**Inventory Management System for B2B SaaS**

**Name –** Aakash Raju Mohole

**Email –** [aakashmohole@gmail.com](mailto:aakashmohole@gmail.com)

**Github –**

**PART 1 - Code Review & Debugging — Approach & Analysis**

1. **Issues in the Provided code**

**Technical Issues –**

* No check before creating product; duplicate SKUs could be created.
* Two commits used instead of one atomic transaction, risking database inconsistency if one commit fails.
* Decimal price – The code trust that data[‘price’] is valid this price should be convert to decimal.
* Implement error handling if something fails it crashes of corrupts all data.
* It is unclear if the Product model is meant to be per-warehouse; if not, the logic here can duplicate products unnecessarily.

**Business Logic Issues**

* The design suggests a new product per warehouse, but product should have a single entry with inventory per warehouse.
* If product creation or inventory insertion in case fails, IDs might become out of sync and this create problem.
* If the app should support multiple companies, there’s no separation implemented we have to implement this.
* If inventory creation fails after product is added, the product remains without inventory—a broken state.

1. **Impact of Each Issue in Production**

* Duplicate SKUs lead to confusion in inventory tracking, order processing, and reporting and data will become redundant.
* Partial failures create split states, making rollback and consistency difficult.
* Invalid data can crash the server or lead to incomplete/corrupt entries.
* Inaccurate prices (due to float imprecision) can cause billing and accounting errors.
* Any exception causes 500 errors and possible data leaks or losses.
* Duplicating product entries for each warehouse breaks analytics, reporting, and increases maintenance burden.

1. **Correct Code Version with explanations**

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.json

# Validate required fields

required\_fields = ['name', 'sku', 'price', 'warehouse\_id']

for field in required\_fields:

if field not in data:

return jsonify({"error": f"Missing required field: {field}"}), 400

# Parse and validate price as Decimal

try:

price = Decimal(str(data['price']))

except:

return jsonify({"error": "Invalid price format"}), 400

# Check SKU uniqueness

if Product.query.filter\_by(sku=data['sku']).first():

return jsonify({"error": "SKU already exists"}), 409

# Begin transaction

try:

product = Product(

name=data['name'],

sku=data['sku'],

price=price

# exclude warehouse\_id, since product is global across warehouses

)

db.session.add(product)

db.session.flush() # gets the product.id before commit

inventory = Inventory(

product\_id=product.id,

warehouse\_id=data['warehouse\_id'],

quantity=data['initial\_quantity']

)

db.session.add(inventory)

db.session.commit()

except IntegrityError as e:

db.session.rollback()

return jsonify({"error": "Database error: " + str(e.orig)}), 500

except Exception as e:

db.session.rollback()

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

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

**PART 2: Database Design**

1. **Database Schema Design**

| **Entity** | **Attributes** | **Relationships** |
| --- | --- | --- |
| **Company** | id (PK), name, created\_at | 1-to-many with Warehouse, 1-to-many with Product |
| **Warehouse** | id (PK), company\_id (FK), name, location, created\_at | Many-to-1 with Company, 1-to-many with Inventory |
| **Product** | id (PK), company\_id (FK), name, sku, price, is\_bundle | Many-to-1 with Company, many-to-many with Supplier, many-to-many with Product (as bundle), 1-to-many with Inventory |
| **Supplier** | id (PK), name, contact\_info, created\_at | many-to-many with Product via Product\_Supplier |
| **Product\_Supplier** | product\_id (PK, FK), supplier\_id (PK, FK) | Many-to-1 with Product, many-to-1 with Supplier |
| **Product\_Bundle** | bundle\_id (PK, FK), product\_id (PK, FK), quantity | Many-to-1 with Product (bundle and component product) |
| **Inventory** | id (PK), product\_id (FK), warehouse\_id (FK), quantity | Many-to-1 with Product, many-to-1 with Warehouse, 1-to-many with Inventory\_Change |
| **Inventory\_Change** | id (PK), inventory\_id (FK), change\_amount, reason, changed\_at, changed\_by | Many-to-1 with Inventory |

* **SQL DDL**

**-- 1. Companies**

CREATE TABLE company (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

-- Add fields like address, contact info, etc.

created\_at TIMESTAMP NOT NULL DEFAULT NOW()

);

