

Backend Engineering Intern Case Study Submission

Name: Rutik Maruti Dhole

Part 1: Code Review & Debugging

Here's my analysis of the initial code snippet, focusing on potential issues and how to resolve them.

Issues and Fixes

- **Input Validation:**
 - **Problem:** The original code blindly trusts incoming data. Missing or improperly formatted fields like price or initial_quantity could cause the application to crash.
 - **Fix:** I've added checks to ensure all required fields are present before processing. This makes the endpoint more robust and prevents unexpected runtime errors.
- **SKU Uniqueness:**
 - **Problem:** The code doesn't enforce the business rule that SKUs must be unique. This could lead to duplicate product entries, causing confusion in inventory management and order fulfillment.
 - **Fix:** The corrected code now queries the database to verify the SKU doesn't already exist before attempting to create a new product.
- **Error Handling:**
 - **Problem:** A failure during the database commit would result in an unhandled exception, crashing the server and providing a poor user experience.
 - **Fix:** I've wrapped the logic in a try...except block. If any part of the process fails, the database transaction is rolled back to prevent partial data insertion, and a clear error message is returned to the client.
- **Warehouse Dependency:**
 - **Problem:** Storing warehouse_id directly in the Product model conflicts with the business logic that a product can exist in multiple warehouses.
 - **Fix:** The warehouse_id has been removed from the product model. Instead, the relationship is correctly managed in the Inventory table, which links a product to a warehouse with a specific quantity.

- **Transactional Integrity:**

- **Problem:** The code uses two separate database commits. If the first one (creating the product) succeeds but the second one (updating inventory) fails, we're left with an orphaned product record without any stock information.
- **Fix:** I've unified both operations under a single `db.session.commit()`. This ensures that either both the product and its initial inventory are created successfully, or neither is.

Corrected Python Code

Python

```
@app.route('/api/products', methods=['POST'])
```

```
def create_product():
```

```
    data = request.get_json()
```

```
    try:
```

```
        # Validate that all necessary fields are present in the request
```

```
        required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
```

```
        for field in required_fields:
```

```
            if field not in data:
```

```
                return {"error": f"Missing field: {field}"}, 400
```

```
        # Check if a product with the same SKU already exists to ensure uniqueness
```

```
        if Product.query.filter_by(sku=data['sku']).first():
```

```
            return {"error": "SKU already exists"}, 409
```

```
        # Create the product instance
```

```
        product = Product(
```

```
            name=data['name'],
```

```
            sku=data['sku'],
```

```
            price=float(data['price'])
```

```

)

db.session.add(product)

db.session.flush() # Use flush to assign an ID to the product before committing


# Create the initial inventory record for the product in its warehouse
inventory = Inventory(
    product_id=product.id,
    warehouse_id=data['warehouse_id'],
    quantity=int(data['initial_quantity'])
)

db.session.add(inventory)


# Commit both operations as a single atomic transaction
db.session.commit()

return {"message": "Product created", "product_id": product.id}, 201


except Exception as e:

    # If any error occurs, roll back the transaction to maintain data integrity
    db.session.rollback()

    return {"error": str(e)}, 500

```

Part 2: Database Design

Here is my proposed database schema designed to be flexible and scalable for the "StockFlow" platform.

-- Table to store basic information about companies using the platform

CREATE TABLE companies (

id SERIAL PRIMARY KEY, -- Every company gets a unique ID

name VARCHAR(255) NOT NULL -- Company name must be provided

```
);
```

-- Each company can manage multiple warehouses

```
CREATE TABLE warehouses (
```

```
    id SERIAL PRIMARY KEY,      -- Unique ID for each warehouse
```

```
    company_id INT REFERENCES companies(id), -- This warehouse belongs to a company
```

```
    name VARCHAR(255) NOT NULL    -- Name of the warehouse (e.g., "Main Warehouse")
```

```
);
```

-- Stores all products sold or managed through the system

```
CREATE TABLE products (
```

```
    id SERIAL PRIMARY KEY,      -- Unique product identifier
```

```
    name VARCHAR(255) NOT NULL,    -- Name of the product
```

```
    sku VARCHAR(100) UNIQUE NOT NULL, -- SKU is a unique product code (like a barcode)
```

```
    price DECIMAL(10,2),        -- Product price with decimal accuracy
```

```
    is_bundle BOOLEAN DEFAULT FALSE -- Marks if this product is a bundle made of other products
```

```
);
```

-- Keeps track of how much stock is available for each product in each warehouse

```
CREATE TABLE inventory (
```

```
    id SERIAL PRIMARY KEY,      -- Unique ID for each stock entry
```

```
    product_id INT REFERENCES products(id), -- Which product this inventory entry is for
```

```
    warehouse_id INT REFERENCES warehouses(id), -- Which warehouse this stock is stored in
```

```
    quantity INT NOT NULL,      -- How many units are available
```

```
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- When this stock was last
updated
);
```

