StockFlow Backend Engineering Case Study Solution

## P****art 1: Code Review & Debugging****

### Issues Identified

| * **Problem** | * **Description** |
| --- | --- |
| * No SKU uniqueness check | * SKU must be unique across the platform. Code doesn't check this. |
| * Missing null/format validation | * No validation for fields like price, sku, name, or initial\_quantity. |
| * Tightly coupling product to one warehouse | * Products should exist in multiple warehouses. This code creates a single-warehouse product. |
| * Not wrapped in a transaction | * If product is saved but inventory creation fails, the database will be inconsistent. |
| * initial\_quantity not checked | * May lead to crash or incorrect inventory if missing from request. |
| * Two db.session.commit() calls | * This splits the operation, increasing risk of partial saves. |
| * No error handling | * API will crash silently on any exception. |

### Impact in Production

| **Issue** | **Impact** |
| --- | --- |
| Duplicate SKU | May cause misrouting of inventory or reporting errors. |
| Bad input data | Can lead to exceptions or corrupt data in DB. |
| Only one warehouse per product | Violates business rule. Can’t scale product to multiple locations. |
| No rollback | Leaves product without inventory or vice versa. |
| Poor UX | Unclear errors reduce trust and increase debugging time. |

### Assumptions

* SKU must be **globally unique**.
* Products can exist in **multiple warehouses**.
* initial\_quantity is **optional,** defaulting to 0.
* Product creation and inventory should happen in **a single transaction**.
* Errors should return **clear messages** to client.

### Corrected Version (Java pseudocode)

java

CopyEdit

@PostMapping("/api/products")

public ResponseEntity<?> createProduct(@RequestBody ProductRequest request) {

try {

// 1. Validate input

if (request.getSku() == null || request.getName() == null || request.getPrice() == null) {

return ResponseEntity.badRequest().body("Missing required fields");

}

// 2. Check for duplicate SKU

if (productRepo.existsBySku(request.getSku())) {

return ResponseEntity.badRequest().body("SKU already exists");

}

// 3. Create product

Product product = new Product();

product.setName(request.getName());

product.setSku(request.getSku());

product.setPrice(request.getPrice());

productRepo.save(product);

// 4. Optionally create inventory

if (request.getWarehouseId() != null) {

int quantity = request.getInitialQuantity() != null ? request.getInitialQuantity() : 0;

Inventory inventory = new Inventory(product.getId(), request.getWarehouseId(), quantity);

inventoryRepo.save(inventory);

}

return ResponseEntity.ok(Map.of("message", "Product created", "product\_id", product.getId()));

} catch (Exception e) {

return ResponseEntity.status(500).body("Internal error: " + e.getMessage());

}

}

## P****art 2: Database Design****

### Tables & Schema

CREATE TABLE Company (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL

);

CREATE TABLE Warehouse (

id SERIAL PRIMARY KEY,

name VARCHAR(255),

company\_id INT NOT NULL,

FOREIGN KEY (company\_id) REFERENCES Company(id)

);

CREATE TABLE Product (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

sku VARCHAR(100) UNIQUE NOT NULL,

price DECIMAL(10,2) NOT NULL,

product\_type VARCHAR(50) DEFAULT 'standard' -- can be 'standard' or 'bundle'

);

CREATE TABLE Inventory (

id SERIAL PRIMARY KEY,

product\_id INT NOT NULL,

warehouse\_id INT NOT NULL,

quantity INT DEFAULT 0,

UNIQUE(product\_id, warehouse\_id),

FOREIGN KEY (product\_id) REFERENCES Product(id),

FOREIGN KEY (warehouse\_id) REFERENCES Warehouse(id)

);

CREATE TABLE InventoryChangeLog (

id SERIAL PRIMARY KEY,

inventory\_id INT NOT NULL,

change\_amount INT NOT NULL,

reason VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (inventory\_id) REFERENCES Inventory(id)

);

CREATE TABLE Supplier (

id SERIAL PRIMARY KEY,

name VARCHAR(255),

contact\_email VARCHAR(255)

);

