

Atul kanadje

Part 1: Code Review & Debugging

1. No input validation

Directly uses data ['field'] without checking if fields exist or are valid.

Impact - Missing or invalid fields will raise key errors.

Bad user input can crash the API.

2. SKU uniqueness not enforced

No check to ensure sku is unique across the platform.

Impact - because of it data can corrupt.

3. Products tied to a single warehouse

warehouse_id=data['warehouse_id']

Product is created with warehouse id.

Impact-Impossible to stock the same product in multiple warehouses later.

4. Multiple commits without transaction safety

db.session.commit() - Product

db.session.commit() - Inventory

Impact-Leaves database in inconsistent state.

5. Price precision issue

price=data['price'], likely stored as float.

Impact-Financial inaccuracies

Correct code -

```
from decimal import Decimal
from sqlalchemy.exc import IntegrityError
from flask import request, jsonify

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

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

    if Product.query.filter_by(sku=data['sku']).first():
        return jsonify({"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()

if 'warehouse_id' in data and 'initial_quantity' in data:
    inventory = Inventory.query.filter_by(
        product_id=product.id,
        warehouse_id=data['warehouse_id']
    ).first()

    if inventory:
        inventory.quantity += data['initial_quantity']
    else:
        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=data['initial_quantity']
        )
    db.session.add(inventory)

db.session.commit()

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

except IntegrityError:
    db.session.rollback()
    return jsonify({"error": "Database constraint violation"}), 400

except Exception as e:
    db.session.rollback()
    return jsonify({"error": str(e)}), 500
```

This code is correct because of i fix the Input validation, Explicit SKU check, Product decoupled from warehouse, Single transaction, Existence check, Proper exception handling etc.

Part 2: Database Design

1. Database Schema Design

1.Companies-Stores organizations that own warehouses and products.

```
companies (
    id      BIGINT PRIMARY KEY,
    name    VARCHAR(255) NOT NULL,
    created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

2.warehouses-Each company can have multiple warehouses.

```
warehouses (
    id      BIGINT PRIMARY KEY,
    company_id  BIGINT NOT NULL,
    name    VARCHAR(255),
    location  VARCHAR(255),
    created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (company_id) REFERENCES companies(id),
    INDEX idx_company_id (company_id)
)
```

3.products-Represents individual sellable products.

```
products (
    id      BIGINT PRIMARY KEY,
    sku     VARCHAR(50) UNIQUE NOT NULL,
    name    VARCHAR(255) NOT NULL,
    price   DECIMAL(10,2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE,
    created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

4.inventory-Tracks quantity of each product in each warehouse.

```
inventory (
    id      BIGINT PRIMARY KEY,
    product_id  BIGINT NOT NULL,
    warehouse_id  BIGINT NOT NULL,
    quantity   INT NOT NULL DEFAULT 0,
```

```
updated_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
FOREIGN KEY (product_id) REFERENCES products(id),  
FOREIGN KEY (warehouse_id) REFERENCES warehouses(id),  
UNIQUE (product_id, warehouse_id),  
INDEX idx_product_warehouse (product_id, warehouse_id)  
)
```

5.inventory changes-Audit table to track inventory history.

```
inventory_changes (  
    id        BIGINT PRIMARY KEY,  
    inventory_id  BIGINT NOT NULL,  
    change_type   VARCHAR(50), -- INBOUND, OUTBOUND, ADJUSTMENT  
    quantity_change INT NOT NULL,  
    reason        VARCHAR(255),  
    created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  
    FOREIGN KEY (inventory_id) REFERENCES inventory(id),  
    INDEX idx_inventory_id (inventory_id)  
)
```

6.suppliers-Represents vendors supplying products.

```
suppliers (  
    id        BIGINT PRIMARY KEY,  
    name      VARCHAR(255) NOT NULL,  
    contact_email  VARCHAR(255),  
    created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
)
```

7.supplier products-Many-to-many mapping between suppliers and products.

```
supplier_products (  
    supplier_id   BIGINT NOT NULL,  
    product_id    BIGINT NOT NULL,  
  
    PRIMARY KEY (supplier_id, product_id),  
    FOREIGN KEY (supplier_id) REFERENCES suppliers(id),  
    FOREIGN KEY (product_id) REFERENCES products(id)  
)
```

