Bynry Backend Engineer Intern Case Study

Submitted by: VEDANT DHOLE
Technology: Java & Spring Boot
This document provides a complete solution to the case study, with explanations and code.

Part 1: Code Review & Debugging

The original Python code had several problems. Let's look at what they were, why they are bad for a live application, and how we can fix them using Java and Spring Boot.

1. Identified Issues & Production Impact (Explained in Simple Terms)

Issue	Simple Explanation	Impact in a Live App (and how Java/Spring helps)
Two Separate Database Saves	The code saves the new product, and then separately saves its inventory. [cite_start]It's like two separate trips to the store. [cite: 13-30]	**Problem:** If the second trip fails (e.g., the app crashes), you have a product in your system with no stock record. This is broken data. **In Java/Spring:** We use the @Transactional annotation. This tells Spring: "Do all the database work in this method as one single action. If anything fails, undo everything you've done so far." This keeps our data consistent.
No Input Validation	[cite_start]The code trusts that the user will send perfect data. [cite: 13] It doesn't check if fields are missing or have the wrong format (like text where a number should be).	**Problem:** Bad data can crash the server or save garbage to the database. **In Java/Spring:** We use Validation annotations like @NotBlank and @NotNull on a special request object (DTO). Spring automatically checks the incoming data and rejects it with a clear error message if it's bad.
No SKU Uniqueness Check	The code doesn't check if a product with the same SKU	**Problem:** You could end up with two different products

	(unique code) already exists before trying to save a new one. [cite_start]The requirements state that SKUs must be unique. [cite: 36]	having the same unique code, which would cause chaos in inventory management. **In Java/Spring:** Before saving, we make a quick database call like productRepository.existsBySku(...). If it returns true, we stop and tell the user the SKU is already taken.
No Real Error Handling	If something unexpected goes wrong, the code just crashes.	**Problem:** The user sees a generic, unhelpful error message, and we might not know what exactly went wrong. **In Java/Spring:** We use try-catch blocks and Spring's @ControllerAdvice to catch specific errors. This lets us send clear, helpful error messages to the user (like "SKU already exists" or "Database is down") while logging the detailed technical error for developers.
Wrong Warehouse Logic	[cite_start]The code saves the warehouse_id directly on the product. [cite: 20] [cite_start]But the rules say a product can be in many warehouses. [cite: 36, 42]	**Problem:** The database design is fundamentally wrong and cannot support the business requirements. **In Java/Spring:** We fix this in the database design itself (see Part 2). A Product doesn't know about warehouses. Instead, an Inventory table connects a Product to a Warehouse and stores the quantity.
Poor Success Message	[cite_start]The code returns a very basic message like "message": "Product created". [cite: 30]	**Problem:** The user's application (the "client") doesn't get the ID or other details of the new product it just created. It would have to make another request to get it. **In Java/Spring:**

		We return a structured JSON object. It's best practice to include a clear message, a status, and the data of the newly created object. We also return the correct HTTP status code, 201 Created.

2. Corrected Version using Java & Spring Boot

Here is the complete, corrected code that fixes all the issues mentioned above.

ProductService.java (The Logic)
This class handles the business logic for creating a product.

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.server.ResponseStatusException;
import org.springframework.http.HttpStatus;

// ... other necessary imports

@Service
public class ProductService {

    // Dependencies are injected by Spring
    private final ProductRepository productRepository;
    private final InventoryRepository inventoryRepository;
    private final WarehouseRepository warehouseRepository;

    public ProductService(ProductRepository productRepository, /*...other repos...*/ ) {
        this.productRepository = productRepository;
        // ... initialize other repos
    }

    /**
     * This whole method runs as a single, safe database transaction.
     * If any part fails, all changes are rolled back automatically.
     */
    @Transactional
    public ProductResponse createProduct(ProductCreateRequest request) {
```

```
        // Issue #3 Fix: Check for SKU Uniqueness first
        if (productRepository.existsBySku(request.getSku())) {
            throw new ResponseStatusException(HttpStatus.CONFLICT, "Product with SKU
'" + request.getSku() + "' already exists.");
        }

        // Issue #2 Fix: Validate that the warehouse exists
        Warehouse warehouse =
warehouseRepository.findById(request.getWarehouseId())
                .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND,
"Warehouse not found."));

        // Create the product object
        Product newProduct = new Product();
        newProduct.setName(request.getName());
        newProduct.setSku(request.getSku());
        newProduct.setPrice(request.getPrice());

        // Issue #1 Fix: Save the product and inventory in one transaction
        Product savedProduct = productRepository.save(newProduct);

        // Create the inventory record linking the product and warehouse
        Inventory newInventory = new Inventory();
        newInventory.setProduct(savedProduct);
        newInventory.setWarehouse(warehouse);
        newInventory.setQuantity(request.getInitialQuantity());
        inventoryRepository.save(newInventory);

        // Issue #6 Fix: Return a structured, useful response
        return new ProductResponse(savedProduct.getId(), savedProduct.getName(),
savedProduct.getSku(), savedProduct.getPrice());
    }
}
```

ProductController.java (The API Endpoint)
This class defines the API endpoint that users will call.

