Part 1: Code Review & Debugging

Task: Review and fix the product creation API.

Assumptions / Missing Requirements:

- Products can exist in multiple warehouses.
- SKU must be unique.
- Price can be decimal.
- `warehouse_id` and `initial_quantity` may be optional.

Edge Cases Considered:

- Missing fields (`name`, `sku`, `price`).
- Duplicate SKUs.
- Invalid price values.
- Optional fields missing.
- Inventory creation fails after product saved → handled via `@Transactional`.

Java Spring Boot Implementation:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private InventoryRepository inventoryRepository;

    @Autowired
```

```
    private WarehouseRepository warehouseRepository;

    // Transactional ensures product + inventory creation is atomic
    @Transactional
    @PostMapping
    public ResponseEntity<?> createProduct(@RequestBody Map<String,
Object> data) {
        try {
            // Validate required fields (name, sku, price)
            // Issue in original code: No validation
            // Impact: Missing field causes runtime exception, API
crashes
            // Fix: Check required fields before proceeding
            if (!data.containsKey("name") || !data.containsKey("sku")
|| !data.containsKey("price")) {
                return ResponseEntity.badRequest().body("Missing
required fields: name, sku, or price");
            }

            String name = data.get("name").toString();
            String sku = data.get("sku").toString();

            // Ensure SKU uniqueness across platform
            // Issue in original code: SKU duplicates allowed
            // Impact: Violates business rules, causes confusion in
inventory tracking
            // Fix: Check if SKU exists, return conflict if duplicate
            if (productRepository.existsBySku(sku)) {
                return
ResponseEntity.status(HttpStatus.CONFLICT).body("SKU already exists");
            }

            // Parse price as BigDecimal
            // Issue in original code: Price type not validated
            // Impact: Invalid decimal can crash DB or financial
calculations
            // Fix: Try-catch parsing, return bad request if invalid
            BigDecimal price;
```

```
            try {
                price = new BigDecimal(data.get("price").toString());
            } catch (NumberFormatException e) {
                return ResponseEntity.badRequest().body("Invalid price
format");
            }

            // Create and save product entity
            // Issue in original code: No transaction handling;
product and inventory saved separately
            // Impact: If inventory creation fails, product already
exists causing data inconsistency
            // Fix: Save product within @Transactional method so
rollback occurs on failure
            Product product = new Product();
            product.setName(name);
            product.setSku(sku);
            product.setPrice(price);
            productRepository.save(product);

            // Handle warehouse and inventory if provided
            // Issue in original code: Assumes product exists in only
one warehouse
            // Impact: Cannot track products across multiple
warehouses
            // Fix: Inventory table links Product → Warehouse →
Quantity
            if (data.containsKey("warehouse_id")) {
                Long warehouseId =
Long.parseLong(data.get("warehouse_id").toString());

                // Check warehouse exists
                Warehouse warehouse =
warehouseRepository.findById(warehouseId)
                        .orElseThrow(() -> new
RuntimeException("Warehouse not found"));

                // Handle optional initial_quantity
```

```
                // Issue in original code: Assumes initial_quantity
exists
                // Impact: Missing quantity crashes API
                // Fix: Default to 0 if missing
                Integer initialQuantity = 0;
                if (data.containsKey("initial_quantity")) {
                    initialQuantity =
Integer.parseInt(data.get("initial_quantity").toString());
                }

                // Create inventory record for this warehouse
                // Additional context: Products can exist in multiple
warehouses
                // Fix: Each warehouse gets its own inventory record
for the same product
                Inventory inventory = new Inventory();
                inventory.setProduct(product);
                inventory.setWarehouse(warehouse);
                inventory.setQuantity(initialQuantity);
                inventoryRepository.save(inventory);
            }

            // Return a success response with the product ID
            Map<String, Object> response = new HashMap<>();
            response.put("message", "Product created successfully");
            response.put("product_id", product.getId());
            return ResponseEntity.ok(response);

        } catch (Exception e) {
            // Handle unexpected exceptions
            // Issue in original code: No exception handling
            // Impact: Crashes API, exposes stack traces
            // Fix: Wrap in try-catch, return Internal Server Error if
something unexpected happens
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .body("Error creating product: " +
e.getMessage());
```

```
        }
    }
}
```

Part 2: Database Design

Requirements Covered:

- Companies → multiple warehouses.
- Products → multiple warehouses with quantities.
- Inventory change tracking.
- Supplier-product mapping.
- Bundle products.

SQL DDL (PostgreSQL)

```
-- Companies
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Warehouses per company
CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    company_id INT NOT NULL REFERENCES companies(id) ON DELETE
CASCADE,
    name VARCHAR(255) NOT NULL,
    location TEXT,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);
```

```
-- Suppliers
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    contact_info TEXT,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Products
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) NOT NULL UNIQUE,
    price NUMERIC(12,2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE,
    low_stock_threshold INT DEFAULT 10,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Bundles
CREATE TABLE product_bundles (
    bundle_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    component_id INT NOT NULL REFERENCES products(id) ON DELETE
CASCADE,
    quantity INT NOT NULL DEFAULT 1,
    PRIMARY KEY (bundle_id, component_id)
);

-- Inventory
CREATE TABLE inventory (
    id SERIAL PRIMARY KEY,
    product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    warehouse_id INT NOT NULL REFERENCES warehouses(id) ON DELETE
CASCADE,
    quantity INT NOT NULL DEFAULT 0,
```

