BYNRY ASSIGNEMENT

Part 1: The given endpoint compiled successfully but failed in production due to several logical and transactional issues. Below, I've identified the key problems, explained their impacts, and provided corrected code following best practices for validation, transactions, and error handling.

ISSUE	IMPACT	EXAMPLE	FIXES
Missing or Invalid Input Fields	If any of the mentioned keys in the JSON is not sent , the entire endpoint will crash.	If the user doesn't include for example warehouse_id, or if the key is misspelled.	Use .get(), it returns None if any missing values exist.
No Validation of Field Types	Random crashes when the data sent doesn't match the data expected.	If the warehouse id expected is a number but the user sends it as a string, DB will throw an error.	Validate field types before saving.
Two Separate commit calls for the Database.	Data becomes inconsistent.	If the first call saves the products but the second call fails to update the inventory, the product will exist without any record in the inventory.	Combine both the transactions under one commit.
No Error-Handling	Harder to debug production level codes.	The errors when occurred won't show what exactly went wrong.	Use the try-catch blocks.
No response status code.	Misinterpretation of the results	No cases for bad request, already exists, etc.	Use meaningful status codes.
SKU uniqueness not checked.	Stock alerts, analysis could go wrong.	Code doesn't check if the sky already exists.	In the database , make the sku unique, also add a code to check if it already exists.
Product tied to only one warehouse.	Will have to create multiple copies of the same product for the warehouses.	Code is storing a warehouse_id directly to a product. Doesn't follow the many-many relationship.	Remove warehouse_id from product.
Monetary value stored as floating number.	Creates accounting mismatches.	Rounding erros.	Use decimal for prices.
Supplier Information Missing	This breaks supplier tracking and reordering functionality.	The API doesn't store or connect any supplier to the product.	Add Supplier_id field

Corrected code:

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

```
from decimal import Decimal

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

    # Input Validation
    required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
    missing = [field for field in required_fields if field not in data]
    if missing:
        return jsonify({"error": f"Missing fields: {', '.join(missing)}"}), 400

    # Type Validation
    try:
        price = Decimal(str(data['price']))  # avoids float rounding issues
        warehouse_id = int(data['warehouse_id'])
        quantity = int(data['initial_quantity'])
    except (ValueError, TypeError):
        return jsonify({"error": "Invalid data type for price, warehouse_id, or
quantity"}), 400

    # Check for duplicate sku
    existing_product = Product.query.filter_by(sku=data['sku']).first()
    if existing_product:
        return jsonify({"error": "SKU already exists"}), 409

    try:
        # Create Product
        product = Product(
            name=data.get('name'),
            sku=data.get('sku'),
            price=price
            supplier_id=supplier_id

        )

        db.session.add(product)
        db.session.flush()  # ensures product.id is available before inventory creation

        # inventory save
        inventory = Inventory(
            product_id=product.id,
            warehouse_id=warehouse_id,
            quantity=quantity
        )
        db.session.add(inventory)

        # One commit for both product and inventory saves.
```

```
    db.session.commit()

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

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

Part 2: Database Design.

1- Company Table:
Each company using the platform will have a record.

```
CREATE TABLE companies (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    contact_email VARCHAR(100)
);
```

Other tables are linked to this table.

2- Warehouse table:

```
CREATE TABLE warehouses (
    id INT PRIMARY KEY AUTO_INCREMENT,
    company_id INT,
    name VARCHAR(100) NOT NULL,
    location VARCHAR(255),
    FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

The company_id links each warehouse to its parent company.

3- Supplier Table:

```
CREATE TABLE suppliers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    company_id INT,
    name VARCHAR(100) NOT NULL,
    contact_email VARCHAR(100),
    phone VARCHAR(20),
    FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

Each supplier provides products to a specific company. (Assumption)

4-Product Table:

```
CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    company_id INT,
    supplier_id INT,
    name VARCHAR(100) NOT NULL,
    sku VARCHAR(50) UNIQUE NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (company_id) REFERENCES companies(id),
```

```
    FOREIGN KEY (supplier_id) REFERENCES suppliers(id)
);
```

Decimal(10,2) avoids floating point errors.
Products belong to companies and are supplied by suppliers.
SKU is unique, and the price uses a decimal type to avoid rounding errors.

5- Inventory Table:

```
CREATE TABLE inventory (
    id INT PRIMARY KEY AUTO_INCREMENT,
    product_id INT,
    warehouse_id INT,
    quantity INT DEFAULT 0,
    threshold INT DEFAULT 10,
    FOREIGN KEY (product_id) REFERENCES products(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses(id),
    UNIQUE (product_id, warehouse_id)
);
```

This table connects products and warehouses (many-to-many).
It stores the quantity of each product per warehouse.
The threshold helps in creating low stock alerts.
Unique(product_id, warehouse_id) ensures one inventory record per product per warehouse.

6- Inventory History Table:

