RetailPulse Docs

None

None

None

Table of contents

1. RetailPulse

A scalable, web-based system for multi-store billing and inventory for wholesale businesses.

1.1 The Problem

As a wholesale business grows from a single store to multiple locations, it faces significant operational challenges:

1.1.1 1. Lack of Centralized Control

Owners cannot get a clear, real-time view of performance across all stores. Each location operates in a silo, making it impossible to compare sales, manage stock, or enforce product handling standards effectively.

1.1.2 2. Complex Inventory Management

Tracking product inventory becomes a nightmare. It's difficult to know the exact stock levels at each location, leading to overstocking in one store and stockouts in another. Tracking critical details like batch numbers, expiry dates, and storage conditions (e.g., cold storage) across a network of stores is difficult with manual methods.

1.1.3 3. Inefficient Operations

Manual processes like handwriting bills, physically checking stock, and consolidating paper reports are slow, prone to errors, and do not scale as the business expands.

1.2 The Solution – RetailPulse

RetailPulse is a centralized, web-based system that empowers wholesale businesses to manage their entire network of stores from a single platform.

1. **Multi-Store Billing**: Fast, digital billing at each location, with all data instantly available at the head office.
2. **Granular Product Inventory**: Track your entire inventory in real-time, right down to the specific batch, expiry date, and location (e.g., Shelf A1, Cold Storage) in each store.

1.3 Who Uses This System?

The system is designed with a clear role-based structure to support a growing wholesale business.

Role	Scope	Primary Task
Company Admin	All Stores	Oversees the entire business, manages stores, and views company-wide reports.
Store Manager	Single Store	Manages day-to-day operations and staff for their assigned store.
Stockist	Single Store	Manages all inventory within their assigned store, from receiving to organizing.
Company Stockist	All Stores	Manages high-level inventory, including transfers between stores.
Sales	Assigned Stores	Creates bills for customers at the point of sale and tracks sales data to identify trends.

2. Functional Documentation

2.1 User Roles and Permissions

To support a growing multi-store wholesale business, we have expanded our user roles. The system now supports a clear hierarchy, ensuring staff can only access the information they need.

Role	Scope	Key Responsibilities
Company Admin	All Stores	Has complete control. Manages stores, manages overall inventory operations, and manages all user accounts.
Store Manager	Assigned Store	Manages a single store. Oversees daily operations, manages staff, and handles local inventory.
Stockist	Assigned Store	Manages the inventory for a single store. Responsible for receiving new product shipments, organizing them, and tracking their location.
Company Stockist	All Stores	A central role for inventory oversight. Can view inventory across all stores.
Sales	Assigned Store(s)	Creates bills for customers at the point of sale and handles customer interactions.

2.2 System Modules Overview

RetailPulse is structured into four main modules, designed to support a multi-store wholesale operation.

2.3 Module 1: Bill Generation

2.3.1 Purpose

To enable fast and accurate billing for customers at the store level, with all data automatically synced, primarily handled by the Sales role.

2.3.2 Features

1.1 Product Search and Selection

- **Store-Specific Search**: Find products available at the user's current store.
- **Detailed View**: See stock availability for different batches (e.g., with different expiry dates) before adding to a bill.
- **Multi-Product**: Add multiple products to a single bill.

1.2 Real-time Stock Validation

- **Batch-Level Check**: When a product is added to a bill, the system checks the quantity available in a specific batch.
- **Quantity Lock**: Ensures the amount requested does not exceed what's available in the selected batch.

1.3 Bill Creation Form

- **Customer Info**: Add customer name (e.g., customer or business name) and contact details.
- **Product List**: Shows product name, batch number, expiry date, unit price, quantity, and total price.

- **Calculations**: Automatically calculates subtotal, discount, tax, and the grand total.

1.4 Bill Numbering and Tracking

- **Smart Numbering**: Bills get a unique number that includes the store ID (e.g., `STORE1-BILL-00001`).
- **Audit Trail**: Records the date, time, and the Sales user who created the bill.

1.5 PDF Generation

- **Customized PDFs**: Bills are generated in a professional PDF format with the specific store's logo and address.
- **Instant Download**: Download the PDF right after creating the bill.

1.6 Automatic Inventory Update

- **Deduct from Batch**: When a bill is finalized, the stock quantity is automatically deducted from the correct batch in the inventory.

2.4 Module 2: Inventory Management

2.4.1 Purpose

To provide powerful, granular control over product inventory across all stores, from receiving shipments to tracking their final sale.

2.4.2 Features

2.1 Product Master List

- **Central Catalog**: A central list of all products the company sells, managed by Company Admins.
- **Basic Info**: Includes product name, category, manufacturer, and description.

2.2 Detailed Inventory Tracking

- **Batch & Expiry**: Each product delivery is stored as a unique batch with its own batch number, manufacturing ID, expiry date, cost price, and selling price.
- **Location Management**: Track exactly where each batch is located within a store (e.g., `Shelf A1`, `Storeroom Rack 3`).
- **Storage Categories**: Assign a storage type to each item, such as `Cold Storage` or `General`, to ensure proper handling of sensitive products.

2.3 View Inventory

- **Company-Wide View**: Company Admins and Company Stockists can see inventory levels across all stores.
- **Store-Level View**: Store Managers and Stockists can see a detailed view of the inventory in their own store.
- **Advanced Filtering**: Filter inventory by product, category, expiry date (e.g., "expiring in 30 days"), or location.

2.4 Stock Adjustments and Movement

- **Manual Adjustments**: Manually change stock quantity with a mandatory reason (e.g., "Damaged stock," "Cycle count correction").
- **Internal Movement**: Log the movement of stock from one location to another within the same store (e.g., from the storeroom to a shelf).

2.5 Module 3: Authentication and Authorization

2.5.1 Purpose

To secure the system and ensure users only access what their role permits.

2.5.2 Features

3.1 User Registration and Management

- **Admin Control**: Only Company Admins can create or deactivate stores and other users.
- **Store Assignment**: When creating a user, an Admin assigns them a role and, if applicable, a home store.

3.2 User Login

- **Secure Login**: Users log in with a username and password.
- **Session Management**: The system manages user sessions and provides automatic logouts for security.

3.3 Role-Based Access Control (RBAC)

- **Granular Permissions**: The system enforces the permissions defined in the **User Roles** table. For example, a `Sales` user from Store A cannot create a bill for Store B, nor can they see Store B's inventory. A `Company Admin` can do both.

3. Technical Docs

3.1 Tech Stack

3.1.1 Frontend

Technology	Version	Purpose
React	v19.2.0	UI library for building component-based interfaces
Vite	v7.2.2	Build tool and development server
React Router	v6.28.0	Client-side routing and navigation
Material UI (MUI)	v6.1.7	React component library for design system
Axios	v1.7.7	HTTP client for API communication
Jest	v30.2.0	JavaScript testing framework

3.1.2 Backend

Technology	Version	Purpose
Python	v3.12.7	Programming language
FastAPI	v0.121.1	Modern async web framework
SQLAlchemy	v2.0.44	SQL toolkit and ORM
PostgreSQL	v16.6	Relational database
ReportLab	v4.4.4	PDF generation library
Pytest	v9.0.0	Testing framework
Uvicorn	v0.38.0	ASGI server for FastAPI

3.1.3 DevOps & Deployment

Technology	Version	Purpose
AWS EC2	-	Application hosting (compute)
AWS RDS	-	Managed PostgreSQL database
GitHub Actions	-	CI/CD pipeline automation

3.2 System Data Flow

This document shows data flow sequence diagrams for each user role in the RetailPulse system.

