Smart Inventory & Sales Tracker (S.I.S.T.)

1) Story

A mid-sized retail chain, **NeoMart**, runs 8 local stores. They currently track products, stock, and sales in Excel. This causes issues:

- Wrong stock counts → lost sales
- No audit trail of who changed what
- No simple way to view daily sales or low-stock alerts
- Data inconsistencies due to manual edits

NeoMart wants a **Java-based Inventory & Sales Tracker** that uses a relational DB, supports **role-based actions** (Admin vs Cashier), logs all operations, and produces basic business reports

You are hired as the backend team to build the **first working version**.

---

2) What you will build

A **console-based Java application** (no UI frameworks required) that:

- Manages **Products, Stock, Customers, Orders, and Payments**
- Persists data in **MySQL / PostgreSQL** using **SQL + JDBC**
- Uses **layered architecture**
- Has **JUnit tests** for core services/validations
- Uses **Log4j** logging for audit/debug
- Uses **Git** with proper branching + commit discipline

---

3) Mandatory Tech/Concept Coverage

Core Java (must use)

- OOP (encapsulation, abstraction, interfaces)
- Collections (List/Map usage in service layer + validations)
- Enums (roles, order status, payment mode/status)
- Java Time API (`LocalDateTime` for timestamps)
- Custom exceptions + proper handling

- Input validation and defensive programming

SQL + JDBC (must use)

- Normalized schema (min 3NF where reasonable)
- CRUD using `PreparedStatement`
- Transactions (`commit/rollback`) in order placement
- Joins & aggregations for reporting queries
- Constraints: PK, FK, UNIQUE, NOT NULL, CHECK (if supported)

Git (must use)

- Feature branches
- Meaningful commits
- PR-style workflow even if solo (merge feature branch to main)

JUnit (must use)

- Unit tests for service-layer logic
- DAO tests optional (bonus)
- Assertions verifying behavior + edge cases

Log4j (must use)

- Separate loggers for app flow + DB errors (or separate categories)
- Levels: INFO/WARN/ERROR/DEBUG
- Log file output + console output
- **No `System.out.println()` for application flow** (only menu prompts / final display outputs)

Architecture + Quality (must use)

- Layered design + clean separation
- Centralized exception strategy
- **No SQL inside service classes** (DAO only)
- Config via properties file
- Code quality: naming, SRP, no massive classes

---

4) Functional Requirements

A) User Roles

ADMIN

- Add/update/deactivate products
- Stock adjustments (add/remove stock + reason)
- View reports

CASHIER

- Search products
- Create customer (optional)
- Create order + payment
- Print invoice (console output)

Authentication can be simple: username + role input (not full security).
Role gating: show/hide menu actions based on role.

B) Product Management

Each product must have:

- `productId` (auto)
- `sku` (unique)
- `name`
- `category`
- `unitPrice`
- `active` (true/false)
- `createdAt`, `updatedAt`

Operations:

- Create product
- Update product details
- Deactivate product (soft delete)
- Search product by name/category/sku

C) Inventory / Stock

Track stock per product:

- `productId` FK
- `availableQuantity`
- `reorderLevel`

Operations

- Increase stock (admin)
- Reduce stock (admin adjustment OR order placement)
- Low stock report: products where `availableQuantity <= reorderLevel`

Audit requirement: every admin stock change must also insert a row in `stock_adjustments` with `deltaQty` and `reason`.

D) Order Placement (Core Workflow)

An order includes:

- `orderId`
- customer info (name + phone OR anonymous)
- items (`productId`, `quantity`, `unitPriceAtPurchase`)
- `orderTotal`
- `status` (CREATED, PAID, CANCELLED)
- `createdAt`

Rules:

- Cannot order inactive products
- Cannot order quantity > available stock
- Must update stock when order is successfully placed
- Payment marks order as PAID
- **Must be transactional**: if any step fails → rollback everything

Design rule (to avoid inconsistent implementations)

✅ **Stock is decremented only after payment is SUCCESS.**
If payment fails → order remains CREATED and stock remains unchanged.