```
import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.http.HttpStatus;
import jakarta.validation.Valid;
import java.util.Map;

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

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @PostMapping("/products")
    public ResponseEntity<Map<String, Object>> createProduct(@Valid @RequestBody
ProductCreateRequest request) {
        // @Valid triggers the input validation
        ProductResponse createdProduct = productService.createProduct(request);

        // Create a nice, structured response map
        Map<String, Object> response = Map.of(
            "message", "Product created successfully",
            "product", createdProduct
        );

        // Return with 201 Created status
        return new ResponseEntity<>(response, HttpStatus.CREATED);
    }
}
```

Part 2: Database Design

1. Proposed Database Schema (SQL DDL)

This is the blueprint for our database tables. [cite_start]It's designed to correctly store the data and the relationships between them based on the requirements. [cite: 41, 42, 43, 44, 45]

```
-- Stores company information
CREATE TABLE companies (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Stores warehouse information, linked to a company
CREATE TABLE warehouses (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    location TEXT,
    FOREIGN KEY (company_id) REFERENCES companies(id)
);

-- Stores supplier information
CREATE TABLE suppliers (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    contact_email VARCHAR(255) UNIQUE
);

-- Stores the core product details. Note: No warehouse_id here!
CREATE TABLE products (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    sku VARCHAR(100) NOT NULL UNIQUE, -- The SKU must be unique
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    low_stock_threshold INT DEFAULT 20,
    primary_supplier_id BIGINT,
    FOREIGN KEY (primary_supplier_id) REFERENCES suppliers(id)
);

-- This table connects Products and Warehouses and stores the quantity.
-- This is the key to letting a product exist in multiple warehouses.
CREATE TABLE inventory (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
```

```
    warehouse_id BIGINT NOT NULL,
    quantity INT NOT NULL DEFAULT 0,
    UNIQUE (product_id, warehouse_id), -- A product can only appear once per
warehouse
    FOREIGN KEY (product_id) REFERENCES products(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
);

-- This table tracks sales to know which products are "active"
CREATE TABLE sales (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    warehouse_id BIGINT NOT NULL,
    quantity_sold INT NOT NULL,
    sale_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

[cite_start]2. Gaps & Questions for the Product Team [cite: 47]

Before finalizing the design, I would ask the product team these questions to avoid future problems:

- [cite_start]**Product Bundles:** How do we track stock for a "bundle" (e.g., a "Gift Basket" containing other products)? [cite: 45] Is its stock based on the component with the lowest quantity?
- [cite_start]**Suppliers:** Can one product be supplied by many different suppliers? [cite: 44] If so, do we need to track a "main" or "preferred" supplier?
- [cite_start]**Low Stock Definition:** The alert threshold "varies by product type." [cite: 53] Is this a general rule (e.g., "all food items have a threshold of 50"), or can we set a specific threshold for every single product? (My design allows for a per-product threshold because it's more flexible).
- [cite_start]**"Recent" Sales:** For the low-stock alert, what does "recent" mean? [cite: 54] The last 7 days? 30 days? This needs to be a clear business rule.
- **User Permissions:** Who is allowed to see what data? Can one company see another company's inventory? (This is important for security).

Part 3: API Implementation Approach

[cite_start]This section explains the plan and the code for building the low-stock alert API endpoint. [cite: 52]

1. High-Level Logic & Pseudocode

This is the step-by-step plan for how the code will work.

FUNCTION getLowStockAlerts(for a specific company_id):

1. [cite_start]Find all warehouses that belong to this company. [cite: 54]
   If there are no warehouses, stop and return an empty list.

2. [cite_start]Find all products that have been sold "recently" from those warehouses. [cite: 54]
   - We'll define "recently" as the last 30 days.
   - We look at the `sales` table to get a list of these active products.