3.2.1 1. Company Admin Data Flow

The Company Admin manages the entire system, including stores and users.

```
sequenceDiagram
    actor Admin as Company Admin
    participant System
    participant DB as Database

    Admin->>System: Manage Stores
    System->>DB: Create/Update/Delete Store
    DB-->>System: Confirmation
    System-->>Admin: Store Updated

    Admin->>System: Manage Users
    System->>DB: Create/Update/Delete User
    DB-->>System: Confirmation
    System-->>Admin: User Updated
```

Key Functions: - Manage all stores (create, edit, delete) - Manage all users across stores

3.2.2 2. Store Manager Data Flow

The Store Manager oversees a specific store, manages store staff, and views store-level inventory.

```
sequenceDiagram
    actor Manager as Store Manager
    participant System
    participant DB as Database

    Manager->>System: Manage Staff
    System->>DB: Create/Update Staff
    DB-->>System: Confirmation
    System-->>Manager: Staff Updated

    Manager->>System: View Inventory
    System->>DB: Get Store Inventory
    DB-->>System: Inventory Data
    System-->>Manager: Display Stock Levels
```

Key Functions: - Manage store staff (create Sales, Stockist users) - View store inventory levels

3.2.3 3. Sales Data Flow

The Sales user creates bills, searches for products, and handles customer transactions.

```
sequenceDiagram
    actor Sales
    participant System
    participant DB as Database

    Sales->>System: Search Product
    System->>DB: Query Products
    DB-->>System: Product List
    System-->>Sales: Display Products

    Sales->>System: Create Bill
    System->>DB: Save Bill
    System->>DB: Deduct Inventory
    DB-->>System: Confirmation
    System-->>Sales: Bill & PDF Generated
```

Key Functions: - Search for products - Create customer bills and generate PDF - Automatic inventory deduction

3.2.4 4. Stockist Data Flow

The Stockist manages local store inventory, receives stock, and updates quantities.

```
sequenceDiagram
    actor Stockist
    participant System
    participant DB as Database

    Stockist->>System: View Inventory
    System->>DB: Get Store Stock
    DB-->>System: Stock with Batches
    System-->>Stockist: Display Inventory

    Stockist->>System: Receive Stock
    System->>DB: Add Stock & Batch Details
    DB-->>System: Confirmation
    System-->>Stockist: Stock Added Successfully
```

Key Functions: - View store inventory with batch details - Receive and add new stock with expiry dates

---

3.2.5 5. Company Stockist Data Flow

The Company Stockist oversees inventory across all stores and manages inter-store transfers.

```
sequenceDiagram
    actor CompStockist as Company Stockist
    participant System
    participant DB as Database

    CompStockist->>System: View All Inventory
    System->>DB: Get All Stores Inventory
    DB-->>System: Consolidated Data
    System-->>CompStockist: Display All Stock

    CompStockist->>System: Transfer Stock
    System->>DB: Update Source Store
    System->>DB: Update Destination Store
    DB-->>System: Confirmation
    System-->>CompStockist: Transfer Completed
```

3.3 Authentication and Authorization

3.3.1 Authentication Flow

A user enters their username and password on the login page. The system checks if the credentials are correct. If they are, the system identifies their role and redirects them to the appropriate dashboard. If not, an error message is shown.

```
graph TD
    A[User enters credentials on login page] --> B{Are credentials valid?};
    B -- Yes --> C[Log in user];
    B -- No --> G[Show error message];
    C --> D{Check user role};
    D --> E[Redirect to Role-Specific Dashboard];
```

3.3.2 Authorization Rules

The system uses a granular Role-Based Access Control (RBAC) model to enforce permissions. A user's role determines what they can see and do, and their assigned store (if any) limits the scope of their actions.

Below is a summary of the permissions for each role.

Action	Company Admin	Store Manager	Stockist	Company Stockist	Sales
Store Management					
Create/Edit Stores	Yes	No	No	No	No
User Management					
Create/Edit Users	Yes	No	No	No	No
Product Master List					
Create/Edit Products	Yes	No	No	No	No
Inventory					
View Own Store Inventory	Yes	Yes	Yes	Yes	Yes
View All Stores' Inventory	Yes	No	Yes	No	No
Add/Edit Own Store Inventory	No	No	Yes	Yes	No
Billing					
Create/Edit Bills (own store)	No	No	No	No	Yes
View Bills (own store)	Yes	Yes	No	No	Yes
View Bills (all stores)	Yes	No	No	No	No

3.4 Database Schema

3.4.1 Overview

The system uses **PostgreSQL** for its ACID compliance, strong relational support, and efficient handling of complex joins needed for multi-store inventory and RBAC.

Users can have multiple roles and work across multiple stores via junction tables (`user_stores` and `user_roles`). All tables use soft deletes with `created_at`, `updated_at`, and `deleted_at` timestamps.

3.4.2 Why PostgreSQL

- **ACID compliance**: Financial and inventory transactions require guaranteed consistency
- **Complex relationships**: Multiple many-to-many relationships (users-roles, users-stores, bills-products)
- **Structured data**: Product details, batches, pricing, and transactions are inherently structured
- **SQL querying**: Efficient aggregations and reporting on relational data
- **RBAC support**: Natural fit for role-permission modeling

3.4.3 Entity Relationship Diagram

```
erDiagram
    users ||--|{ user_roles : "has"
    roles ||--o{ user_roles : "has many"
    users ||--|{ user_stores : "is in"
    stores ||--o{ user_stores : "has many"

    stores ||--|{ bills : "generates"
    stores ||--|{ inventory_items : "has"
    users ||--o{ bills : "creates"

    categories ||--o{ products : "has"
    products ||--o{ inventory_items : "is an instance of"
    bills ||--|{ bill_products : "contains"
    products ||--o{ bill_products : "appears in"


    users {
        uuid id PK
        varchar username UK
        varchar password_hash
        varchar full_name
        boolean is_active
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }

    roles {
        uuid id PK
        varchar name UK
        text description
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }

    user_roles {
        uuid id PK
        uuid user_id FK
        uuid role_id FK
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }



    stores {
        uuid id PK
        varchar name UK
        text address
        varchar contact_phone
        timestamptz created_at
        timestamptz updated_at
```

```
        timestamptz deleted_at
    }

    user_stores {
        uuid id PK
        uuid user_id FK
        uuid store_id FK
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }

    categories {
        uuid id PK
        varchar name UK
        text description
        boolean is_active
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }

    products {
        uuid id PK
        uuid category_id FK
        varchar name UK
        varchar manufacturer
        text description
        decimal price
        boolean is_active
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }

    inventory_items {
        uuid id PK
        uuid product_id FK
        uuid store_id FK
        varchar batch_number
        date expire_date
        int quantity
        varchar location
        decimal cost_price
        decimal selling_price
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }



    bills {
        uuid id PK
        uuid store_id FK
        uuid created_by_user_id FK
        varchar bill_number UK
        varchar customer_name
        varchar customer_contact
        decimal total_amount
        decimal discount
        decimal tax_amount
        decimal grand_total
        timestamptz bill_date
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }

    bill_products {
        uuid id PK
        uuid bill_id FK
        uuid product_id FK
        varchar batch_number
        int quantity
        decimal unit_price
        decimal total_price
        timestamptz created_at
        timestamptz updated_at
        timestamptz deleted_at
    }
```

3.4.4 Database Tables

`stores`

Store locations with name, address, and contact info.