E) Payments

Payment details:

- `paymentId`
- `orderId` FK
- `mode` (CASH, CARD, UPI)
- `amount`
- `status` (SUCCESS, FAILED)
- `paidAt`

Rules

- Amount must equal orderTotal (v1)

F) Reporting Requirements (SQL-heavy)

Implement as menu options:

1. **Daily Sales Summary**
    - Date → total orders, total revenue, top 3 sold products
2. **Product Sales Report**
    - product-wise quantity sold + revenue (date range)
3. **Low Stock Report**
4. **Inactive Products List**
5. **Order Lookup**
    - by orderId OR by customer phone (sorted by `created_at DESC`)

Reports must use SQL joins/aggregations.

---

5) Layered Architecture (Required)

Use this structure (or very close):

- presentation/controller (console menus, input)
- service (business rules, validation, transactions)
- dao (JDBC code only)
- model (POJOs/entities)
- exception (custom exceptions)
- util (DB connection, validators, helpers)
- config/resources (db.properties, log4j2.xml)

Rule: UI layer never directly calls DAO.

---

6) Exception Handling Rules (Non-negotiable)

Create custom exceptions such as:

- `ValidationException`
- `EntityNotFoundException`
- `InsufficientStockException`
- `DatabaseOperationException`

Guidelines:

- Validate early in service layer
- Catch SQLExceptions in DAO and wrap into `DatabaseOperationException`
- Log errors with stack traces at ERROR level
- Show user-friendly message in console (no stack trace in UI)

---

7) JUnit Testing Requirements (Must Have)

You must include **at least 12 meaningful tests**, including these **minimum required** tests:

1. `shouldRejectNegativePrice()`
2. `shouldRejectPriceWithMoreThan2Decimals()`
3. `shouldRejectEmptySku()`
4. `shouldRejectDuplicateSku()`
5. `shouldDeactivateProductAndPreventOrdering()`
6. `shouldRejectOrderWithZeroItems()`
7. `shouldRejectOrderItemWithNonPositiveQty()`
8. `shouldThrowInsufficientStockWhenQuantityExceedsAvailable()`
9. `shouldComputeOrderTotalUsingUnitPriceAtSale()`
10. `shouldFailPaymentWhenAmountMismatch()`
11. `shouldRollbackWhenAnyStepFailsDuringPlaceOrder()` *(bonus if done via fake DAO / controlled failure)*
12. `shouldNormalizePhoneAndCreateOrFetchCustomer()`

Clear naming example:

- `shouldThrowInsufficientStockWhenQuantityExceedsAvailable()`

---

8) Git Requirements

Repository must show:

- main branch protected in spirit (don't commit directly)
- Feature branches like:
  - `feature/product-management`
  - `feature/order-flow`
  - `feature/report`

- **At least 20 commits** with meaningful messages
  ✅ `Add ProductService validation and exceptions`
  ❌ `final code, done, changes`

---

9) Submission Deliverables

You must submit:

- Source code (Git repo)
- `README.md`
- `schema.sql`
- `db.properties` template (no passwords committed)
- `log4j2.xml` or `log4j2.properties`
- JUnit test output (screenshots or console output)

---

10) Evaluation Rubric

- Architecture & separation of concerns: **25%**
- SQL schema + query quality + JDBC correctness: **20%**
- Order transaction correctness + rollback safety: **15%**
- Testing quality (JUnit): **15%**
- Logging & exception handling: **10%**
- Git hygiene + README quality: **10%**
- Code cleanliness (naming, SRP, duplication): **5%**