**-- 2. Warehouses**

CREATE TABLE warehouse (

id SERIAL PRIMARY KEY,

company\_id INTEGER NOT NULL REFERENCES company(id) ON DELETE CASCADE,

name VARCHAR(255) NOT NULL,

location VARCHAR(255),

created\_at TIMESTAMP NOT NULL DEFAULT NOW()

);

**-- 3. Products**

CREATE TABLE product (

id SERIAL PRIMARY KEY,

company\_id INTEGER NOT NULL REFERENCES company(id) ON DELETE CASCADE,

name VARCHAR(255) NOT NULL,

sku VARCHAR(64) NOT NULL,

price DECIMAL(12, 2) NOT NULL,

is\_bundle BOOLEAN NOT NULL DEFAULT FALSE,

-- Additional fields: description, etc.

UNIQUE (company\_id, sku) -- SKU unique within company

);

**-- 4. Bundled Products (For bundles containing other products)**

CREATE TABLE product\_bundle (

bundle\_id INTEGER NOT NULL REFERENCES product(id) ON DELETE CASCADE,

product\_id INTEGER NOT NULL REFERENCES product(id) ON DELETE CASCADE,

quantity INTEGER NOT NULL DEFAULT 1,

PRIMARY KEY (bundle\_id, product\_id)

);

**-- 5. Suppliers**

CREATE TABLE supplier (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

contact\_info VARCHAR(255),

-- Add address, phone, etc.

created\_at TIMESTAMP NOT NULL DEFAULT NOW()

);

**-- 6. Product - Supplier mapping (Products can have one or more** suppliers)

CREATE TABLE product\_supplier (

product\_id INTEGER NOT NULL REFERENCES product(id) ON DELETE CASCADE,

supplier\_id INTEGER NOT NULL REFERENCES supplier(id) ON DELETE CASCADE,

PRIMARY KEY (product\_id, supplier\_id)

);

**-- 7. Inventory (quantity per product per warehouse)**

CREATE TABLE inventory (

id SERIAL PRIMARY KEY,

product\_id INTEGER NOT NULL REFERENCES product(id) ON DELETE CASCADE,

warehouse\_id INTEGER NOT NULL REFERENCES warehouse(id) ON DELETE CASCADE,

quantity INTEGER NOT NULL DEFAULT 0,

UNIQUE (product\_id, warehouse\_id)

);

**-- 8. Inventory Changes (Audit/history log)**

CREATE TABLE inventory\_change (

id SERIAL PRIMARY KEY,

inventory\_id INTEGER NOT NULL REFERENCES inventory(id) ON DELETE CASCADE,

change\_amount INTEGER NOT NULL,

reason VARCHAR(255),

changed\_at TIMESTAMP NOT NULL DEFAULT NOW(),

changed\_by INTEGER -- Optionally FK to user table

);

1. **Identify Gaps / Product Questions**

* Will we implement user access control to define which users can view or manage specific company data?
* How should we model suppliers? Can a single supplier provide products to multiple companies, or should supplier relationships be limited to one company? In other words, are suppliers global entities or company-specific?
* Regarding product bundles, can a bundle include other bundles (creating a nested structure), or are bundles composed solely of individual products?
* How will inventory for bundles be handled? Will bundles have their own distinct inventory, or will their availability be derived from the inventory of their constituent components?
* What types of inventory transactions do we anticipate tracking? (e.g., sales, returns, damages, restocks, transfers).
* What is the desired scope of SKU uniqueness? Should SKUs be unique across the entire system globally, within each company, or per warehouse?Any difference between warehouse types? (online, physical, drop-ship, etc.)
* Do we need to differentiate between various warehouse types (e.g., online fulfilment centres, physical retail stores, drop-ship locations)? If so, what are the key distinctions?
* Should we store specific warehouse attributes such as capacity, physical address, or other relevant details?
* Is it necessary to track product pricing on a per-supplier basis?

1. **Explain Decisions:**

**Indexes and Constraints –**

**a) Indexes and Constraints:**

**Unique (company\_id, sku):**

* Keeps SKUs unique within a company—prevents duplicates but allows same SKU in different companies (fits multi-tenancy).

**Foreign Keys:**

* Ensure no orphaned warehouses, products, inventory, etc.

**Unique (product\_id, warehouse\_id) for Inventory:**

* Guarantees one inventory record per product/location.

**Primary Keys/Composite Keys in Mapping Tables:**

* Supports fast JOIN operations and avoids duplicates.