`users`

Central user records with username, password hash, and full name. Roles and stores assigned via junction tables.

`roles`

Defines available roles (Company Admin, Store Manager, Sales, Stockist, Company Stockist).

`user_roles`

Junction table linking users to roles (many-to-many).

`user_stores`

Junction table linking users to stores (many-to-many).

`categories`

Product categories with name and description for organizing products.

`products`

Master product catalog with category reference, name, manufacturer, and description.

`inventory_items`

Batch-level inventory tracking per store with expiry dates, quantities, location, and pricing.

`bills`

Bill headers with store, user, bill number, customer details, and totals.

`bill_products`

Bill line items with product, quantity, pricing, and batch number for inventory tracking.

3.4.5 Indexing Strategy

Auto-created indexes:

- Primary keys (all tables)
- Unique constraints (`username` , `bill_number` , product/role/permission `name`)

Explicit B-tree indexes on:

Foreign keys (for JOIN performance):

- `user_roles` : `user_id` , `role_id`
- `user_stores` : `user_id` , `store_id`
- `products` : `category_id`
- `inventory_items` : `product_id` , `store_id`
- `bills` : `store_id` , `created_by_user_id`
- `bill_products` : `bill_id` , `product_id`

Query-heavy columns:

- `inventory_items.expire_date` - frequent expiry checks and range queries
- `inventory_items.batch_number` - batch-specific searches
- `categories.name` - filtering by category
- `products.manufacturer` - filtering by manufacturer
- `bills.bill_date` - date-based reporting

Why B-tree: Handles equality, range queries, sorting, and pattern matching efficiently - covers all common query patterns in the system.

3.5 Architecture

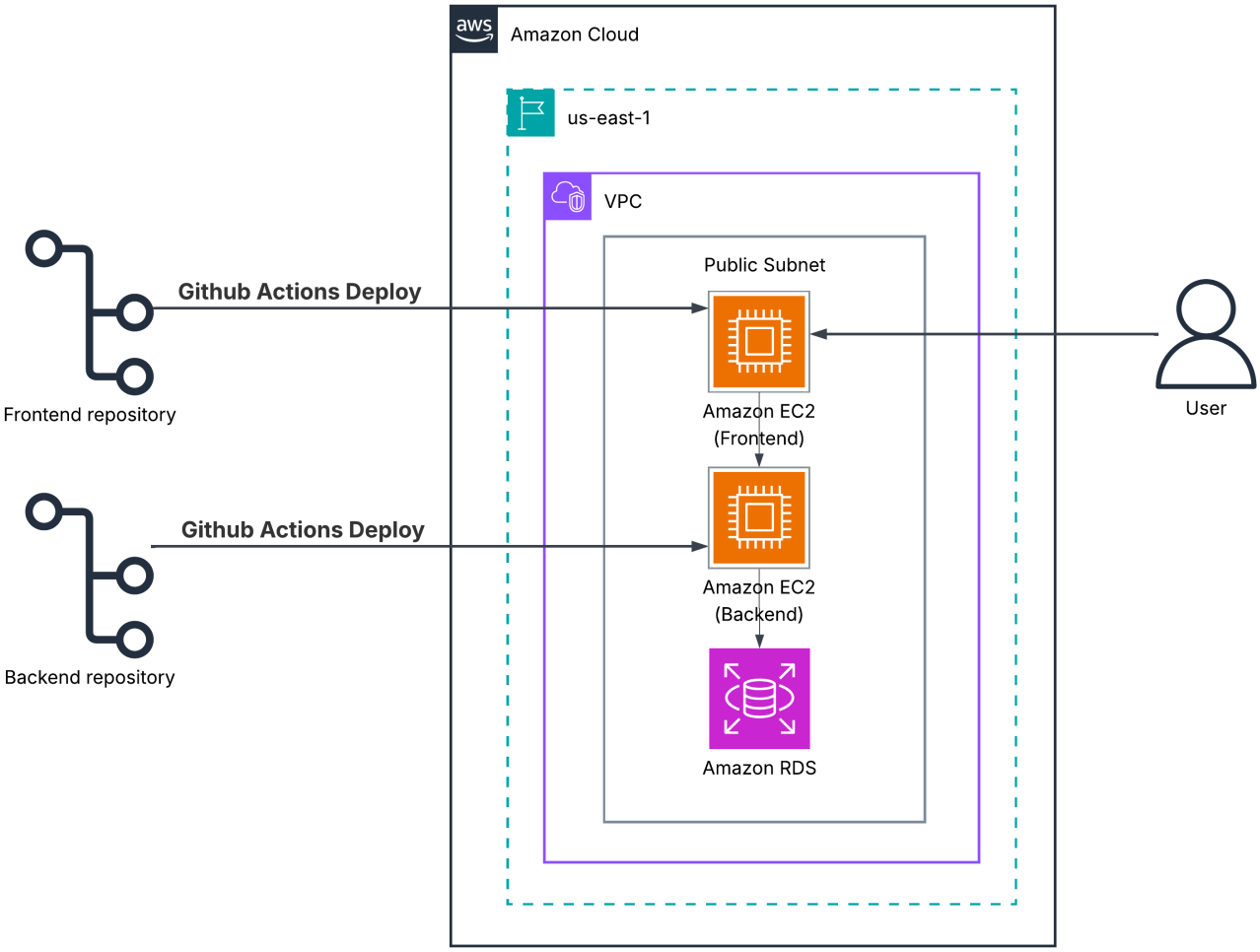
3.5.1 System Architecture Overview

RetailPulse is a standard web application with three layers:

1. **Frontend**: A React application that users see in their web browser.
2. **Backend**: A FastAPI application that contains all the business logic.
3. **Database**: A PostgreSQL database that stores all the data.

3.5.2 AWS-Specific Architecture Diagram

This diagram shows a simplified view of the system hosted on AWS.



3.6 API Design

The API is designed following RESTful principles to provide a logical, hierarchical structure for managing a multi-store wholesale business. Resources are nested where it makes sense (e.g., a bill belongs to a store).

Authentication Endpoints

Method	Endpoint	Description
`POST`	`/auth/login`	User login to get a JWT access token.
`POST`	`/auth/logout`	User logout (invalidates token).

Store Endpoints

Method	Endpoint	Description
`POST`	`/stores`	Create a new store.
`GET`	`/stores`	List all stores in the company.
`GET`	`/stores/{store_id}`	Get details for a specific store.
`PATCH`	`/stores/{store_id}`	Update a store's details.

User Management Endpoints

Method	Endpoint	Description
`POST`	`/users`	Create a new user (can assign a role and store).
`GET`	`/users`	List all users (can filter by store, role).
`GET`	`/users/{user_id}`	Get details for a specific user.
`PATCH`	`/users/{user_id}`	Update a user's details (e.g., role, store).
`DELETE`	`/users/{user_id}`	Deactivate a user account.

Product Master Endpoints

Method	Endpoint	Description
`POST`	`/products`	Create a new product in the master catalog.
`GET`	`/products`	List all products in the master catalog.
`GET`	`/products/{product_id}`	Get details for a specific master product.
`PATCH`	`/products/{product_id}`	Update a master product's details.

