add new line item
   ├─ Update totals
   ├─ Auto-focus on quantity for quick edit
   └─ Clear barcode input for next scan
```

Backend Barcode Service

```
# app/services/barcode_service.py
class BarcodeService:
```

```python
    @staticmethod
    def validate_barcode(barcode: str) -> bool:
        """Validate barcode format"""
        # EAN-13
        if len(barcode) == 13:
            return BarcodeService.validate_ean13(barcode)
        # UPC-A
        elif len(barcode) == 12:
            return BarcodeService.validate_upc_a(barcode)
        # Code128
        elif len(barcode) <= 20:
            return True
        return False

    @staticmethod
    def lookup_product(barcode: str, store_id: int):
        """Find product by barcode"""
        product = db.query(Product).filter(
            Product.barcode == barcode,
            Product.store_id == store_id
        ).first()

        if not product:
            raise ProductNotFound(f"No product with barcode {barcode}")

        return product

    @staticmethod
    def generate_barcode(product_id: int) -> str:
        """Generate barcode for new product"""
        import barcode
        from barcode.writer import ImageWriter

        product = db.query(Product).get(product_id)
        ean = barcode.get_barcode_class('ean13')

        # Generate unique EAN-13
        barcode_num = f"978{product.id:010d}"
        return barcode_num
```

IndexedDB Cache Strategy

```javascript
// Cache for frequently accessed products
const db_open = indexedDB.open('GroceryPOS', 1);

db_open.onupgradeneeded = (event) => {
  const db = event.target.result;

  // Products store with barcode index
  const productStore = db.createObjectStore('products', { keyPath: 'id' });
  productStore.createIndex('barcode', 'barcode', { unique: true });
```

```
  productStore.createIndex('sku', 'sku');
  productStore.createIndex('name', 'product_name');
};

// Add product to cache
async function cacheProduct(product) {
  const db = await openDB('GroceryPOS');
  const tx = db.transaction('products', 'readwrite');
  tx.objectStore('products').put(product);
}

// Lookup in cache
async function lookupProductInCache(barcode) {
  const db = await openDB('GroceryPOS');
  const tx = db.transaction('products', 'readonly');
  const index = tx.objectStore('products').index('barcode');
  return await index.get(barcode);
}

// Sync cache with server (hourly)
setInterval(async () => {
  const products = await fetchProducts();
  for (const product of products) {
    await cacheProduct(product);
  }
}, 3600000); // Every hour
```

Multiple Barcode Format Support

```
# Supported formats
BARCODE_FORMATS = {
    'EAN-13': {
        'length': 13,
        'validator': validate_ean13_checksum,
        'example': '5901234123457'
    },
    'EAN-8': {
        'length': 8,
        'validator': validate_ean8_checksum,
        'example': '96385074'
    },
    'UPC-A': {
        'length': 12,
        'validator': validate_upc_checksum,
        'example': '123456789012'
    },
    'Code128': {
        'length': 'variable',
        'validator': None,  # No strict validation
        'example': 'ABC123'
```

```
    }
}
```

---

10. Deployment Strategy

Local PC Deployment

For single-store deployment:

```yaml
# docker-compose.yml
version: '3.8'

services:
  postgres:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: grocerypos
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secure_password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    environment:
      DATABASE_URL: postgresql://admin:secure_password@postgres:5432/grocerypos
      JWT_SECRET: your_secret_key_here
      JWT_ALGORITHM: HS256
      CORS_ORIGINS: http://localhost:3000
    ports:
      - "8000:8000"
    depends_on:
      - postgres
    volumes:
      - ./backend:/app