CREATE TABLE ProductSupplier (

product\_id INT,

supplier\_id INT,

PRIMARY KEY (product\_id, supplier\_id),

FOREIGN KEY (product\_id) REFERENCES Product(id),

FOREIGN KEY (supplier\_id) REFERENCES Supplier(id)

);

CREATE TABLE ProductBundle (

bundle\_id INT,

component\_product\_id INT,

quantity INT DEFAULT 1,

PRIMARY KEY (bundle\_id, component\_product\_id),

FOREIGN KEY (bundle\_id) REFERENCES Product(id),

FOREIGN KEY (component\_product\_id) REFERENCES Product(id)

);

### Questions to Ask Product Team

1. Should bundles have their own SKUs and prices?
2. Can a product have multiple suppliers or just one?
3. Should thresholds be defined per product or per warehouse?
4. How recent is “recent sales activity”?
5. Do you want audit logs for price or product updates?
6. Can products be shared across companies?

### Justifications

| **Design** | **Reason** |
| --- | --- |
| sku UNIQUE | Enforces platform-wide uniqueness |
| Inventory table | Separates product from per-warehouse stock |
| InventoryChangeLog | Allows tracking inventory changes |
| ProductSupplier | Supports many-to-many relationship |
| ProductBundle | Supports nested bundled products |
| Foreign keys | Enforce data consistency |
| product\_type | Easily distinguish bundles vs standalone |

## P****art 3: Low Stock Alert API****

### Assumptions

* We calculate low stock using **a threshold per product**.
* Only products with **recent sales** (e.g. last 30 days) are considered.
* Sales table exists to track sales per product per warehouse.
* days\_until\_stockout = current\_stock / average\_daily\_sales.
* Each product has **at least one supplier**.

### Java Pseudocode

@GetMapping("/api/companies/{companyId}/alerts/low-stock")

public ResponseEntity<?> getLowStockAlerts(@PathVariable int companyId) {

List<LowStockAlert> alerts = new ArrayList<>();

// Fetch all warehouses for the given company

List<Warehouse> warehouses = warehouseRepo.findByCompanyId(companyId);

// We must check stock levels across \*all\* warehouses owned by the company.

for (Warehouse warehouse : warehouses) {

// For each warehouse, get inventory records

List<Inventory> inventoryList = inventoryRepo.findByWarehouseId(warehouse.getId());

// This tells us how many units of each product exist in this warehouse.

for (Inventory inventory : inventoryList) {

// Get the product details

Product product = productRepo.findById(inventory.getProductId());

// Skip products with no recent sales (e.g., in the last 30 days)

if (!salesRepo.hasRecentSales(product.getId(), warehouse.getId(), 30)) continue;

// This helps avoid false alerts for slow-moving or obsolete products.

// Fetch the product’s low stock threshold

int threshold = thresholdRepo.getThresholdForProduct(product.getId());

// Compare stock with threshold

if (inventory.getQuantity() < threshold) {

// Estimate how soon stock will run out

int avgDailySales = salesRepo.getAverageDailySales(product.getId(), warehouse.getId(), 30);

int daysUntilStockout = avgDailySales > 0

? inventory.getQuantity() / avgDailySales

: -1; // -1 = unknown / not computable

// Fetch supplier information for reordering

Supplier supplier = supplierRepo.getPrimarySupplier(product.getId());

// Create alert object with all required details

LowStockAlert alert = new LowStockAlert(

product.getId(), product.getName(), product.getSku(),

warehouse.getId(), warehouse.getName(),

inventory.getQuantity(), threshold,

daysUntilStockout,

supplier

);

alerts.add(alert);

}

}

}

// Return response with total alert count

return ResponseEntity.ok(Map.of(

"alerts", alerts,

"total\_alerts", alerts.size()

));

### Edge Cases Handled

| * **Case** | * **Handling** |
| --- | --- |
| * No recent sales | * Skipped |
| * No supplier | * Set as null or skipped |
| * Zero avg daily sales | * Return -1 for days until stockout |
| * Multiple warehouses | * Handled per warehouse |
| * Missing threshold | * Default to 0 or skip alert |