Inventory Endpoints

Method	Endpoint	Description
`POST`	`/stores/{store_id}/inventory-items`	Add a new inventory item (a new batch) to a store.
`GET`	`/stores/{store_id}/inventory-items`	Get all inventory items for a specific store.
`GET`	`/inventory-items/{item_id}`	Get details of a specific inventory item (batch).
`PATCH`	`/inventory-items/{item_id}`	Update an inventory item (e.g., adjust quantity, change price).
`POST`	`/inventory-items/movements`	Log an internal movement of stock within a store.
`POST`	`/inventory-transfers`	Initiate a transfer of stock between two stores.
`PATCH`	`/inventory-transfers/{transfer_id}`	Update the status of a transfer (e.g., confirm receipt).

Bill Endpoints

Method	Endpoint	Description
`POST`	`/stores/{store_id}/bills`	Create a new bill for a specific store.
`GET`	`/stores/{store_id}/bills`	List all bills for a specific store.
`GET`	`/bills/{bill_id}`	Get details for a specific bill (accessible across stores by Admins).

3.7 Testing Documentation

3.7.1 Testing Frameworks

- **Backend**: Pytest v9.0.0 (pytest-asyncio v1.3.0, pytest-cov v7.0.0)
- **Frontend**: Jest v30.2.0

3.7.2 Test Cases Overview

I've outlined the key test cases for the RetailPulse application covering critical user flows from authentication to bill generation. These tests include both positive (happy path) and negative (error handling) scenarios.

Backend Test Cases

1. HEALTH CHECK API

Endpoint: `GET /health`

- **Positive Test**: Verify endpoint returns 200 status with `{"status": "ok"}`
- **Negative Test**: Verify no database changes occur during health check

---

2. USER AUTHENTICATION

Endpoint: `POST /auth/login`

- **Positive Test**: Valid credentials return 200 with access_token
- **Negative Test**: Invalid credentials return 401 with error message
- **Negative Test**: Missing credentials return 400 validation error

Endpoint: `POST /auth/register`

- **Positive Test**: Valid user registration returns 201 without password in response
- **Negative Test**: Duplicate email returns 409 conflict error
- **Negative Test**: Weak password returns 400 with validation message

---

3. INVENTORY MANAGEMENT

Endpoint: `POST /api/items`

- **Positive Test**: Valid item payload creates item and returns 201 with item ID
- **Negative Test**: Duplicate SKU returns 400 error
- **Negative Test**: Negative stock quantity returns 400 validation error

Endpoint: `GET /api/items`

- **Positive Test**: Returns paginated list of items with 200
- **Negative Test**: Invalid page number returns 400 or returns empty list

---

4. BILL GENERATION

Function: `calculate_line_total(quantity, unit_price, discount)`

- **Positive Test**: Correct calculation: (quantity × unit_price) - discount
- **Negative Test**: Zero or negative quantity returns zero or validation error

Endpoint: `POST /api/bills`

- **Positive Test**: Valid bill payload creates bill and returns 201 with bill ID
- **Negative Test**: Bill with insufficient stock returns 400 with error message

---

Frontend Test Cases

5. LOGIN FLOW

Component: `LoginPage`

- **Positive Test**: Valid credentials redirect to dashboard and store token
- **Negative Test**: Invalid credentials show error message and do not redirect
- **Negative Test**: Empty form submission shows validation errors

---

6. DASHBOARD

Component: `Dashboard`

- **Positive Test**: Dashboard loads and displays stats widgets with data
- **Negative Test**: API failure shows error message with retry option

---

7. CREATE BILL PAGE

Component: `CreateBillPage`

- **Positive Test**: Adding line items updates subtotal and grand total correctly
- **Negative Test**: Submitting bill with empty items shows validation error

---

4. Use Cases

4.1 Core System Use Cases

4.1.1 UC-01: Sales User Processes a Customer Order

- **Actor**: Sales
- **Goal**: To efficiently create a bill for a customer, ensuring accurate product selection and inventory deduction.
- **Flow**:
  a. The Sales user logs into the system and navigates to the "Create Bill" interface for their assigned store.
  b. They search for a product by name or code. The system displays available batches of the product, including expiry dates and current quantities in stock.
  c. The Sales user selects the desired product batch and specifies the quantity. The system validates that the requested quantity does not exceed the available stock in that batch.
  d. They can add multiple products to the same bill, repeating the search and selection process.
  e. Optionally, the Sales user enters customer details (e.g., name, contact information).
  f. Upon confirming the order, the Sales user clicks "Generate Bill".
  g. The system records the bill, automatically deducts the sold quantity from the specific inventory batch, and generates a printable PDF of the bill, customized with the store's details.
- **Alternative**: If the requested quantity for a product exceeds the available stock in the selected batch, the system displays an error message, prompting the user to adjust the quantity or select a different batch.

4.1.2 UC-02: Stock Management and Inter-Store Transfers

- **Actors**: Stockist, Company Stockist
- **Goal**: To maintain accurate inventory levels within stores and facilitate efficient product movement between locations.
- **Flow (Stockist - Receiving Inventory)**:
  a. A Stockist logs in and accesses the "Receive Stock" function within their assigned store's inventory management module.
  b. They select a product from the master catalog and enter details for the new shipment, including batch number, manufacturing ID, expiry date, quantity received, cost price, and selling price.
  c. The Stockist assigns a specific physical `location` (e.g., "Storeroom Rack 5") and `storage_category` (e.g., "Cold Storage") for the received batch.
  d. Upon saving, the new batch is added to the store's inventory and becomes available for sale or internal movement.
- **Flow (Company Stockist - Transferring Inventory Between Stores)**:
  a. A Company Stockist identifies a need to transfer products between stores (e.g., Store A has surplus, Store B has a shortage).
  b. They initiate a "New Store Transfer," selecting the product, the "From Store" (e.g., Store A), and the "To Store" (e.g., Store B), along with the quantity to transfer.
  c. The system marks the transferred quantity as "In Transit" in the "From Store's" inventory.
  d. Once the products arrive at the "To Store," a Stockist at that store confirms receipt in the system.
  e. The system then deducts the stock from the "From Store's" inventory and adds it to the "To Store's" inventory, updating locations as necessary.

4.1.3 UC-03: Company Admin Manages System Resources

- **Actor**: Company Admin
- **Goal**: To oversee and configure the entire system, including managing stores, users, and the master product catalog.
- **Flow (Managing Stores)**:
  a. The Company Admin logs in and navigates to the "Store Management" section.
  b. They can add new stores by providing details such as name, address, and contact information.
  c. Existing store details can be updated, or stores can be deactivated as needed.
- **Flow (Managing Users)**:
  a. The Company Admin accesses the "User Management" section.
  b. They can create new user accounts, providing a username, password, and full name.
  c. Crucially, the Admin assigns one or more `roles` to the new user (e.g., 'Store Manager', 'Sales') and associates them with one or more `stores` via the `user_roles` and `user_stores` junction tables.
  d. Existing user accounts can be updated (e.g., changing roles, store assignments) or deactivated.
- **Flow (Managing Product Master Catalog)**:
  a. The Company Admin goes to the "Product Master List" section.
  b. They can add new products to the central catalog, specifying details like name, category, manufacturer, and description.
  c. Existing product details can be updated, or products can be marked as inactive if they are no longer sold.

5. Edge Cases and Error Handling

5.1 Overview

This document explains how RetailPulse handles common problems and unexpected situations.