```
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),
    UNIQUE(product_id, warehouse_id)
);

-- Inventory History
CREATE TABLE inventory_history (
    id SERIAL PRIMARY KEY,
    inventory_id INT NOT NULL REFERENCES inventory(id) ON DELETE
CASCADE,
    change INT NOT NULL,
    reason VARCHAR(255),
    created_at TIMESTAMP DEFAULT NOW()
);

-- Supplier-product mapping
CREATE TABLE supplier_products (
    supplier_id INT NOT NULL REFERENCES suppliers(id) ON DELETE
CASCADE,
    product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    PRIMARY KEY (supplier_id, product_id)
);
```

Decisions & Justifications:

- **UNIQUE constraints** on SKU and `(product_id, warehouse_id)` to avoid duplicates.
- **Foreign keys + cascade** to maintain referential integrity.
- **`inventory_history`** tracks changes for auditing.
- **Bundles table** supports many-to-many mapping of products.
- **NUMERIC(12,2)** ensures accurate price storage.

Gaps / Questions:

- Supplier-to-warehouse mapping?
- Bundle pricing (auto vs manual)?
- Nested bundles allowed?
- User tracking for inventory changes?

Part 3: API Implementation – Low Stock Alerts

Endpoint: `GET /api/companies/{company_id}/alerts/low-stock`

Assumptions:

- Recent sales = last 30 days.
- `product.lowStockThreshold` defines per-product threshold.
- `days_until_stockout = current_stock / avg_daily_sales`.
- Supplier info linked via `supplier_products`.

Java Spring Boot Implementation:

```
@RestController
@RequestMapping("/api/companies")
public class AlertController {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private InventoryRepository inventoryRepository;

    @Autowired
    private WarehouseRepository warehouseRepository;

    @Autowired
```

```
    private SupplierRepository supplierRepository;

    @Autowired
    private SalesRepository salesRepository; // Tracks recent sales

    @GetMapping("/{companyId}/alerts/low-stock")
    public ResponseEntity<?>
getLowStockAlerts(@PathVariable("companyId") Long companyId) {

        try {
            List<Map<String, Object>> alerts = new ArrayList<>();

            // Fetch warehouses for the company
            List<Warehouse> warehouses =
warehouseRepository.findByCompanyId(companyId);

            for (Warehouse warehouse : warehouses) {
                List<Inventory> inventories =
inventoryRepository.findByWarehouseId(warehouse.getId());

                for (Inventory inventory : inventories) {
                    Product product = inventory.getProduct();

                    // Skip products with no recent sales (last 30
days)
                    int recentSalesCount =
salesRepository.countRecentSales(product.getId(), 30);
                    if (recentSalesCount == 0) continue;

                    int currentStock = inventory.getQuantity();
                    int threshold = product.getLowStockThreshold();

                    // Only alert if stock <= threshold
                    if (currentStock > threshold) continue;

                    // Calculate days until stockout
                    double averageDailySales = recentSalesCount /
30.0;
```

```
                    int daysUntilStockout = (averageDailySales > 0) ?
(int) Math.ceil(currentStock / averageDailySales) : -1;

                    // Get supplier info
                    Supplier supplier =
supplierRepository.findFirstByProductId(product.getId());

                    Map<String, Object> alert = new HashMap<>();
                    alert.put("product_id", product.getId());
                    alert.put("product_name", product.getName());
                    alert.put("sku", product.getSku());
                    alert.put("warehouse_id", warehouse.getId());
                    alert.put("warehouse_name", warehouse.getName());
                    alert.put("current_stock", currentStock);
                    alert.put("threshold", threshold);
                    alert.put("days_until_stockout",
daysUntilStockout);

                    if (supplier != null) {
                        Map<String, Object> supplierInfo = new
HashMap<>();
                        supplierInfo.put("id", supplier.getId());
                        supplierInfo.put("name", supplier.getName());
                        supplierInfo.put("contact_email",
supplier.getContactEmail());
                        alert.put("supplier", supplierInfo);
                    }

                    alerts.add(alert);
                }
            }

            Map<String, Object> response = new HashMap<>();
            response.put("alerts", alerts);
            response.put("total_alerts", alerts.size());

            return ResponseEntity.ok(response);
```

```
        } catch (Exception e) {
            // Handle unexpected exceptions
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .body("Error fetching low stock alerts: " +
e.getMessage());
        }
    }
}
```

Edge Case Handling:

- Company has no warehouses → returns empty alerts.
- Warehouse has no inventory → skipped.
- Product has no recent sales → skipped.
- Supplier missing → `supplier` field null.
- Avoid division by zero in `days_until_stockout`.

Assumptions Made:

- Recent sales = last 30 days.
- Low stock threshold per product.
- Each inventory record is per warehouse.
- Bundles not nested.
- Supplier per product, not per warehouse.

Summary of Decisions

Part	Decisions & Reasoning

Code Review	Added validation, transactional handling, optional field handling, SKU uniqueness, exception handling.
Database Design	Designed normalized schema, enforced constraints, added audit/history tables, supported bundles, multi-warehouse stock, suppliers.
API Implementation	Checked low-stock per warehouse, recent sales only, included supplier info, handled edge cases, calculated days until stockout.