Starter Kit

None

```
neomart-sist/
├─ README.md
│  └─ HINT: Setup, DB steps, how to run, sample menu flows, and design decisions (esp. transaction timing).
│
├─ schema.sql
│  └─ HINT: Tables + constraints + seed data (8+ products). Normalize and enforce FK/UNIQUE/NOT NULL.
│
├─ pom.xml (or build.gradle)
│  └─ HINT: Dependencies: JDBC driver, Log4j2, JUnit5 + Surefire plugin.
│
├─ src/
│  ├─ main/
│  │  ├─ java/com/neomart/sist/
│  │  │  ├─ App.java
│  │  │  │  └─ HINT: Entry point. Show main menu, route to controllers, handle global errors.
│  │  │  │
│  │  │  ├─ controller/
│  │  │  │  ├─ ProductController.java
│  │  │  │  │  └─ HINT: Read input, call ProductService/InventoryService, print results. No SQL here.
│  │  │  │  ├─ OrderController.java
│  │  │  │  │  └─ HINT: Cashier flow: phone/anonymous → add items by SKU → checkout → invoice print.
│  │  │  │  └─ ReportController.java
│  │  │  │     └─ HINT: Read date/range input, call ReportService, print tabular report output.
│  │  │  │
│  │  │  ├─ service/
│  │  │  │  ├─ ProductService.java
│  │  │  │  │  └─ HINT: Validations + uniqueness + orchestration. No SQL strings.
│  │  │  │  ├─ InventoryService.java
│  │  │  │  │  └─ HINT: Admin stock adjust must update inventory + insert stock_adjustments with reason.
│  │  │  │  ├─ CustomerService.java
│  │  │  │  │  └─ HINT: Normalize phone, create/find customer, support getOrCreate by phone.
│  │  │  │  ├─ OrderService.java
│  │  │  │  │  └─ HINT: Transaction boundary: lock inventory, validate, insert order+items, payment, stock decrement,
commit.
│  │  │  │  └─ ReportService.java
│  │  │  │     └─ HINT: Validate date/range, call ReportDao queries, return DTOs/rows for printing.
│  │  │  │
│  │  │  ├─ dao/
│  │  │  │  ├─ ProductDao.java
│  │  │  │  │  └─ HINT: CRUD/search only. No validation.
│  │  │  │  ├─ InventoryDao.java
│  │  │  │  │  └─ HINT: Stock reads/updates. Ensure no negative quantity.
│  │  │  │  ├─ CustomerDao.java
│  │  │  │  │  └─ HINT: Create/find/search. Phone is UNIQUE.
│  │  │  │  ├─ OrderDao.java
│  │  │  │  │  └─ HINT: Insert order header + items + update status/total + fetch for invoice.
│  │  │  │  ├─ PaymentDao.java
│  │  │  │  │  └─ HINT: Insert payment + fetch by orderId.
│  │  │  │  ├─ StockAdjustmentDao.java
│  │  │  │  │  └─ HINT: Insert adjustment records (delta+reason). Optional query history (bonus).
│  │  │  │  └─ impl/
│  │  │  │     ├─ JdbcProductDao.java
│  │  │  │     ├─ JdbcInventoryDao.java
│  │  │  │     ├─ JdbcCustomerDao.java
│  │  │  │     ├─ JdbcOrderDao.java
│  │  │  │     ├─ JdbcPaymentDao.java
│  │  │  │     └─ JdbcStockAdjustmentDao.java
│  │  │  │        └─ HINT: JDBC only. Catch SQLException and throw DatabaseOperationException.
│  │  │  │
│  │  │  ├─ model/
│  │  │  │  ├─ Product.java
│  │  │  │  ├─ Inventory.java
│  │  │  │  ├─ Customer.java
```

```
|  |  |  |  ├─ Order.java
|  |  |  |  ├─ OrderItem.java
|  |  |  |  ├─ Payment.java
|  |  |  |  └─ report/ (optional DTOs)
|  |  |  |     └─ HINT: Report rows like DailySalesRow, TopProductRow for clean printing.
|  |  |  |
|  |  |  ├─ exception/
|  |  |  |  ├─ ValidationException.java
|  |  |  |  ├─ EntityNotFoundException.java
|  |  |  |  ├─ InsufficientStockException.java
|  |  |  |  └─ DatabaseOperationException.java
|  |  |  |
|  |  |  ├─ util/
|  |  |  |  ├─ DbConnectionFactory.java
|  |  |  |  |  └─ HINT: Single place to create Connections from db.properties.
|  |  |  |  ├─ InputUtil.java
|  |  |  |  |  └─ HINT: Safe input parsing for int/string/date.
|  |  |  |  ├─ ValidationUtil.java
|  |  |  |  |  └─ HINT: Reusable validators (sku, phone normalization, qty, price scale).
|  |  |  |  └─ MoneyUtil.java
|  |  |  |     └─ HINT: BigDecimal formatting + parsing (scale=2). No doubles.
|  |  |  |
|  |  |  └─ constants/
|  |  |     ├─ Role.java
|  |  |     ├─ OrderStatus.java
|  |  |     └─ PaymentMode.java
|  |  |
|  |  └─ resources/
|  |     ├─ db.properties
|  |     |  └─ HINT: Commit template only (no password). Use env/local config for real password.
|  |     └─ log4j2.xml
|  |        └─ HINT: Console + file appenders. INFO for flow, ERROR for stack traces.
|  |
|  └─ test/java/com/neomart/sist/service/
|     ├─ ProductServiceTest.java
|     ├─ OrderServiceTest.java
|     └─ ValidationUtilTest.java
|
```