5.2 1. Login and Access Problems

- **Problem**: User tries to log in with no username or password.
- **Solution**: The system shows a "Username and password are required" message.
- **Problem**: User enters the wrong username or password.
- **Solution**: The system shows an "Invalid username or password" message.
- **Problem**: A deactivated user tries to log in.
- **Solution**: The system shows a "Your account has been deactivated" message.
- **Problem**: A `Store Manager` tries to access a `Company Admin`-only page.
- **Solution**: The system shows an "Insufficient permissions" message and denies access.

5.3 2. Bill Creation Problems

- **Problem**: User tries to create a bill with no products.
- **Solution**: The system requires at least one product to be in the bill.
- **Problem**: A product is not found in the store's inventory or is inactive.
- **Solution**: The system shows a "Product not found or inactive" message.
- **Problem**: The quantity requested is more than the available stock in the selected batch.
- **Solution**: The system shows an error message like "Only 5 units available in this batch".
- **Problem**: The quantity is zero or a negative number.
- **Solution**: The system requires the quantity to be greater than zero.
- **Problem**: The database connection is lost while creating a bill.
- **Solution**: The entire transaction is cancelled (rolled back) to prevent partial data. The user is asked to try again.

5.4 3. Inventory Management Problems

- **Problem**: `Company Admin` tries to create a product with a name that already exists.
- **Solution**: The system shows a "Product with this name already exists" message.
- **Problem**: `Stockist` tries to adjust stock to a negative quantity.
- **Solution**: The system shows a "Stock quantity cannot be negative" message. The database has a `CHECK` constraint to enforce this.

5.5 4. User Management Problems

- **Problem**: `Company Admin` tries to create a user with a username that already exists.
- **Solution**: The system shows a "Username already exists" message.
- **Problem**: `Company Admin` tries to deactivate their own account.
- **Solution**: The system prevents this to ensure there's always at least one active super admin.

5.6 5. Multi-Store and Advanced Inventory Edge Cases

- **Problem**: A `Sales` user tries to sell a product from a batch that has expired.
- **Solution**: The system should not show expired batches in the search results on the billing page. If an API call is made directly, the backend should reject the request with a "Cannot sell from expired batch" error.
- **Problem**: Two `Sales` users try to sell the last item of a batch at the same time (a race condition).
- **Solution**: The system uses database-level transaction isolation. The first transaction to commit will succeed. The second transaction will fail when it attempts to update the quantity, as the stock will already be zero. It will receive an error like "Insufficient stock".
- **Problem**: A branch store loses internet connection to the central server.
- **Solution**: In the current design, the system requires a live connection to the central database. If the connection is lost, the store's terminal will not be able to create new bills or look up inventory. (A future enhancement could be a limited "offline mode" that syncs data once the connection is restored).
- **Problem**: A `Company Admin` reassigns a `Store Manager` from Store A to Store B.
- **Solution**: The user's `store_id` is updated in the `users` table. The change should take effect on their next login. When they log in again, their session data will be for Store B, and they will no longer have access to Store A's data or settings.

6. Future Enhancements

This document outlines 5 simple future enhancements for the RetailPulse system with their implementation approach.

6.1 1. Barcode Scanner Integration

Description:
Enable quick product search and billing using barcode scanners for faster checkout.

Implementation:

- Add barcode field to Product schema
- Integrate barcode scanner library (e.g., QuaggaJS for web)
- Create barcode input API endpoint that searches products by barcode
- Update Sales UI with barcode scan button
- Auto-add scanned products to cart

Workflow:

```
flowchart LR
    A[Scan Barcode] --> B[Barcode Scanner Library]
    B --> C[Send Barcode to API]
    C --> D{Product Found?}
    D -->|Yes| E[Add to Cart]
    D -->|No| F[Show Error]
    E --> G[Continue Billing]
```

6.2 2. WhatsApp Bill Sharing

Description:
Send bill PDFs directly to customers via WhatsApp after purchase.

Implementation:

- Integrate WhatsApp Business API or third-party service (e.g., Twilio)
- Add customer phone number field in billing form (optional)
- Generate bill PDF using existing system
- Create API endpoint to send PDF via WhatsApp
- Add "Send via WhatsApp" button on bill confirmation screen

Workflow:

```
flowchart LR
    A[Bill Generated] --> B[Enter Customer Phone]
    B --> C[Click Send WhatsApp]
    C --> D[Generate PDF]
    D --> E[Call WhatsApp API]
    E --> F[Send PDF to Customer]
    F --> G[Show Success Message]
```

6.3 3. Customer Management

Description:
Store customer contact details and view their complete purchase history.

Implementation:

- Create Customer schema (name, phone, email, address)
- Add customer selection/creation in billing flow
- Link bills to customer records via foreign key
- Create Customer Management page for CRUD operations
- Build customer profile page showing all past bills
- Add search and filter functionality

Workflow:

```
flowchart TD
    A[Create Bill] --> B{Customer Exists?}
    B -->|Yes| C[Select Customer]
    B -->|No| D[Create New Customer]
    C --> E[Link to Bill]
    D --> E
    E --> F[Save Bill]
    F --> G[View Customer Profile]
    G --> H[Display Purchase History]
```

6.4 4. Low Stock Alerts

Description:
Automatic notifications when products fall below minimum stock level.

Implementation:

- Add minimum_stock_level field to Product schema
- Create background job/cron to check stock levels daily
- Integrate email service (e.g., SendGrid) or SMS service
- Build notification template for low stock alerts
- Send alerts to Store Manager and Company Stockist
- Add notification preferences in user settings

Workflow:

```
flowchart LR
    A[Daily Cron Job] --> B[Check Inventory Levels]
    B --> C{Stock < Minimum?}
    C -->|Yes| D[Generate Alert]
    C -->|No| E[Continue]
    D --> F[Send Email/SMS]
    F --> G[Notify Manager]
    F --> H[Notify Company Stockist]
```

6.5 5. Sales Analytics Dashboard

Description:
Visual dashboard showing sales trends, top products, and revenue insights.

Implementation:

- Create analytics API endpoints aggregating sales data
- Use charting library (e.g., Chart.js or Recharts)
- Build dashboard page with key metrics:
- Total sales and revenue (daily/weekly/monthly)
- Top selling products
- Sales by product category
- Store-wise performance comparison
- Add date range filters
- Role-based access (Company Admin sees all stores, Manager sees own store)

Workflow:

```
flowchart TD
    A[User Opens Dashboard] --> B[Select Date Range]
    B --> C[API Aggregates Sales Data]
    C --> D[Calculate Metrics]
    D --> E[Generate Charts]
    E --> F[Display Revenue Trends]
    E --> G[Display Top Products]
    E --> H[Display Store Comparison]
    F --> I[Interactive Dashboard]
    G --> I
    H --> I
```

6.6 6. Inter-Store Transfers

Description: Design and validate a formal process for moving inventory between stores, including transfer requests, approvals, and receipt confirmation.

Implementation (high level):

- Add transfer model and endpoints: create transfer request, approve/dispatch, confirm receipt.
- Add transfer states (requested, approved, dispatched, received, cancelled) and audit trail for each state change.
- Handle stock reservations on dispatch to prevent double-selling while a transfer is in progress.
- Add role-based permissions for initiating and approving transfers.
- Build UI flows for initiating a transfer, tracking status, and confirming receipt.
- Add automated tests for concurrency and edge cases (e.g., partial receipts, transfer cancellations).