3. Get the current inventory details for these active products in those warehouses.

4. [cite_start]From that list, filter it down to only the items where the `quantity` is less than or equal to the `low_stock_threshold`. [cite: 53]

5. [cite_start]For each low-stock item, build a nicely formatted alert. [cite: 57]
   - [cite_start]Get the full product name and SKU. [cite: 63, 64]
   - [cite_start]Get the warehouse name. [cite: 66]
   - [cite_start]Get the supplier's name and email. [cite: 70]
   - [cite_start](Bonus) Try to calculate how many days are left until it's out of stock. [cite: 69]
   - Add this complete alert object to our final list.

6. [cite_start]Return the final list of alerts and the total count. [cite: 59, 77]

2. Java & Spring Boot Implementation

Here is the actual Java code that follows the plan above.

AlertService.java (The Logic)

```
import org.springframework.stereotype.Service;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Collections;
import java.util.List;
```

```
import java.util.stream.Collectors;

@Service
public class AlertService {

    private static final int RECENT_SALES_DAYS = 30;

    private final WarehouseRepository warehouseRepository;
    private final SalesRepository salesRepository;
    private final InventoryRepository inventoryRepository;

    public AlertService(/*...repositories are injected...*/) {
        this.warehouseRepository = warehouseRepository;
        this.salesRepository = salesRepository;
        this.inventoryRepository = inventoryRepository;
    }

    public LowStockAlertResponse getLowStockAlerts(Long companyId) {
        // Step 1: Find all warehouses for the company.
        List<Warehouse> warehouses =
warehouseRepository.findByCompanyId(companyId);
        if (warehouses.isEmpty()) {
            return new LowStockAlertResponse(Collections.emptyList(), 0);
        }
        List<Long> warehouseIds =
warehouses.stream().map(Warehouse::getId).collect(Collectors.toList());

        // Step 2: Find products with recent sales.
        Instant thirtyDaysAgo = Instant.now().minus(RECENT_SALES_DAYS,
ChronoUnit.DAYS);
        List<Long> productIdsWithRecentSales =
salesRepository.findDistinctProductIdsWithRecentSales(warehouseIds,
thirtyDaysAgo);
        if (productIdsWithRecentSales.isEmpty()) {
            return new LowStockAlertResponse(Collections.emptyList(), 0);
        }

        // Step 3 & 4: Get relevant inventory and filter for low stock.
        List<LowStockAlertResponse.Alert> alerts = inventoryRepository.findAll().stream()
```

```
                .filter(inv -> warehouseIds.contains(inv.getWarehouse().getId()))
                .filter(inv -> productIdsWithRecentSales.contains(inv.getProduct().getId()))
                .filter(inv -> inv.getQuantity() <= inv.getProduct().getLowStockThreshold())
                // Step 5: Build the alert response for each item.
                .map(this::createAlertFromInventory)
                .collect(Collectors.toList());

        return new LowStockAlertResponse(alerts, alerts.size());
    }

    // Helper method to keep the main logic clean
    private LowStockAlertResponse.Alert createAlertFromInventory(Inventory inv) {
        Product product = inv.getProduct();
        Warehouse warehouse = inv.getWarehouse();

        // Safely build supplier info
        LowStockAlertResponse.SupplierInfo supplierInfo = null;
        if (product.getPrimarySupplier() != null) {
            Supplier s = product.getPrimarySupplier();
            supplierInfo = new LowStockAlertResponse.SupplierInfo(s.getId(), s.getName(),
s.getContactEmail());
        }

        return new LowStockAlertResponse.Alert(
                product.getId(),
                product.getName(),
                product.getSku(),
                warehouse.getId(),
                warehouse.getName(),
                inv.getQuantity(),
                product.getLowStockThreshold(),
                null, // daysUntilStockout calculation is complex, omitted for simplicity here
                supplierInfo
        );
    }
}
```

AlertController.java (The API Endpoint)

```
import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;
import org.springframework.web.server.ResponseStatusException;
import org.springframework.http.HttpStatus;

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

    private final AlertService alertService;
    private final CompanyRepository companyRepository;

    public AlertController(AlertService alertService, CompanyRepository
companyRepository) {
        this.alertService = alertService;
        this.companyRepository = companyRepository;
    }

    @GetMapping("/companies/{companyId}/alerts/low-stock")
    public ResponseEntity<LowStockAlertResponse> getLowStockAlerts(@PathVariable
Long companyId) {
        // First, check if the company even exists.
        if (!companyRepository.existsById(companyId)) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Company not
found with ID: " + companyId);
        }

        // Call the service to do the hard work
        LowStockAlertResponse response = alertService.getLowStockAlerts(companyId);

        return ResponseEntity.ok(response);
    }
}
```