# **StockFlow B2B Inventory Management Platform**

## **Candidate: Pragati Tabe**

## **Submission Time: 29-7-2025**

## **Part 1: Code Review & Debugging**

### 1. Identified Issues:

#### a. SKU Uniqueness Not Enforced

* **Problem**: sku is assumed to be unique but no validation exists.
* **Impact**: Duplicate SKUs can lead to conflicts across warehouses.

#### b. Missing Error Handling

* **Problem**: No error handling if required fields are missing or invalid.
* **Impact**: Application may crash or allow incomplete data.

#### c. Product Creation Should Be Atomic

* **Problem**: Two separate db.session.commit() calls.
* **Impact**: If the second call fails, database will be in inconsistent state (product created, inventory not).

#### d. initial\_quantity Field Is Not Validated

* **Problem**: Assumes it always exists in the request.
* **Impact**: KeyError if missing; could break production code.

#### e. Business Rule Violation: Product Can Exist in Multiple Warehouses

* **Problem**: Product model seems to assume a single warehouse per product.
* **Impact**: Violates business logic that products can exist in *multiple* warehouses.

### 2. Corrected Version using Spring Boot (Java)

@RestController  
@RequestMapping("/api/products")  
public class ProductController {  
  
 @Autowired  
 private ProductRepository productRepository;  
  
 @Autowired  
 private InventoryRepository inventoryRepository;  
  
 @PostMapping  
 public ResponseEntity<?> createProduct(@RequestBody ProductRequest request) {  
 // Validate required fields  
 if (request.getName() == null || request.getSku() == null || request.getPrice() == null ||  
 request.getWarehouseId() == null || request.getInitialQuantity() == null) {  
 return ResponseEntity.badRequest().body("Missing required fields");  
 }  
  
 // Check for existing SKU  
 if (productRepository.existsBySku(request.getSku())) {  
 return ResponseEntity.status(HttpStatus.CONFLICT).body("SKU already exists");  
 }  
  
 try {  
 // Save product  
 Product product = new Product();  
 product.setName(request.getName());  
 product.setSku(request.getSku());  
 product.setPrice(request.getPrice());  
 product = productRepository.save(product);  
  
 // Save inventory  
 Inventory inventory = new Inventory();  
 inventory.setProduct(product);  
 inventory.setWarehouseId(request.getWarehouseId());  
 inventory.setQuantity(request.getInitialQuantity());  
 inventoryRepository.save(inventory);  
  
 return ResponseEntity.status(HttpStatus.CREATED)  
 .body(Map.of("message", "Product created", "product\_id", product.getId()));  
  
 } catch (Exception e) {  
 return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR)  
 .body("An error occurred: " + e.getMessage());  
 }  
 }  
}

**ProductRequest.java (DTO)**

public class ProductRequest {  
 private String name;  
 private String sku;  
 private BigDecimal price;  
 private Long warehouseId;  
 private Integer initialQuantity;  
 // Getters and setters  
}

## **Part 2: Database Design**

### Schema Design (PostgreSQL-Style SQL DDL)

-- Companies Table  
CREATE TABLE companies (  
 id SERIAL PRIMARY KEY,  
 name VARCHAR(255) NOT NULL UNIQUE  
);  
  
-- Warehouses Table  
CREATE TABLE warehouses (  
 id SERIAL PRIMARY KEY,  
 company\_id INTEGER REFERENCES companies(id),  
 name VARCHAR(255) NOT NULL  
);  
  
-- Products Table  
CREATE TABLE products (  
 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 'single', -- or 'bundle'  
 threshold INTEGER DEFAULT 10  
);  
  
-- Inventory Table  
CREATE TABLE inventory (  
 id SERIAL PRIMARY KEY,  
 product\_id INTEGER REFERENCES products(id),  
 warehouse\_id INTEGER REFERENCES warehouses(id),  
 quantity INTEGER NOT NULL,  
 UNIQUE(product\_id, warehouse\_id)  
);  
  
-- Inventory History Table  
CREATE TABLE inventory\_history (  
 id SERIAL PRIMARY KEY,  
 product\_id INTEGER REFERENCES products(id),  
 warehouse\_id INTEGER REFERENCES warehouses(id),  
 change\_quantity INTEGER NOT NULL,  
 change\_type VARCHAR(50), -- e.g., 'sale', 'restock'  
 change\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Suppliers Table  
CREATE TABLE suppliers (  
 id SERIAL PRIMARY KEY,  
 name VARCHAR(255),  
 contact\_email VARCHAR(255)  
);  
  