-- Acts as a history log for inventory changes — useful for audits and debugging

```
CREATE TABLE inventory_history (
    id SERIAL PRIMARY KEY,          -- Unique ID for each log entry
    product_id INT,                 -- Which product's stock changed
    warehouse_id INT,              -- Which warehouse was affected
    change INT,                    -- Quantity change (e.g., +10 for restock, -5 for sale)
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Time when the change
occurred
);
```

-- Stores contact details of suppliers who provide products

```
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,          -- Unique ID for each supplier
    name VARCHAR(255),             -- Name of the supplier company
    contact_email VARCHAR(255)     -- Email address to place orders or contact them
);
```

-- Connects products to the suppliers who provide them

```
CREATE TABLE product_suppliers (
    id SERIAL PRIMARY KEY,          -- Unique link ID between a product and a supplier
    product_id INT REFERENCES products(id), -- The product being supplied
    supplier_id INT REFERENCES suppliers(id) -- The supplier who provides this product
);
```

-- Defines what's inside a bundled product (e.g., gift sets or combos)

```
CREATE TABLE product_bundles (  
    id SERIAL PRIMARY KEY,           -- Unique ID for each bundle composition  
    bundle_id INT REFERENCES products(id), -- The main bundle product  
    child_product_id INT REFERENCES products(id), -- The individual product inside the  
    bundle  
    quantity INT                     -- How many units of the child product are included  
);
```

Design Rationale

- **Normalization:** I've separated entities like suppliers and bundles into their own tables with mapping tables (product_suppliers, product_bundles). This avoids data duplication and makes the schema easier to maintain.
- **Indexes:** Creating UNIQUE and FOREIGN KEY constraints on columns like sku, product_id, and warehouse_id automatically creates indexes, which significantly speeds up data retrieval.
- **Timestamps:** The last_updated and inventory_history tables provide a clear audit trail of when inventory levels change, which is crucial for tracking down discrepancies.
- **Scalability:** The modular design, where each table has a distinct responsibility, allows the system to grow. For example, we can add more details to warehouses without affecting the products table.

Gaps and Clarifying Questions

To refine this design, I would ask the product team:


1. **SKU Scope:** Should SKUs be unique across the entire platform, or can two different companies use the same SKU?
2. **Sales Granularity:** Is it necessary to track which warehouse a sale originated from? This would impact the design of a sales table.
3. **Bundle Pricing:** How is the price of a bundle determined? Is it a sum of its components, or can it have its own custom price?
4. **Supplier Complexity:** Can a product be provided by more than one supplier? Can suppliers also provide bundles directly?

Part 3: API Implementation (Java Spring Boot)

Below is a robust and well-structured implementation of the low-stock alert API using Java Spring Boot.

JPA Entities (The Data Model)

These Java classes map directly to the database tables.

 Product, Warehouse, and Inventory Entities

Java

@Entity

```
public class Product {  
    @Id private Long id;  
    private String name; // Name of the product  
    private String sku; // Unique identifier across all products  
    private String type; // Optional field to categorize product (e.g., 'bundle', 'single')  
}
```

@Entity

```
public class Warehouse {  
    @Id private Long id;  
    private String name; // Warehouse name  
    @ManyToOne private Company company; // Warehouse belongs to a company  
}
```

@Entity

```
public class Inventory {  
    @Id private Long id;  
    private int quantity; // Available stock in this warehouse  
    private LocalDateTime lastUpdated; // Timestamp of last update  
    @ManyToOne private Product product; // Product linked to this inventory
```

```
    @ManyToOne private Warehouse warehouse; // Which warehouse this inventory
belongs to
}
```

Sale, Threshold, and Supplier Entities

Java

@Entity

```
public class Sale {
```

```
    @Id private Long id;
```

```
    private LocalDateTime soldOn; // Sale timestamp
```

```
    @ManyToOne private Product product; // Which product was sold
```

```
    @ManyToOne private Warehouse warehouse; // From which warehouse the product
was sold
}
```

@Entity

```
public class ProductThreshold {
```

```
    @Id private Long id;
```

```
    private int threshold; // Minimum stock level before alerting
```

```
    @OneToOne private Product product; // Threshold is defined per product
```

```
}
```

@Entity

```
public class Supplier {
```

```
    @Id private Long id;
```

```
    private String name;
```

```
    private String contactEmail; // Email used for reordering
```

```
}
```


@Entity

```
public class ProductSupplier {  
    @Id private Long id;  
    @ManyToOne private Product product;  
    @ManyToOne private Supplier supplier; // Product can have multiple suppliers  
}
```

DTO: LowStockAlertResponse.java (The API Response Structure)

This class defines the exact format of the JSON object returned by the API .

Java

@Data

```
public class LowStockAlertResponse {  
    private Long productId;  
    private String productName;  
    private String sku;  
    private Long warehouseId;  
    private String warehouseName;  
    private int currentStock;  
    private int threshold;  
    private Integer daysUntilStockout; // Optional: estimated stockout based on recent sales  
    private SupplierDTO supplier;
```

@Data

```
public static class SupplierDTO {  
    private Long id;  
    private String name;  
    private String contactEmail;  
}
```

```
}
```

Service: AlertService.java (The Business Logic)

This service contains the core logic for identifying low-stock products.