Minimal bootstrap classes (copy/paste)

Java

```java
package com.autocare.wms;

import com.autocare.wms.controller.BookingController;
import com.autocare.wms.controller.InvoiceController;
import com.autocare.wms.controller.PartController;
import com.autocare.wms.controller.ReportController;
import com.autocare.wms.util.InputUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {
    private static final Logger log = LogManager.getLogger(App.class);

    public static void main(String[] args) {
```

```
        log.info("AutoCare WMS started");

        BookingController bookingController = new BookingController();
        InvoiceController invoiceController = new InvoiceController();
        PartController partController = new PartController();
        ReportController reportController = new ReportController();

        while (true) {
            System.out.println("\n=== AutoCare WMS ===");
            System.out.println("1. Bookings");
            System.out.println("2. Invoices");
            System.out.println("3. Parts (Admin)");
            System.out.println("4. Reports");
            System.out.println("0. Exit");

            int choice = InputUtil.readInt("Choose: ");
            switch (choice) {
                case 1 -> bookingController.menu();
                case 2 -> invoiceController.menu();
                case 3 -> partController.menu();
                case 4 -> reportController.menu();
                case 0 -> {
                    log.info("AutoCare WMS stopped");
                    System.out.println("Bye!");
                    return;
                }
                default -> System.out.println("Invalid option.");
            }
        }
    }
}
```

`DbConnectionFactory.java` (single place for connections)

Java

```
package com.neomart.sist.util;

import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public final class DbConnectionFactory {
    private static final Properties props = new Properties();

    static {
        try (InputStream in =
DbConnectionFactory.class.getClassLoader().getResourceAsStream("db.propertie
s")) {
            if (in == null) throw new IllegalStateException("db.properties
not found in resources/");
            props.load(in);
        } catch (Exception e) {
```

```
            throw new ExceptionInInitializerError("Failed to load
db.properties: " + e.getMessage());
        }
    }

    private DbConnectionFactory() {}

    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(
                    props.getProperty("db.url"),
                    props.getProperty("db.username"),
                    props.getProperty("db.password")
            );
        } catch (Exception e) {
            throw new RuntimeException("DB connection failed: " +
e.getMessage(), e);
        }
    }
}
```

db.properties (template)

```
db.url=jdbc:mysql://localhost:3306/autocare_wms
db.username=root
db.password=YOUR_PASSWORD
```