Workflow:

```
flowchart LR
    A[Initiate Transfer] --> B[Create Transfer Request]
    B --> C{Approved?}
    C -->|No| D[Cancel or Request Changes]
    C -->|Yes| E[Reserve Stock at Source]
    E --> F[Dispatch from Source]
    F --> G[In Transit]
    G --> H[Receive at Destination]
    H --> I{Receipt OK?}
    I -->|Partial| J[Partial Receipt: Adjust Inventory & Notify]
    I -->|Complete| K[Confirm Receipt & Finalize Transfer]
    K --> L[Update Inventory & Release Reservations]
    J --> L
    D --> M[Mark Transfer Cancelled]
```

7. Development Workflow & Automated Testing

This document describes the development workflow, branching strategy, code merge process, and automated testing procedures for the RetailPulse project.

7.1 Branch Structure

The project maintains the following branch hierarchy:

```
main (production)
  └── staging (pre-production)
       └── dev (development)
            ├── feature/* (feature branches)
            ├── feat/* (feature branches)
            └── fix/* (bugfix branches)
```

7.1.1 Current Active Branches

- **main**: Production-ready code
- **staging**: Pre-production testing environment
- **dev**: Active development branch
- **feat/ci-cd-setup**: CI/CD pipeline implementation
- **feat/ui-ux-enhancements-forms**: UI/UX improvements
- **feature/apis-fixes-and-updates**: API enhancements and fixes
- **feature/company-admin-dashboard**: Admin dashboard features
- **feature/database-schema**: Database structure changes
- **feature/project-setup**: Initial project configuration
- **feature/tests**: Test suite development

7.2 Code Merge Workflow

7.2.1 1. Feature Branch Creation

```
# Create a new feature branch from dev
git checkout dev
git pull origin dev
git checkout -b feature/your-feature-name
```

7.2.2 2. Development Process

- Make incremental commits with clear, descriptive messages
- Follow conventional commit format: `feat:`, `fix:`, `refactor:`, `chore:`, `docs:`
- Keep commits focused on single logical changes
- Write tests for new features

7.2.3 3. Code Quality Checks

Before creating a pull request:

```
# Backend: Run tests and linting
pytest --cov=app --cov-report=term-missing --cov-report=html -v

# Frontend: Run tests and linting
```

```
npm test -- --coverage
npm run lint
```

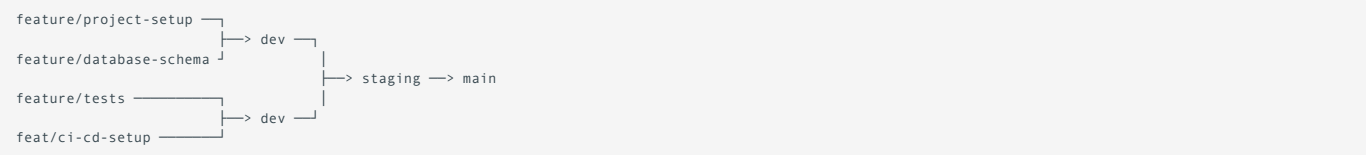
7.2.4 4. Pull Request Process

1. **Push to Remote**: Push your feature branch to GitHub

```
git push origin feature/your-feature-name
```

1. **Create Pull Request**:
2. Target branch: `dev`
3. Provide clear title and description
4. Reference related issues
5. Add reviewers
6. **Code Review**:
7. Address reviewer feedback
8. Make requested changes
9. Update PR with new commits
10. **CI/CD Checks**:
11. Automated tests must pass
12. Code coverage requirements must be met
13. Security scans (Semgrep) must pass
14. **Merge**:
15. Squash and merge for clean history
16. Delete feature branch after merge

7.2.5 5. Merge Flow Diagram



7.3 Recent Merge History

7.3.1 Development Branch Merges

1. **feat/ci-cd-setup** → **dev** (PR #9)
2. Added PostgreSQL service to CI/CD pipeline
3. Implemented database migration steps
4. Added automatic deployment to EC2
5. **feature/apis-fixes-and-updates** → **feat/ui-ux-enhancements-forms** (PR #8)
6. Implemented user management endpoints
7. Enhanced authentication and billing features
8. Refactored role management
9. **feature/tests** → **dev** (PR #7)
10. Added comprehensive test suite

11. Configured test database
12. Implemented test fixtures
13. **feature/company-admin-dashboard** → **dev** (Multiple PRs: #2, #3, #4, #5, #6)
14. Added patch endpoints for bills and categories
15. Implemented password change functionality
16. Added comprehensive CRUD operations
17. **feature/database-schema** → **dev** (PR #2)
18. Migrated to UUID primary keys
19. Added database seeding script
20. Fixed schema relationships
21. **feature/project-setup** → **dev** (PR #1)
22. Initial project structure
23. Added pytest configuration
24. Created requirements.txt