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      REACT_APP_API_URL: http://localhost:8000/api/v1
```

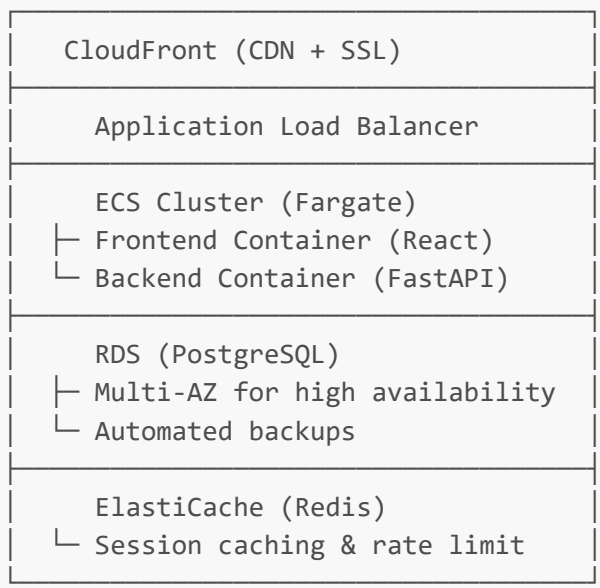
```
volumes:
  postgres_data:
```

Installation Steps:

1. Install Docker and Docker Compose
2. Clone repository
3. `docker-compose up -d`
4. Backend runs on http://localhost:8000
5. Frontend runs on http://localhost:3000
6. Database accessible on localhost:5432

Online Cloud Deployment

Recommended: AWS Architecture



Alternative Simple Deployment (Render or Railway):

1. Push code to GitHub
2. Connect to Render/Railway
3. Auto-deploy on push
4. Free tier available for development

---

11. Scalability Roadmap

Phase 1: Single Store (Current)

- One database instance
- Monolithic backend
- Single frontend deployment

- Target: 50-100 transactions/day

Phase 2: Multi-Store (Q2 2026)

- Database design ready for multi-tenancy
- Store-level filtering in all queries
- Separate admin dashboard for store management
- Target: 10-20 stores, 500-1000 transactions/day

Phase 3: Advanced Features (Q3 2026)

- Customer loyalty program
- Integration with payment gateways
- Mobile app (React Native)
- Advanced analytics with ML insights
- Target: 50+ stores

Phase 4: Enterprise (Q4 2026)

- API for third-party integrations
- Custom report builder
- Warehouse management system integration
- Multi-currency support
- Target: 100+ stores

Performance Optimization Strategy

- Database query optimization (indexes, query analysis)
- Caching layer (Redis)
- CDN for static assets
- API rate limiting
- Async processing for reports
- Background jobs for batch operations

---

12. Project Timeline

Sprint 0: Setup (Week 1)

- ☐ Project repository setup
- ☐ Development environment configuration
- ☐ Database schema creation
- ☐ API documentation setup
- ☐ Testing framework setup

Sprint 1: Authentication & Core Setup (Weeks 2-3)

- ☐ Landing Page design & implementation
- ☐ User registration system
- ☐ Login system with JWT auth

- [ ] Password reset flow
- [ ] Database migrations
- [ ] Basic API endpoints

Sprint 2: Product Management (Weeks 4-5)

- [ ] Products page (list, create, edit, delete)
- [ ] Barcode integration
- [ ] Category management
- [ ] Bulk product upload (CSV)
- [ ] Product search API
- [ ] IndexedDB caching

Sprint 3: Billing System (Weeks 6-7)

- [ ] Billing page UI
- [ ] Barcode scanner implementation
- [ ] Shopping cart state management
- [ ] Transaction creation API
- [ ] Payment method integration
- [ ] Receipt generation
- [ ] Offline capability

Sprint 4: Inventory & Reports (Weeks 8-9)

- [ ] Inventory page
- [ ] Stock adjustment workflows
- [ ] Low stock alerts
- [ ] Reports page
- [ ] Dashboard with charts
- [ ] Report export (PDF/Excel)

Sprint 5: Settings & Security (Weeks 10-11)

- [ ] Store settings page
- [ ] User management
- [ ] Subscription management
- [ ] Audit logging
- [ ] Security hardening
- [ ] Performance testing

Sprint 6: Testing & Deployment (Weeks 12-13)

- [ ] Unit testing
- [ ] Integration testing
- [ ] End-to-end testing
- [ ] Load testing
- [ ] Docker setup
- [ ] Production deployment

- [ ] User documentation
- [ ] Training materials

Appendix A: API Response Standards

Success Response

```
{
  "success": true,
  "data": { /* Actual data */ },
  "message": "Operation completed successfully"
}
```

Error Response

```
{
  "success": false,
  "error": {
    "code": "PRODUCT_NOT_FOUND",
    "message": "Product with ID 123 not found",
    "status": 404
  }
}
```

Appendix B: Key Metrics for Success

1. **System Performance**
    - Page load time: < 2 seconds
    - API response time: < 500ms (p95)
    - Barcode scan to cart: < 1 second
2. **User Metrics**
    - Transactions per minute: 10+
    - Average checkout time: < 3 minutes
    - User onboarding time: < 30 minutes
3. **Business Metrics**
    - Monthly active stores: Grows 20% MoM
    - Customer retention: >80% annually
    - System uptime: >99.5%

Appendix C: Risks and Mitigation

Risk	Impact	Mitigation
Internet outage	Loss of sales	Offline mode with sync queue
Database corruption	Data loss	Daily backups, replication
Barcode format issues	Scanning failures	Support multiple formats, fallback to manual entry
Payment gateway failure	Cannot accept payments	Fallback to offline payment recording
High concurrency load	System slowdown	Caching, database optimization, load balancing

Document Version: 1.0
Last Updated: January 7, 2026
Next Review: January 14, 2026