Requirement Specification Document

Project Name: Inventory Management System (IMS)
Technology Stack: Java, Spring Boot, RESTful APIs
Version: 1.0
Date: 03/10/2025

---

1. Introduction

1.1. Purpose

This document outlines the requirements for the backend code of an Inventory Management System (IMS). The backend will be developed using Java with Spring Boot, implementing a set of RESTful APIs for managing inventory operations such as item management, stock tracking, and order processing.

1.2. Scope

The backend will handle inventory data, user authentication, stock updates, and reporting. The APIs will be consumed by front-end applications or external services to interact with the inventory system.

1.3. Target Audience

This document is intended for developers, testers, and other stakeholders involved in the development of the IMS backend.

---

2. System Overview

The system is designed to manage and track the flow of inventory items within an organization. The backend will handle CRUD operations (Create, Read, Update, Delete) for products, inventory levels, and stock movements.

---

3. Functional Requirements

3.1. User Authentication and Authorization

- **Requirement:** Implement JWT (JSON Web Token) based authentication for secure API access.
- **Endpoints:**

    - POST /api/auth/login: Login endpoint to authenticate users and generate JWT tokens.
    - POST /api/auth/register: User registration endpoint.

3.2. Inventory Management

- **Requirement:** Implement CRUD operations for managing inventory items.
- **Endpoints:**
    - POST /api/inventory: Add a new inventory item (e.g., product name, description, price, stock quantity).
    - GET /api/inventory: Retrieve a list of all inventory items.
    - GET /api/inventory/{id}: Retrieve a single inventory item by its ID.
    - PUT /api/inventory/{id}: Update an existing inventory item (e.g., price, stock level).
    - DELETE /api/inventory/{id}: Delete an inventory item from the system.

3.3. Stock Management

- **Requirement:** Track stock levels and update inventory when stock is added or removed.
- **Endpoints:**
    - POST /api/stock/add: Add stock to an item in the inventory.
    - POST /api/stock/remove: Remove stock from an item in the inventory.
    - GET /api/stock/{id}: View the current stock level of a specific inventory item.

3.4. Order Processing

- **Requirement:** Manage orders, including creation, retrieval, and order status updates.
- **Endpoints:**
    - POST /api/orders: Create a new order.
    - GET /api/orders: Retrieve all orders.
    - GET /api/orders/{id}: Retrieve order details by ID.

    - PUT /api/orders/{id}: Update order status (e.g., processing, shipped, completed).
    - DELETE /api/orders/{id}: Cancel an order.

3.5. Reporting

- **Requirement:** Generate reports on stock levels, orders, and inventory.
- **Endpoints:**
    - GET /api/reports/inventory: Generate an inventory report showing current stock levels and product details.
    - GET /api/reports/orders: Generate a report showing order details and statuses.

---

4. Non-Functional Requirements

4.1. Performance Requirements

- API response time should be within 1 second for most operations.
- The system should support up to 5 concurrent users.

4.2. Security

- JWT token expiration should be set to 1 hour.
- Sensitive data such as passwords must be encrypted.
- All sensitive API endpoints must be protected with authentication and authorization checks.

4.3. Scalability

- The system should be designed to handle an increase in product inventory and orders.
- Use a scalable database solution (e.g., PostgreSQL, MySQL) with proper indexing for large datasets.

4.4. Data Integrity

- Ensure transactional integrity when updating stock levels or creating orders.
- Use database transactions to ensure consistency.

4.5. Availability

- The system should be highly available, with minimal downtime for updates or maintenance.
- Use logging and monitoring to track the health of the system.

---

5. API Design

5.1. Request and Response Format

- All APIs will follow the REST architectural style.
- The request and response format will be in JSON.
- Sample Response for a Successful API Call:

```
{
  "status": "success",
  "message": "Inventory item created successfully",
  "data": {
    "id": 1,
    "name": "Product A",
    "price": 100.00,
    "stock": 500
  }
}
```

5.2. Error Handling

- Standardized error responses with proper HTTP status codes.
- Example:

```
{
  "status": "error",
  "message": "Invalid input data",
  "error": "BadRequest"
}
```

---

6. Database Design

6.1. Entities

- **Product**: Stores information about each inventory item.
  - Fields: id, name, description, price, quantity_in_stock, created_at, created_by, updated_at, updated_by
- **Order**: Stores information about customer orders.
  - Fields: id, product_id, quantity, total_price, status, created_at, created_by, updated_at, updated_by

6.2. Relationships

- A product can have many stock movements.
- An order will reference a product and track quantities ordered.

---

7. Technology Stack

7.1. Backend Framework

- **Java 17+** for backend development.
- **Spring Boot 2.x** for creating the RESTful APIs.
- **Spring Security** for authentication and authorization.
- **Hibernate ORM** for database interaction.

7.2. Database

- **Relational Database (e.g., PostgreSQL, MySQL)** for storing inventory and order data.

7.3. Testing

- **JUnit** for unit testing.
- **Postman** or **Swagger** for API testing.

---

8. Assumptions and Constraints

- The system will assume the use of HTTPS for all production environments to ensure secure communication.
- The backend will not handle payment processing directly; it will interface with a payment gateway via third-party APIs if needed.

---

9. Appendices

9.1. Class Diagram

9.2. API Flow

9.1. Class Diagram

Class Diagram Overview

The class diagram will represent the entities and relationships within the backend code. Below are the key entities that will be part of the backend system, and their relationships:

- **User**: Represents users in the system.
- **Product**: Represents items in the inventory.
- **Order**: Represents customer orders.
- **StockMovement**: Represents changes to the stock levels.
- **AuthService**: Handles login, JWT generation, and validation.
- **InventoryService**: Handles CRUD operations for inventory items.
- **OrderService**: Handles order processing.

9.2. API Flow Diagrams

Here are the flow diagrams showing the process of various API calls:

9.2.1. User Login Flow

[User] -----> [POST /api/auth/login] -----> [AuthService] -----> [Generate JWT Token] -----> [Response: JWT]

9.2.2. Inventory Management Flow

[User] -----> [POST /api/inventory] -----> [InventoryService] -----> [Create Product in Database] -----> [Response: Success]

[User] -----> [GET /api/inventory] -----> [InventoryService] -----> [Retrieve List of Products] -----> [Response: List of Products]

[User] -----> [PUT /api/inventory/{id}] -----> [InventoryService] -----> [Update Product] -----> [Response: Success]

[User] -----> [DELETE /api/inventory/{id}] -----> [InventoryService] -----> [Delete Product] -----> [Response: Success]

9.2.3. Order Processing Flow

[User] -----> [POST /api/orders] -----> [OrderService] -----> [Create Order in Database] -----> [Response: Success]

[User] -----> [GET /api/orders] -----> [OrderService] -----> [Retrieve List of Orders] -----> [Response: List of Orders]

[User] -----> [PUT /api/orders/{id}] -----> [OrderService] -----> [Update Order Status] -----> [Response: Success]

[User] -----> [DELETE /api/orders/{id}] -----> [OrderService] -----> [Cancel Order] -----> [Response: Success]