```
CREATE TABLE inventory_history (
    id INT PRIMARY KEY AUTO_INCREMENT,
    inventory_id INT,
    change_type VARCHAR(20),
    quantity_changed INT,
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (inventory_id) REFERENCES inventory(id)
);
```

This table records stock changes over time (e.g., sales, restocks).

7- Product Bundle Table:

```
CREATE TABLE product_bundles (
    bundle_id INT,
    component_id INT,
    quantity INT,
    PRIMARY KEY (bundle_id, component_id),
    FOREIGN KEY (bundle_id) REFERENCES products(id),
    FOREIGN KEY (component_id) REFERENCES products(id)
);
```

8- Sales Table (to track recent sales activity)

```
CREATE TABLE sales (
    id INT PRIMARY KEY AUTO_INCREMENT,
    product_id INT NOT NULL,
    warehouse_id INT NOT NULL,
    company_id INT NOT NULL,
    quantity INT NOT NULL,
    date DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES products(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses(id),
    FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

Questions /Gaps in requirements to clarify with the Product team:

- Can a product have multiple suppliers? (Here, one supplier is linked to one product but in reality some products may come from different suppliers)
- Should we track reorder quantity or just threshold? (Can be linked together, whenever the threshold is reached , quantity to be reordered agains needs to be mentioned)
- Should Inventory History also record who made the changes?
- Can warehouses be shared among companies? (Current design assumes, each warehouse belongs to one company)
- Do the bundles also contain bundles?
- Are bundle products treated as separate SKUs?
- Do the warehouses have capacity limit? (If yes, products could be relocated to other warehouses)

Part 3: API Implementation

The endpoint has to :
- Identify all the products belonging to a company across all the warehouses.
- Check inventory level for all the products and compare with the corresponding threshold for it.
- Filter results according to recent sales activity.
- Fetch supplier results to reorder.
- Return results according to the given JSON format.

Assumptions I made:
- Each company has one or more warehouses, but warehouses are not shared across companies.
- Each product is linked to one supplier.
- The low stock threshold is stopped in the inventory table.
- Recent Sales are determined by checking entries in a sales table where sales occurred in the past 30 days.
- If stock or supplier details are missing , product is skipped in final response.

Endpoint Specification:

GET /api/companies/{company_id}/alerts/low-stock

Expected Response Format:

```
{
  "alerts": [
    {
      "product_id": 123,
      "product_name": "Widget A",
      "sku": "WID-001",
      "warehouse_id": 456,
      "warehouse_name": "Main Warehouse",
      "current_stock": 5,
      "threshold": 20,
      "days_until_stockout": 12,
      "supplier": {
        "id": 789,
        "name": "Supplier Corp",
        "contact_email": "orders@supplier.com"
      }
    }
  ],
  "total_alerts": 1
}
```

Implementation:

```
from flask import Flask, jsonify, request
from datetime import datetime, timedelta
from models import db, Company, Warehouse, Product, Inventory, Supplier, Sales

app = Flask(__name__)

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def get_low_stock_alerts(company_id):
    try:
        # Get all warehouses for this company
        warehouses = Warehouse.query.filter_by(company_id=company_id).all()
        if not warehouses:
            return jsonify({"message": "No warehouses found for this company"}), 404

        warehouse_ids = [w.id for w in warehouses]

        # Get inventory items for these warehouses
        inventories =
Inventory.query.filter(Inventory.warehouse_id.in_(warehouse_ids)).all()
        alerts = []

        for inv in inventories:
            # Skip if product not found
            product = Product.query.get(inv.product_id)
            if not product:
                continue

            # Check recent sales (within last 30 days)
            recent_sales = Sales.query.filter(
                Sales.product_id == product.id,
                Sales.date >= datetime.now() - timedelta(days=30)
            ).count()

            if recent_sales == 0:
                continue  # skip inactive products

            # Check low stock condition
            if inv.quantity <= inv.threshold:
                supplier = Supplier.query.get(product.supplier_id)
                warehouse = Warehouse.query.get(inv.warehouse_id)

                alert = {
                    "product_id": product.id,
                    "product_name": product.name,
                    "sku": product.sku,
                    "warehouse_id": warehouse.id,
                    "warehouse_name": warehouse.name,
```

```
                "current_stock": inv.quantity,
                "threshold": inv.threshold,
                "days_until_stockout": 10,  # sample estimate
                "supplier": {
                    "id": supplier.id,
                    "name": supplier.name,
                    "contact_email": supplier.contact_email
                } if supplier else None
            }
            alerts.append(alert)

    return jsonify({
        "alerts": alerts,
        "total_alerts": len(alerts)
    }), 200

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