DB Schema

None

Save as **schema.sql**.

```
-- Create DB
CREATE DATABASE IF NOT EXISTS neomart_sist;
USE neomart_sist;

-- Products
CREATE TABLE products (
  product_id      BIGINT PRIMARY KEY AUTO_INCREMENT,
  sku             VARCHAR(50) NOT NULL UNIQUE,
  name            VARCHAR(120) NOT NULL,
  category        VARCHAR(60) NOT NULL,
  unit_price      DECIMAL(10,2) NOT NULL,
  active          BOOLEAN NOT NULL DEFAULT TRUE,
```

```
  created_at      DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at      DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

-- Inventory (1 row per product)
CREATE TABLE inventory (
  product_id         BIGINT PRIMARY KEY,
  available_quantity INT NOT NULL,
  reorder_level      INT NOT NULL DEFAULT 5,
  updated_at         DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  CONSTRAINT fk_inventory_product
    FOREIGN KEY (product_id) REFERENCES products(product_id)
    ON DELETE RESTRICT
);

-- Customers (optional for cashier, allow null customer on order by design
choice)
CREATE TABLE customers (
  customer_id  BIGINT PRIMARY KEY AUTO_INCREMENT,
  name         VARCHAR(120) NOT NULL,
  phone        VARCHAR(20) NOT NULL UNIQUE,
  created_at   DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- Orders
CREATE TABLE orders (
  order_id     BIGINT PRIMARY KEY AUTO_INCREMENT,
  customer_id  BIGINT NULL,
  order_total  DECIMAL(12,2) NOT NULL,
  status       VARCHAR(20) NOT NULL, -- CREATED, PAID, CANCELLED
  created_at   DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_orders_customer
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
    ON DELETE SET NULL
);

-- Order items
CREATE TABLE order_items (
  order_item_id        BIGINT PRIMARY KEY AUTO_INCREMENT,
  order_id             BIGINT NOT NULL,
  product_id           BIGINT NOT NULL,
  quantity             INT NOT NULL,
  unit_price_at_sale   DECIMAL(10,2) NOT NULL,
  line_total           DECIMAL(12,2) NOT NULL,
  CONSTRAINT fk_items_order
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
    ON DELETE CASCADE,
  CONSTRAINT fk_items_product
    FOREIGN KEY (product_id) REFERENCES products(product_id)
    ON DELETE RESTRICT
);

-- Payments (1 payment per order in v1)
```

```
CREATE TABLE payments (
  payment_id   BIGINT PRIMARY KEY AUTO_INCREMENT,
  order_id     BIGINT NOT NULL UNIQUE,
  mode         VARCHAR(20) NOT NULL, -- CASH, CARD, UPI
  amount       DECIMAL(12,2) NOT NULL,
  status       VARCHAR(20) NOT NULL, -- SUCCESS, FAILED
  paid_at      DATETIME NULL,
  CONSTRAINT fk_payments_order
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
    ON DELETE CASCADE
);

-- Stock adjustment audit (admin stock changes with reason)
CREATE TABLE stock_adjustments (
  adjustment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  product_id    BIGINT NOT NULL,
  delta_qty     INT NOT NULL, -- + for add, - for reduce
  reason        VARCHAR(200) NOT NULL,
  created_at    DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_adj_product
    FOREIGN KEY (product_id) REFERENCES products(product_id)
    ON DELETE RESTRICT
);

-- Seed sample products (8)
INSERT INTO products (sku, name, category, unit_price, active) VALUES
('SKU-1001', 'Eco Bottle 1L', 'Kitchen', 399.00, true),
('SKU-1002', 'Bamboo Toothbrush', 'Personal Care', 99.00, true),
('SKU-1003', 'Organic Cotton Towel', 'Home', 599.00, true),
('SKU-1004', 'LED Desk Lamp', 'Electronics', 1299.00, true),
('SKU-1005', 'Ceramic Mug', 'Kitchen', 249.00, true),
('SKU-1006', 'Yoga Mat', 'Fitness', 899.00, true),
('SKU-1007', 'Notebook A5', 'Stationery', 149.00, true),
('SKU-1008', 'Reusable Straw Set', 'Kitchen', 199.00, true);

-- Seed inventory
INSERT INTO inventory (product_id, available_quantity, reorder_level)
SELECT product_id, 25, 5 FROM products;
```

Sample menu flow (console UX)

Main

```
=== NeoMart S.I.S.T. ===
1. Products
2. Orders
3. Reports
0. Exit
```

Products Menu

```
--- Products ---
1. Add product (ADMIN)
2. Update product (ADMIN)
3. Deactivate product (ADMIN)
4. Search product (name/category/sku)
5. View product by SKU
6. Stock adjustment (ADMIN)
0. Back
```

Orders Menu

```
--- Orders ---
1. Create order (CASHIER)
2. Add item to order (during creation)
3. Checkout + payment
4. Find order by ID
5. Find orders by customer phone
0. Back
```

Reports Menu

```
--- Reports ---
1. Daily sales summary (date)
2. Sales by product (date range)
3. Low stock report
4. Inactive products
0. Back
```

Recommended flow for “Create order”:

1. ask phone (or choose “anonymous”)
2. show product search
3. add items (sku + qty) in a loop
4. show cart preview + total
5. select payment mode + confirm
6. place order transactionally (insert order + items + reduce stock + insert payment + update order status)