Java

@Service

```
public class AlertService {
```

```
    @Autowired private WarehouseRepository warehouseRepo;
```

```
    @Autowired private InventoryRepository inventoryRepo;
```

```
    @Autowired private SaleRepository saleRepo;
```

```
    @Autowired private ProductThresholdRepository thresholdRepo;
```

```
    @Autowired private ProductSupplierRepository productSupplierRepo;
```

```
    public List<LowStockAlertResponse> getLowStockAlerts(Long companyId) {
```

```
        List<LowStockAlertResponse> alerts = new ArrayList<>();
```

```
        // 1. Get all warehouses for the given company
```

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

```
        // 2. Loop through each warehouse to check its inventory
```

```
        for (Warehouse warehouse : warehouses) {
```

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

```
            for (Inventory inv : inventories) {
```

```
                Product product = inv.getProduct();
```

```
                // 3. Check for recent sales activity to avoid alerting on stagnant stock
```

```

        LocalDateTime sevenDaysAgo = LocalDateTime.now().minusDays(7);

        long salesCount =
saleRepo.countByProductIdAndWarehouseIdAndSoldOnAfter(
            product.getId(), warehouse.getId(), sevenDaysAgo);

        if (salesCount == 0) continue; // Skip if no recent sales

        // 4. Get the low-stock threshold for this specific product

        Optional<ProductThreshold> optionalThreshold =
thresholdRepo.findByProductId(product.getId());

        if (optionalThreshold.isEmpty()) continue; // Skip if no threshold is set

        int threshold = optionalThreshold.get().getThreshold();

        // 5. Trigger an alert only if the current stock is at or below the threshold
        if (inv.getQuantity() > threshold) continue;

        // 6. Calculate an estimated 'days until stockout' based on recent sales velocity
        double avgDailySales = salesCount / 7.0;
        Integer daysUntilStockout = avgDailySales > 0 ?
            (int) Math.round(inv.getQuantity() / avgDailySales) : null;

        // 7. Find the supplier information for reordering

        Optional<ProductSupplier> ps =
productSupplierRepo.findFirstByProductId(product.getId());

        // 8. Build the final alert object with all the necessary information

        LowStockAlertResponse alert = new LowStockAlertResponse();
        alert.setProductId(product.getId());

```

```

        alert.setProductName(product.getName());
        alert.setSku(product.getSku());
        alert.setWarehouseld(warehouse.getId());
        alert.setWarehouseName(warehouse.getName());
        alert.setCurrentStock(inv.getQuantity());
        alert.setThreshold(threshold);
        alert.setDaysUntilStockout(daysUntilStockout);

        // 9. Add supplier details if they exist
        ps.isPresent(s -> {
            LowStockAlertResponse.SupplierDTO dto = new
LowStockAlertResponse.SupplierDTO();
            dto.setId(s.getSupplier().getId());
            dto.setName(s.getSupplier().getName());
            dto.setContactEmail(s.getSupplier().getContactEmail());
            alert.setSupplier(dto);
        });

        alerts.add(alert);
    }
}
return alerts;
}
}

```

Controller: AlertController.java (The API Endpoint)

This controller exposes the service logic as a public API endpoint.

Java

@RestController

```

@RequestMapping("/api/companies")

public class AlertController {

    @Autowired

    private AlertService alertService;

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

    public ResponseEntity<Map<String, Object>> getLowStockAlerts(@PathVariable Long
companyId) {

        // Fetch the list of low-stock alerts from the service

        List<LowStockAlertResponse> alerts = alertService.getLowStockAlerts(companyId);

        // Format the response to match the specification [cite: 57-77]

        Map<String, Object> response = new HashMap<>();

        response.put("alerts", alerts);

        response.put("total_alerts", alerts.size());

        return ResponseEntity.ok(response); // Return a 200 OK response with the data
    }
}

```

JPA Repositories (Data Access Layer)

These interfaces allow the service to communicate with the database without writing boilerplate SQL.

Java

```

// Finds all warehouses belonging to a specific company

public interface WarehouseRepository extends JpaRepository<Warehouse, Long> {

    List<Warehouse> findByCompanyId(Long companyId);

}

```

// Finds all inventory items within a specific warehouse

```
public interface InventoryRepository extends JpaRepository<Inventory, Long> {  
    List<Inventory> findByWarehouseId(Long warehouseId);  
}
```

// Counts sales for a product in a warehouse after a certain date

```
public interface SaleRepository extends JpaRepository<Sale, Long> {  
    long countByProductIdAndWarehouseIdAndSoldOnAfter(Long productId, Long  
warehouseId, LocalDateTime after);  
}
```

// Finds the low-stock threshold for a given product

```
public interface ProductThresholdRepository extends  
JpaRepository<ProductThreshold, Long> {  
    Optional<ProductThreshold> findByProductId(Long productId);  
}
```

// Finds the first associated supplier for a given product

```
public interface ProductSupplierRepository extends JpaRepository<ProductSupplier,  
Long> {  
    Optional<ProductSupplier> findFirstByProductId(Long productId);  
}
```