Part 1: Code Review & Debugging

List of Issues Identified and Solved

1. Missing Input Validation

- Issue & Impact: The original code did not check whether all required fields like name, sku, price, and warehouse_id were present in the request.
- Fix: I added a validation step to ensure all required fields are provided before proceeding, which helps prevent crashes and inconsistent data entries.

2. No SKU Uniqueness Check

- Issue & Impact: The original code did not verify whether the SKU already existed in the database, potentially leading to duplicate entries.
- Fix: I added a database query to check for SKU uniqueness. If a duplicate is found, the API now returns an appropriate error response.

3. Price Format and Validation

- Issue & Impact: The code accepted the price field without validating its type or whether it was negative or not, which could result in incorrect or invalid data.
- Fix: I included checks to make sure the price is a valid float and non negative. If it is not, a clear error message is returned.

4. No Warehouse Existence Check

- Issue & Impact: The code assumed the warehouse_id existed without checking, which could cause foreign key constraint violations.
- Fix: I added a check using Warehouse.query.get() to ensure the specified warehouse actually exists.

5. Missing or Optional initial_quantity Field

- Issue & Impact: The original code expected initial_quantity to be present, which could lead to a KeyError if it was not provided.
- Fix: I used data.get('initial_quantity', 0) to safely handle missing values by defaulting to 0.

6. Multiple Commits Causing Inconsistency

- Issue & Impact: The product was committed to the database before the inventory entry. If inventory creation failed afterward, it would leave behind a partially created product.
- Fix: I used db.session.flush() to retrieve the product ID without committing immediately. Then I committed both the product and inventory together in a single transaction, maintaining consistency.

7. No Error Handling

- Issue & Impact: Any unexpected error would crash the server and return a 500 error without proper rollback.
- Fix: I wrapped the operation in a try/except block, performed a rollback on failure, and returned a controlled and error message.

8. No HTTP Status Codes or User-Friendly Responses

- Issue & Impact: The original response did not use appropriate HTTP status codes or clear messages, which made it hard for clients to understand what went wrong.
- Fix: I ensured that each response has meaningful status codes (like 400 for bad requests, 201 for success) and structured, JSON messages.

9. Partial Business Logic Awareness (Product in Multiple Warehouses)

- Issue & Impact: The original code incorrectly assumed that a product is tied to only one warehouse.
- My Fix: While I did not completely refactor for full multi warehouse support, I structured the logic so that inventory is handled per warehouse, making the system more aligned with real world warehouse operations.

10. No Safe Defaulting or Flexibility

- Issue & Impact: The code assumed all fields were always present and correct, which reduced flexibility and robustness.
- My Fix: I implemented default values (like for initial_quantity) and added thorough validations to make the API more flexible, user-friendly, and resilient.

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

        if not data:
            return {"error": "No data provided"}, 400
```

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

existing_product = Product.query.filter_by(sku=data['sku']).first()
if existing_product:
    return {"error": "SKU already exists"}, 400

try:
    price = float(data['price'])
    if price < 0:
        return {"error": "Price cannot be negative"}, 400
except:
    return {"error": "Price must be a number"}, 400

warehouse = Warehouse.query.get(data['warehouse_id'])
if not warehouse:
    return {"error": "Warehouse not found"}, 400

product = Product(
    name=data['name'],
    sku=data['sku'],
    price=price,
    warehouse_id=data['warehouse_id']
)
db.session.add(product)
db.session.flush()  # This gets us the product.id without committing yet
```

```
        initial_quantity = data.get('initial_quantity', 0)

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

        db.session.commit()

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

    except Exception as e:
        db.session.rollback()
        return {"error": "Something went wrong"}, 500
```

Part 2: Database Design

1. Companies

```
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

2. Warehouses

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

3. Suppliers

```
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    contact_email VARCHAR(255)
);
```

4. Products

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) UNIQUE NOT NULL,
    price NUMERIC(10,2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE
);
```

5. Product-Supplier Relationship

```
CREATE TABLE product_suppliers (
   product_id INT REFERENCES products(id),
   supplier_id INT REFERENCES suppliers(id),
   PRIMARY KEY (product_id, supplier_id)
);
```

6. Inventory

```
CREATE TABLE inventory (
   product_id INT REFERENCES products(id),
   warehouse_id INT REFERENCES warehouses(id),
   quantity INT NOT NULL CHECK (quantity >= 0),
   PRIMARY KEY (product_id, warehouse_id)
);
```

Updated Gaps / Questions for Product Team

1. How should we track inventory changes over time?
2. Are product bundles considered their own product type, or are they more for marketing?
3. How to track which supplier delivered which batch of products (and when)?
4. Do suppliers provide products to multiple companies or only one?
5. What are valid units for quantity (units, boxes)?
6. Should product pricing vary per company or warehouse?