-- Product-Supplier Mapping  
CREATE TABLE product\_suppliers (  
 product\_id INTEGER REFERENCES products(id),  
 supplier\_id INTEGER REFERENCES suppliers(id),  
 PRIMARY KEY (product\_id, supplier\_id)  
);  
  
-- Product Bundles Table  
CREATE TABLE product\_bundles (  
 bundle\_id INTEGER REFERENCES products(id),  
 component\_id INTEGER REFERENCES products(id),  
 quantity INTEGER NOT NULL,  
 PRIMARY KEY (bundle\_id, component\_id)  
);

Questions for Product Team

1. Can a supplier provide only some variants of a product (e.g., sizes)?
2. Do bundles have their own SKUs?
3. Is stock automatically adjusted when a bundle is sold?
4. Should we track expiration dates for inventory?
5. Do thresholds vary per product *per warehouse*?
6. Should we log user activity for inventory updates?

### Design Decisions

* **Composite keys**: Ensure unique product-warehouse entries in inventory.
* **Indexes**: On sku, product\_id, and warehouse\_id for faster lookups.
* **History table**: Enables tracking and analytics of inventory trends.
* **Bundles**: Represented via a mapping table for flexible nesting.

## 

## **Part 3: API Implementation (Low Stock Alert)**

### Assumptions

* Each product has a threshold field.
* Recent activity = sales in the last 30 days.
* Sales table exists with product\_id, warehouse\_id, sale\_date, quantity.
* Supplier mapping exists via product\_suppliers.

### Java Spring Boot Endpoint

@RestController  
@RequestMapping("/api/companies")  
public class AlertController {  
  
 @Autowired  
 private InventoryRepository inventoryRepository;  
  
 @Autowired  
 private SaleRepository saleRepository;  
  
 @Autowired  
 private SupplierRepository supplierRepository;  
  
 @GetMapping("/{companyId}/alerts/low-stock")  
 public ResponseEntity<?> getLowStockAlerts(@PathVariable Long companyId) {  
 LocalDateTime cutoff = LocalDateTime.now().minusDays(30);  
 List<Warehouse> warehouses = warehouseRepository.findByCompanyId(companyId);  
 List<Long> warehouseIds = warehouses.stream().map(Warehouse::getId).toList();  
  
 List<Inventory> lowStockInventories = inventoryRepository.findLowStockByWarehouses(warehouseIds);  
  
 List<Map<String, Object>> alerts = new ArrayList<>();  
  
 for (Inventory inventory : lowStockInventories) {  
 Product product = inventory.getProduct();  
 List<Sale> recentSales = saleRepository.findByProductAndWarehouseAfterDate(  
 product.getId(), inventory.getWarehouseId(), cutoff  
 );  
  
 if (recentSales.isEmpty()) continue;  
  
 int totalSold = recentSales.stream().mapToInt(Sale::getQuantity).sum();  
 double dailyAvg = totalSold / 30.0;  
 int daysUntilStockout = dailyAvg == 0 ? 0 : (int) (inventory.getQuantity() / dailyAvg);  
  
 Supplier supplier = supplierRepository.findFirstByProductId(product.getId());  
  
 Map<String, Object> alert = Map.of(  
 "product\_id", product.getId(),  
 "product\_name", product.getName(),  
 "sku", product.getSku(),  
 "warehouse\_id", inventory.getWarehouseId(),  
 "warehouse\_name", inventory.getWarehouse().getName(),  
 "current\_stock", inventory.getQuantity(),  
 "threshold", product.getThreshold(),  
 "days\_until\_stockout", daysUntilStockout,  
 "supplier", Map.of(  
 "id", supplier != null ? supplier.getId() : null,  
 "name", supplier != null ? supplier.getName() : "N/A",  
 "contact\_email", supplier != null ? supplier.getContactEmail() : "N/A"  
 )  
 );  
 alerts.add(alert);  
 }  
  
 return ResponseEntity.ok(Map.of("alerts", alerts, "total\_alerts", alerts.size()));  
 }  
}

## 

## **Final Notes**

### Assumptions Made

* Sales table exists and is required for activity tracking.
* Thresholds are stored per product (can be extended to warehouse level).
* One supplier per product for simplicity.

### Thought Process

* Prioritized data integrity and isolation in API.
* Used schema normalization to maintain scalability.
* Focused on real-world usability for alerting system.