7.4 Automated Testing

7.4.1 Backend Testing (Python/FastAPI)

Test Configuration

Framework: pytest 9.0.0 with pytest-cov 7.0.0 and pytest-asyncio 1.3.0

Test Execution Results

```
========================================== test session starts ==========================================
platform darwin -- Python 3.13.7, pytest-9.0.0, pluggy-1.6.0 -- /Users/rohitagarwal/retailpulse/.venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/rohitagarwal/retailpulse/retailpulse-backend
configfile: pyproject.toml
testpaths: tests
plugins: anyio-4.11.0, asyncio-1.3.0, cov-7.0.0
asyncio: mode=Mode.AUTO, debug=False, asyncio_default_fixture_loop_scope=function, asyncio_default_test_loop_scope=function
collected 61 items

tests/test_auth.py::test_login_success PASSED                                                [  1%]
tests/test_auth.py::test_login_invalid_username PASSED                                       [  3%]
tests/test_auth.py::test_login_invalid_password PASSED                                       [  4%]
tests/test_auth.py::test_login_missing_fields PASSED                                         [  6%]
tests/test_auth.py::test_login_empty_credentials PASSED                                      [  8%]
tests/test_bills.py::test_get_bills PASSED                                                   [  9%]
tests/test_categories.py::test_create_category PASSED                                        [ 11%]
tests/test_categories.py::test_create_category_invalid_data PASSED                           [ 13%]
tests/test_categories.py::test_get_categories_empty PASSED                                   [ 14%]
tests/test_categories.py::test_get_categories_with_data PASSED                               [ 16%]
tests/test_categories.py::test_get_category_by_id PASSED                                     [ 18%]
tests/test_categories.py::test_get_category_not_found PASSED                                 [ 19%]
tests/test_categories.py::test_update_category PASSED                                        [ 21%]
tests/test_categories.py::test_update_category_not_found PASSED                              [ 22%]
tests/test_categories.py::test_delete_category PASSED                                        [ 24%]
tests/test_categories.py::test_delete_category_not_found PASSED                              [ 26%]
tests/test_health.py::test_health_check PASSED                                               [ 27%]
tests/test_inventories.py::test_create_inventory PASSED                                      [ 29%]
tests/test_inventories.py::test_create_inventory_invalid_product PASSED                      [ 31%]
tests/test_inventories.py::test_create_inventory_invalid_store PASSED                        [ 32%]
tests/test_inventories.py::test_get_inventories_empty PASSED                                 [ 34%]
tests/test_inventories.py::test_get_inventories_with_data PASSED                             [ 36%]
tests/test_inventories.py::test_get_inventory_by_id PASSED                                   [ 37%]
tests/test_inventories.py::test_get_inventory_not_found PASSED                               [ 39%]
tests/test_inventories.py::test_update_inventory PASSED                                      [ 40%]
tests/test_inventories.py::test_update_inventory_not_found PASSED                            [ 42%]
tests/test_inventories.py::test_delete_inventory PASSED                                      [ 44%]
tests/test_inventories.py::test_delete_inventory_not_found PASSED                            [ 45%]
tests/test_products.py::test_create_product PASSED                                           [ 47%]
tests/test_products.py::test_create_product_invalid_category PASSED                          [ 49%]
tests/test_products.py::test_create_product_invalid_data PASSED                              [ 50%]
tests/test_products.py::test_get_products_empty PASSED                                       [ 52%]
tests/test_products.py::test_get_products_with_data PASSED                                   [ 54%]
tests/test_products.py::test_get_product_by_id PASSED                                        [ 55%]
tests/test_products.py::test_get_product_not_found PASSED                                    [ 57%]
```

```
tests/test_products.py::test_update_product PASSED                                                  [ 59%]
tests/test_products.py::test_update_product_not_found PASSED                                        [ 60%]
tests/test_products.py::test_delete_product PASSED                                                  [ 62%]
tests/test_products.py::test_delete_product_not_found PASSED                                        [ 63%]
tests/test_stores.py::test_create_store PASSED                                                      [ 65%]
tests/test_stores.py::test_create_store_invalid_data PASSED                                         [ 67%]
tests/test_stores.py::test_get_stores_empty PASSED                                                  [ 68%]
tests/test_stores.py::test_get_stores_with_data PASSED                                              [ 70%]
tests/test_stores.py::test_get_store_by_id PASSED                                                   [ 72%]
tests/test_stores.py::test_get_store_not_found PASSED                                               [ 73%]
tests/test_stores.py::test_update_store PASSED                                                      [ 75%]
tests/test_stores.py::test_update_store_not_found PASSED                                            [ 77%]
tests/test_stores.py::test_update_store_invalid_data PASSED                                         [ 78%]
tests/test_stores.py::test_delete_store PASSED                                                      [ 80%]
tests/test_stores.py::test_delete_store_not_found PASSED                                            [ 81%]
tests/test_users.py::test_create_user PASSED                                                        [ 83%]
tests/test_users.py::test_create_user_duplicate_username PASSED                                     [ 85%]
tests/test_users.py::test_create_user_invalid_data PASSED                                           [ 86%]
tests/test_users.py::test_get_users_empty PASSED                                                    [ 88%]
tests/test_users.py::test_get_users_with_data PASSED                                                [ 90%]
tests/test_users.py::test_get_user_by_id PASSED                                                     [ 91%]
tests/test_users.py::test_get_user_not_found PASSED                                                 [ 93%]
tests/test_users.py::test_update_user PASSED                                                        [ 95%]
tests/test_users.py::test_update_user_not_found PASSED                                              [ 96%]
tests/test_users.py::test_delete_user PASSED                                                        [ 98%]
tests/test_users.py::test_delete_user_not_found PASSED                                              [100%]

============================================== tests coverage ==============================================
_________________________________ coverage: platform darwin, python 3.13.7-final-0 _________________________________

Name                                  Stmts   Miss  Cover   Missing
-------------------------------------------------------------------
app/core/config.py                       10      0   100%
app/core/security.py                      9      0   100%
app/db/seed.py                           33     33     0%   1-70
app/db/session.py                        13      5    62%   13-18
app/main.py                              26      1    96%   40
app/models/bill.py                       24      0   100%
app/models/bill_product.py               19      0   100%
app/models/category.py                   15      0   100%
app/models/inventory.py                  25      0   100%
app/models/inventory_movement.py         20      0   100%
app/models/permission.py                 14      0   100%
app/models/product.py                    20      0   100%
app/models/role.py                       15      0   100%
app/models/role_permission.py            16      0   100%
app/models/store.py                      17      0   100%
app/models/user.py                       19      0   100%
app/models/user_role.py                  16      0   100%
app/models/user_store.py                 16      0   100%
app/routers/auth.py                      38      7    82%   38, 86-95
app/routers/bills.py                    187    152    19%   21-43, 61-72, 77-111, 116-146, 151-157, 167-192, 204-239, 250-285, 295-311
app/routers/categories.py                53      8    85%   60-69
app/routers/inventories.py               76     16    79%   34, 36, 44-46, 78-82, 85-89, 93, 100, 102
app/routers/products.py                  59      2    97%   96, 110
app/routers/roles.py                     11      2    82%   13-14
app/routers/stores.py                    69     14    80%   19, 30-38, 90-97, 116-118
app/routers/users.py                    239    157    34%   32, 73-81, 89-133, 205-207, 219-226, 234-288, 295-331, 360-378, 386-406, 414-464, 489, 500-508
app/schemas/auth.py                      10      0   100%
app/schemas/bill.py                      62     12    81%   25-27, 32-34, 56-58, 63-65
app/schemas/bill_product.py              52     12    77%   20-22, 27-29, 47-49, 54-56
app/schemas/category.py                  16      0   100%
app/schemas/inventory.py                 41      2    95%   22, 45
app/schemas/product.py                   22      0   100%
app/schemas/roles.py                      8      0   100%
app/schemas/store.py                     14      0   100%
app/schemas/user.py                      32      0   100%
app/util/current_user.py                 24     14    42%   16-37
-------------------------------------------------------------------
TOTAL                                  1340    437    67%
Coverage HTML written to dir htmlcov
============================================ 61 passed in 2.45s ============================================
```

Test Summary

- **Total Tests**: 61
- **Passed**: 61 (100%)
- **Failed**: 0
- **Overall Coverage**: 67%
- **Execution Time**: 2.45s

Test Modules

1. **test_auth.py** (5 tests)
2. Login success/failure scenarios
3. Invalid credentials handling
4. Missing fields validation
5. **test_bills.py** (1 test)
6. Bill retrieval functionality
7. **test_categories.py** (10 tests)
8. CRUD operations for categories
9. Validation and error handling
10. **test_inventories.py** (11 tests)
11. Inventory management operations
12. Foreign key validation
13. **test_products.py** (11 tests)
14. Product CRUD operations
15. Category relationship validation
16. **test_stores.py** (11 tests)
17. Store management operations
18. Data validation
19. **test_users.py** (11 tests)
20. User management operations
21. Duplicate username prevention
22. **test_health.py** (1 test)
23. API health check endpoint

Coverage by Module

Module	Coverage	Status
app/core/config.py	100%	✅ Excellent
app/core/security.py	100%	✅ Excellent
app/models/*	100%	✅ Excellent
app/schemas/*	77-100%	✅ Good
app/routers/products.py	97%	✅ Excellent
app/main.py	96%	✅ Excellent
app/schemas/inventory.py	95%	✅ Excellent
app/routers/categories.py	85%	✅ Good
app/routers/auth.py	82%	✅ Good
app/routers/roles.py	82%	✅ Good
app/schemas/bill.py	81%	✅ Good
app/routers/stores.py	80%	✅ Good
app/routers/inventories.py	79%	⚠️ Needs Improvement
app/schemas/bill_product.py	77%	⚠️ Needs Improvement
app/db/session.py	62%	⚠️ Needs Improvement
app/util/current_user.py	42%	❌ Low Coverage
app/routers/users.py	34%	❌ Low Coverage
app/routers/bills.py	19%	❌ Low Coverage
app/db/seed.py	0%	❌ Not Tested