Decisions Explanation:

1. companies

This represents a business or organization using the platform.
Each company operates in its own space, so its data is kept separate from others.
A company can have one or more warehouses associated with it.

- id: Unique identifier for the company.
- name: The name of the company.

2. warehouses

These are the physical locations where a company stores its products.
Each warehouse belongs to a specific company, and a company can manage multiple warehouses to track inventory across locations.

- id: Unique identifier for the warehouse.
- company_id: Links the warehouse to its company.

3. products

This is the central catalog of items that are stored, sold, or tracked.
Each product has a unique SKU and a price, and you can later add more details like descriptions, categories, or whether the product is active.

- id: Unique identifier for the product.
- sku: A unique code used to identify the product (indexed for fast searches).
- price: Stored precisely using NUMERIC(10,2) for accurate currency handling.

4. suppliers

Suppliers are the businesses or individuals who provide products to the companies on the platform. This is a basic contact record for now, but it can be expanded to include things like preferred supplier status, auditing, or order histories in the future.

- id: Unique identifier for the supplier.
- name / contact info: Simple fields to identify and reach the supplier (can be expanded later).

5. product_suppliers

This is a join table that connects products to suppliers.
That means a single product can have multiple suppliers, and a supplier can offer multiple products.

- product_id and supplier_id: Together, they form a unique combination (composite key).

6. inventory

This table tracks how much of each product is available at each warehouse.
It supports the idea that a product can exist in multiple warehouses, and the quantity may vary by location.

- product_id + warehouse_id: Composite key ensures each product is tracked once per warehouse.
- quantity: Integer value showing current stock level. Must be 0 or higher (CHECK (quantity >= 0)).

Part 3: API Implementation

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

app = Flask(__name__)

LOW_STOCK_THRESHOLDS = {
    "electronics": 15,
    "furniture": 10,
    "grocery": 30,
    "default": 20
}

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
    try:
        alerts = []
        warehouses = Warehouse.query.filter_by(company_id=company_id).all()
        if not warehouses:
            return jsonify({"alerts": [], "total_alerts": 0}), 200

        warehouse_ids = [w.id for w in warehouses]
        thirty_days_ago = datetime.utcnow() - timedelta(days=30)

        recent_sales = db.session.query(Sales.product_id).filter(
            Sales.warehouse_id.in_(warehouse_ids),
            Sales.sold_at >= thirty_days_ago
        ).distinct().all()
```

```
    recent_product_ids = [row.product_id for row in recent_sales]
    if not recent_product_ids:
        return jsonify({"alerts": [], "total_alerts": 0}), 200

    inventories = Inventory.query.filter(
        Inventory.product_id.in_(recent_product_ids),
        Inventory.warehouse_id.in_(warehouse_ids)
    ).all()

    for inv in inventories:
        product = Product.query.get(inv.product_id)
        warehouse = next((w for w in warehouses if w.id == inv.warehouse_id), None)

        threshold = LOW_STOCK_THRESHOLDS.get(product.type,
LOW_STOCK_THRESHOLDS["default"])

        if inv.quantity < threshold:
            sales_count = db.session.query(Sales).filter(
                Sales.product_id == product.id,
                Sales.warehouse_id == inv.warehouse_id,
                Sales.sold_at >= thirty_days_ago
            ).count()

            avg_daily_sales = sales_count / 30 if sales_count else 0.1
            days_until_stockout = int(inv.quantity / avg_daily_sales)

            supplier = db.session.query(Supplier).join(ProductSupplier).filter(
                ProductSupplier.product_id == product.id
            ).first()

            alerts.append({
```

```
                "product_id": product.id,
                "product_name": product.name,
                "sku": product.sku,
                "warehouse_id": warehouse.id,
                "warehouse_name": warehouse.name,
                "current_stock": inv.quantity,
                "threshold": threshold,
                "days_until_stockout": days_until_stockout,
                "supplier": {
                    "id": supplier.id if supplier else None,
                    "name": supplier.name if supplier else None,
                    "contact_email": supplier.contact_email if supplier else None
                }
            })

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

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

In this endpoint, I start by fetching all warehouses associated with the given company. Then, I identify products that have recent sales activity (within the last 30 days) in those warehouses. For these products, I check current inventory levels and compare them against predefined low stock thresholds based on product type. If a product's quantity is below the threshold, estimate how many days of stock are left based on recent sales rate. The final response includes a list of each alert with supplier information to support quick reordering. The response includes all alerts along with a total count. The logic is wrapped in error handling to ensure graceful failure in edge cases.