8.product bundles-Defines bundle composition (product contains other products).

```
product_bundles (  
    bundle_id     BIGINT NOT NULL,  
    component_id   BIGINT NOT NULL,  
    quantity      INT NOT NULL,  
  
    PRIMARY KEY (bundle_id, component_id),
```

```
FOREIGN KEY (bundle_id) REFERENCES products(id),  
FOREIGN KEY (component_id) REFERENCES products(id)  
)
```

2.List of questions I'll ask the product team about missing requirements

- 1.Should SKU be unique globally or per company?
- 2.Can the same supplier supply different prices for the same product?
- 3.Are inventory adjustments manual or system-generated only?
- 4.Can bundles be nested (bundle inside bundle)?
- 5.Should inventory changes record user or system actor?
- 6.Are warehouses allowed to share stock?

3.Design Decisions Explained

- Normalization
- Reduced redundancy by separating products, warehouses, and inventory
Supports scale and flexibility

Audit Table

- Inventory changes stored separately
- Prevents loss of historical data
- Enables compliance and analytics

constraints

- UNIQUE (product_id, warehouse_id) prevents data duplication
- FOREIGN KEY constraints enforce referential integrity

Part 3: API Implementation

1.Assumptions-Because requirements are incomplete, the following assumptions are made:
Product
Each product has a product_type
Low-stock threshold is derived from product_types.threshold

Inventory-
Inventory is tracked per product_id + warehouse_id
Inventory quantity is always non-negative

Sales

Recent sales = at least 1 sale in the last 30 days

Sales data exists in a sales table

Supplier

Each product has a **primary supplier**

Supplier contact info is required for reordering

2. Relevant Tables

companies(id)

warehouses(id, company_id, name)

products(id, sku, name, product_type_id)

product_types(id, name, low_stock_threshold)

inventory(id, product_id, warehouse_id, quantity)

sales(id, product_id, warehouse_id, quantity, sold_at)

suppliers(id, name, contact_email)

supplier_products(product_id, supplier_id)

3. API Implementation

```
from datetime import datetime, timedelta
```

```
from flask import jsonify
```

```
from sqlalchemy import func
```

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

```
def low_stock_alerts(company_id):
```

```
    """
```

Returns low-stock alerts for a company.

```
    """
```

```
    alerts = []
```

```
    total_alerts = 0
```

```
    recent_days = 30
```

```
    recent_date = datetime.utcnow() - timedelta(days=recent_days)
```

```
    warehouses = Warehouse.query.filter_by(company_id=company_id).all()
```

```
    warehouse_ids = [w.id for w in warehouses]
```

```
if not warehouse_ids:  
    return jsonify({"alerts": [], "total_alerts": 0})  
  
results = (  
    db.session.query(  
        Inventory,  
        Product,  
        Warehouse,  
        ProductType,  
        Supplier  
    )  
    .join(Product, Inventory.product_id == Product.id)  
    .join(ProductType, Product.product_type_id == ProductType.id)  
    .join(Warehouse, Inventory.warehouse_id == Warehouse.id)  
    .join(SupplierProduct, SupplierProduct.product_id == Product.id)  
    .join(Supplier, SupplierProduct.supplier_id == Supplier.id)  
    .filter(Warehouse.id.in_(warehouse_ids))  
    .all()  
)
```

for inventory, product, warehouse, product_type, supplier in results:

```
recent_sales = (  
    db.session.query(func.sum(Sale.quantity))  
    .filter(  
        Sale.product_id == product.id,  
        Sale.warehouse_id == warehouse.id,  
        Sale.sold_at >= recent_date  
    )  
    .scalar()  
)
```

```
if not recent_sales or recent_sales <= 0:  
    continue
```

```
avg_daily_sales = recent_sales / recent_days
```

```
if inventory.quantity >= product_type.low_stock_threshold:  
    continue
```

```
days_until_stockout = int(inventory.quantity / avg_daily_sales)
```

```
alerts.append({  
    "product_id": product.id,
```

```
"product_name": product.name,
"sku": product.sku,
"warehouse_id": warehouse.id,
"warehouse_name": warehouse.name,
"current_stock": inventory.quantity,
"threshold": product_type.low_stock_threshold,
"days_until_stockout": days_until_stockout,
"supplier": {
    "id": supplier.id,
    "name": supplier.name,
    "contact_email": supplier.contact_email
}
})

total_alerts += 1

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