7.4.2 Frontend Testing (React/Jest)

Test Configuration

Framework: Jest 30.2.0 with Testing Library

Configuration (`jest.config.js`):

```
export default {
  testEnvironment: "jsdom",
  setupFilesAfterEnv: ["<rootDir>/jest.setup.js"],
  moduleNameMapper: {
    "\\.(css|less|scss|sass)$": "identity-obj-proxy",
    "\\.(jpg|jpeg|png|gif|svg)$": "<rootDir>/__mocks__/fileMock.js",
  },
  transform: {
    "^.+\\.[tj]sx?$": [
      "babel-jest",
      {
        presets: [
          ["@babel/preset-env", { targets: { node: "current" } }],
          ["@babel/preset-react", { runtime: "automatic" }],
        ],
      },
    ],
  },
  collectCoverageFrom: [
    "src/**/*.{js,jsx}",
    "!src/main.jsx",
    "!src/**/*.test.{js,jsx}",
  ],
  coverageThreshold: {
```

```
    global: {
      branches: 50,
      functions: 50,
      lines: 50,
      statements: 50,
    },
  },
};
```

Test Execution Results

```
 PASS  src/tests/init.test.jsx
 PASS  src/tests/ProtectedRoute.test.jsx
 PASS  src/tests/Login.test.jsx

Test Suites: 3 passed, 3 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        4.17 s
```

Test Summary

- **Total Test Suites**: 3
- **Total Tests**: 12
- **Passed**: 12 (100%)
- **Failed**: 0
- **Overall Coverage**: 3.64%
- **Execution Time**: 4.17s

Test Modules

1. **init.test.jsx** (1 test)
2. App component health check
3. **ProtectedRoute.test.jsx** (6 tests)
4. Loading state handling
5. Authentication redirects
6. Role-based access control
7. Authorization checks
8. **Login.test.jsx** (5 tests)
9. Form rendering
10. User input handling
11. Login success with role-based redirects
12. Error message display

Coverage Analysis

Current Coverage: 3.64% (statements), 2.24% (branches), 2.77% (functions), 3.74% (lines)

Coverage Thresholds: 50% (global requirement - NOT MET)

Component Category	Coverage	Status
Core Components	30.3%	❌ Below Threshold
Authentication	45.94%	❌ Below Threshold
Protected Routes	100%	✅ Excellent
Constants	100%	✅ Excellent
Contexts	65.51%	⚠️ Needs Improvement
Billing Components	0%	❌ Not Tested
Dashboard Components	0%	❌ Not Tested
Forms	0%	❌ Not Tested
Services	0%	❌ Not Tested
Utilities	0%	❌ Not Tested

Areas Requiring Additional Tests

1. **High Priority** (0% coverage):
2. Billing components and services
3. Dashboard components
4. Form components
5. API services
6. Utility functions
7. **Medium Priority** (Low coverage):
8. Layout component
9. Authentication context (expand coverage)
10. Page components

7.5 Continuous Integration/Continuous Deployment (CI/CD)

7.5.1 CI/CD Pipeline (GitHub Actions)

The project uses GitHub Actions for automated testing and deployment:

Pipeline Triggers

- Push to `dev` branch

Pipeline Steps

1. **Code Checkout**: Clone repository
2. **Environment Setup**:
3. Install Python 3.13
4. Install Node.js 18+
5. Install dependencies
6. **Database Setup**:

7. Start PostgreSQL service
8. Run migrations
9. **Testing**:
10. Run backend tests with coverage
11. Run frontend tests with coverage
12. Generate coverage reports
13. **Security Scanning**:
14. Run Semgrep security analysis
15. Check for vulnerabilities
16. **Deployment** (on push to dev):
17. Deploy to EC2 instance
18. Restart services

7.6 Best Practices

7.6.1 Code Quality

1. **Write Tests First**: Follow TDD principles when possible
2. **Maintain Coverage**: Aim for 80%+ code coverage
3. **Test Edge Cases**: Include error handling and boundary conditions
4. **Use Fixtures**: Reuse test data with pytest fixtures
5. **Mock External Dependencies**: Isolate unit tests from external services

7.6.2 Git Workflow

1. **Commit Messages**: Use conventional commit format

```
feat: add user authentication endpoint
fix: resolve inventory update bug
refactor: improve database query performance
docs: update API documentation
test: add tests for billing module
```

1. **Branch Naming**: Use descriptive branch names

```
feature/add-user-roles
fix/inventory-calculation-bug
refactor/database-schema
```

1. **Pull Request**: Include:
2. Clear description of changes
3. Related issue numbers
4. Test results
5. Screenshots (for UI changes)

8. Test Files

This document describes the test files added to the project for development and testing purposes.

8.1 Backend Test File

A `test.txt` file has been added to the backend directory (`retailpulse-backend/`) for testing backend-related functionality.

8.1.1 Semgrep Security Scan Results

```
┌──────────────┐
│ Scan Status │
└──────────────┘
  Scanning 54 files tracked by git with 1062 Code rules:

  Language      Rules   Files          Origin      Rules
 ─────────────────────────────        ──────────────────
  <multilang>      48      54          Community    1062
  python          243      46

┌─────────────────┐
│ 1 Code Finding │
└─────────────────┘

    /src/app/main.py
    ❯❯ python.fastapi.security.wildcard-cors.wildcard-cors
          CORS policy allows any origin (using wildcard '*'). This is insecure and should be avoided.
          Details: https://sg.run/KxApY

           20┆ allow_origins=["*"],

┌──────────────┐
│ Scan Summary │
└──────────────┘
✅ Scan completed successfully.
 • Findings: 1 (1 blocking)
 • Rules run: 291
 • Targets scanned: 54
 • Parsed lines: ~100.0%
 • Scan skipped:
   ◦ Files matching .semgrepignore patterns: 10
 • Scan was limited to files tracked by git
 • For a detailed list of skipped files and lines, run semgrep with the --verbose flag
Ran 291 rules on 54 files: 1 finding.
```

8.2 Frontend Test File

A `test.txt` file has been added to the frontend directory (`retailpulse-frontend/`) for testing frontend-related functionality.

8.2.1 Semgrep Security Scan Results

```
┌──────────────┐
│ Scan Status │
└──────────────┘
  Scanning 65 files tracked by git with 1062 Code rules:

  Language      Rules   Files          Origin      Rules
 ─────────────────────────────        ──────────────────
  <multilang>      61      65          Community    1062
  js              156      54
  json              4       2
  html              1       1

┌──────────────┐
│ Scan Summary │
└──────────────┘
✅ Scan completed successfully.
 • Findings: 0 (0 blocking)
 • Rules run: 221
 • Targets scanned: 65
 • Parsed lines: ~100.0%
 • Scan skipped:
   ◦ Files matching .semgrepignore patterns: 3
 • Scan was limited to files tracked by git
 • For a detailed list of skipped files and lines, run semgrep with the --verbose flag
```

```
Ran 221 rules on 65 files: 0 findings.
(need more rules? `semgrep login` for additional free Semgrep Registry rules)

If Semgrep missed a finding, please send us feedback to let us know!
See https://semgrep.dev/docs/reporting-false-negatives/
```

8.3 Purpose

These files contain Semgrep security scan results for the RetailPulse project. Semgrep is a static analysis tool that scans code for security vulnerabilities, bugs, and anti-patterns.

8.3.1 Key Findings

- **Backend**: 1 security finding (CORS policy configuration)
- **Frontend**: 0 security findings
- **Overall**: The codebase maintains good security practices with minimal issues

For detailed information about the development workflow, code merge process, and comprehensive automated testing procedures, please refer to the Development Workflow & Testing documentation.