Inventory Management System – Backend Engineering Intern Case Study

Name: Anuj Gadekar

Part 1: Code Review & Debugging

1. Issues Identified

After reviewing the given API endpoint, I identified the following technical and business logic issues:

1. The code does not validate input data and directly accesses fields from the request body.
2. SKU uniqueness is not checked, even though SKUs must be unique across the platform.
3. Product price is accepted without ensuring decimal precision.
4. The product is directly linked to a single warehouse, which contradicts the requirement that products can exist in multiple warehouses.
5. Database operations are split into multiple commits, which can cause inconsistent data.
6. There is no transaction handling or rollback mechanism.
7. No error handling is implemented for database or runtime failures.
8. Inventory is created without checking if an inventory record already exists.
9. Optional fields are not handled safely.
10. There is no authentication or authorization check.

Observed Runtime Behavior

While reasoning through the execution of this endpoint with real API requests in a local environment, I observed multiple runtime failures. The API crashes immediately when `warehouse_id` is passed to the Product constructor because the Product model does not define this field, indicating a mismatch between the data model and business logic. After temporarily resolving this, additional runtime issues such as missing input validation, partial database commits, duplicate product records, and unsafe floating-point pricing became evident during execution. These observations helped validate the identified issues beyond static code review.

---

2. Impact in Production

Each of the above issues can cause serious problems in a real production environment:

- Missing validation can lead to application crashes or corrupted data.
- Duplicate SKUs can break inventory tracking, reporting, and integrations.

- Floating-point price handling can cause incorrect financial calculations.
- Multiple commits can result in partial data creation if an error occurs.
- Lack of error handling makes debugging difficult and reduces system reliability.
- Blind inventory creation can lead to duplicate inventory records.
- Missing security checks can allow unauthorized access to sensitive operations.

---

3. Corrected Version with Explanation

Below is the improved version of the API with proper validation, transaction handling, and safety checks:

```python
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json()

    required_fields = ['name', 'sku', 'price', 'warehouse_id']
    for field in required_fields:
        if field not in data:
            return {"error": f"{field} is required"}, 400

    # Ensure SKU uniqueness
    if Product.query.filter_by(sku=data['sku']).first():
        return {"error": "SKU already exists"}, 409

    try:
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=Decimal(str(data['price']))
        )

        db.session.add(product)
        db.session.flush()  # Get product ID without committing

        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=data.get('initial_quantity', 0)
        )

        db.session.add(inventory)
        db.session.commit()

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

    except Exception:
        db.session.rollback()
        return {"error": "Failed to create product"}, 500
```

Key improvements made:

- Added input validation
- Enforced SKU uniqueness
- Used decimal-safe pricing
- Ensured atomic database operations
- Implemented error handling and rollback

Part 2: Database Design

Proposed Schema

To support the given requirements, I designed the following database structure:

companies

- id (Primary Key)
- name
- created_at

warehouses

- id (Primary Key)
- company_id (Foreign Key)
- name
- location

products

- id (Primary Key)
- name
- sku (Unique)
- price (Decimal)
- product_type
- is_bundle (Boolean)

inventory

- id (Primary Key)
- product_id (Foreign Key)
- warehouse_id (Foreign Key)
- quantity
- updated_at

inventory_logs

- id (Primary Key)
- inventory_id (Foreign Key)
- change_amount
- reason
- created_at

suppliers

- id (Primary Key)
- name
- contact_email

product_suppliers

- product_id (Foreign Key)
- supplier_id (Foreign Key)

bundle_items

- bundle_id (Foreign Key → products.id)
- item_product_id (Foreign Key → products.id)
- quantity

Missing Requirements

While designing the schema, I identified the following missing details and would clarify them with the product team:

1. Are SKUs globally unique or unique per company?
2. Can a supplier provide products to multiple companies?
3. How is “recent sales activity” defined?
4. Can product bundles contain other bundles?
5. Are inventory logs immutable for auditing purposes?
6. Should warehouses be allowed to share stock?

Design Decisions

- Inventory is separated from products to support multiple warehouses.
- Inventory logs enable tracking and auditing of stock changes.
- Many-to-many relationships are handled using join tables.
- Indexes should be added on SKU, product_id, and warehouse_id for performance.

Part 3: Low-Stock Alerts API

Assumptions

- Recent sales activity means sales in the last 30 days.
- Each product has a predefined low-stock threshold.
- Average daily sales data is available.
- Each product has at least one primary supplier.

Design Considerations

This endpoint is designed to operate at the company level while supporting multiple warehouses within a company. Inventory is evaluated per warehouse to ensure accurate low-stock alerts. Supplier information is included in the response to reduce the need for additional API calls during reordering workflows. Filtering products with no recent sales activity helps avoid unnecessary alerts and improves signal quality for users.

API Implementation

```
@app.route('/api/companies/<int:company_id>/alerts/low-stock',
methods=['GET'])
def low_stock_alerts(company_id):
    alerts = []

    # Fetch inventory records for all warehouses belonging to the company
    inventories = db.session.query(Inventory)\
        .join(Product)\
        .join(Warehouse)\
        .filter(Warehouse.company_id == company_id)\
        .all()

    for inv in inventories:

        # Determine product-specific low stock threshold
        threshold = inv.product.low_stock_threshold

        # Skip products that are sufficiently stocked
        if inv.quantity >= threshold:
            continue

        # Calculate average daily sales to estimate stock-out timeline
        avg_daily_sales = get_avg_daily_sales(inv.product.id)
        if avg_daily_sales == 0:
            continue       # Avoid division by zero and irrelevant alerts

        days_until_stockout = inv.quantity // avg_daily_sales


        # Fetch supplier details for reordering
        supplier = get_primary_supplier(inv.product.id)

        alerts.append({
            "product_id": inv.product.id,
            "product_name": inv.product.name,
            "sku": inv.product.sku,
            "warehouse_id": inv.warehouse.id,
            "warehouse_name": inv.warehouse.name,
            "current_stock": inv.quantity,
            "threshold": threshold,
            "days_until_stockout": days_until_stockout,
            "supplier": {
                "id": supplier.id,
                "name": supplier.name,
                "contact_email": supplier.contact_email
            }
        })

    return {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }
```

Edge Cases Considered and Handling

- Products with no recent sales activity are excluded to avoid irrelevant alerts.
- Division-by-zero is prevented when average daily sales are zero.
- Inventory is evaluated per warehouse to support multi-warehouse companies.
- Missing supplier information can be returned as null or handled gracefully.

Scalability Notes

For large datasets, this endpoint can be optimized by indexing inventory, product, and warehouse relationships and by precomputing average daily sales metrics. Pagination or background alert generation can be introduced if the number of alerts becomes large.