**b) Audit Table (inventory\_change):**

* Tracks inventory adjustments over time for history and troubleshooting.

**c) Bundles (product\_bundle):**

* Supports "products as bundles" with flexible contents; easy to navigate for inventory checks or UI displays.

**d) Scalability:**

* Indexed lookup on SKU, company/product IDs.
* Avoids denormalization; easy to extend (add more product attributes or relationships later).

**e) Flexibility:**

* Schema supports per-company isolation (key for B2B SaaS/multi-tenant).
* Bundles and product-supplier associations are generic; can accommodate changing requirements without major refactoring.

**PART 3: API Implementation**

**Assumptions & Schema Extensions**

**To implement the endpoint, I assume**:

* There is a table sales with columns:
* id, product\_id, warehouse\_id, quantity, sale\_date.
* Product types are implied by a product\_type field in product table (e.g. "standard", "bundle").
* Low-stock thresholds are stored in a table product\_type\_threshold (product\_type → threshold stock level).
* supplier table has a contact\_email column.
* Only sales within the last 30 days count as recent sales activity.
* Days until stockout calculated as:
* current\_stock / average\_daily\_sales for that product across the company (all warehouses).
* If average\_daily\_sales = 0, set days\_until\_stockout = null or very high

**API CODE -**

from flask import Flask, jsonify

from flask\_sqlalchemy import SQLAlchemy

from datetime import datetime, timedelta

from sqlalchemy import func, Boolean

app = Flask(\_\_name\_\_)

# Replace this with your actual NeonDB connection string

app.config['SQLALCHEMY\_DATABASE\_URI'] = (

"postgresql://neondb\_owner:npg\_oAji9kbcJa5L@"

"ep-cool-cherry-aeklwh08-pooler.c-2.us-east-2.aws.neon.tech/neondb"

"?sslmode=require&channel\_binding=require"

)

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

####################################################

# Database models (reflect your schema exactly)

####################################################

class Company(db.Model):

\_\_tablename\_\_ = 'company'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(255), nullable=False)

created\_at = db.Column(db.DateTime, nullable=False, server\_default=func.now())

class Warehouse(db.Model):

\_\_tablename\_\_ = 'warehouse'

id = db.Column(db.Integer, primary\_key=True)

company\_id = db.Column(db.Integer, db.ForeignKey('company.id', ondelete='CASCADE'), nullable=False)

name = db.Column(db.String(255), nullable=False)

location = db.Column(db.String(255))

created\_at = db.Column(db.DateTime, nullable=False, server\_default=func.now())

class Product(db.Model):

\_\_tablename\_\_ = 'product'

id = db.Column(db.Integer, primary\_key=True)

company\_id = db.Column(db.Integer, db.ForeignKey('company.id', ondelete='CASCADE'), nullable=False)

name = db.Column(db.String(255), nullable=False)

sku = db.Column(db.String(64), nullable=False)

price = db.Column(db.Numeric(12, 2), nullable=False)

is\_bundle = db.Column(db.Boolean, nullable=False, default=False) # use this for threshold join

class Supplier(db.Model):

\_\_tablename\_\_ = 'supplier'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(255), nullable=False)

contact\_info = db.Column(db.String(255)) # used as contact\_email in response

created\_at = db.Column(db.DateTime, nullable=False, server\_default=func.now())

class ProductSupplier(db.Model):

\_\_tablename\_\_ = 'product\_supplier'

product\_id = db.Column(db.Integer, db.ForeignKey('product.id', ondelete='CASCADE'), primary\_key=True)

supplier\_id = db.Column(db.Integer, db.ForeignKey('supplier.id', ondelete='CASCADE'), primary\_key=True)

class Inventory(db.Model):

\_\_tablename\_\_ = 'inventory'

id = db.Column(db.Integer, primary\_key=True)

product\_id = db.Column(db.Integer, db.ForeignKey('product.id', ondelete='CASCADE'), nullable=False)

warehouse\_id = db.Column(db.Integer, db.ForeignKey('warehouse.id', ondelete='CASCADE'), nullable=False)

quantity = db.Column(db.Integer, nullable=False, default=0)

class Sale(db.Model):

\_\_tablename\_\_ = 'sales'

id = db.Column(db.Integer, primary\_key=True)

product\_id = db.Column(db.Integer, db.ForeignKey('product.id'), nullable=False)

warehouse\_id = db.Column(db.Integer, db.ForeignKey('warehouse.id'), nullable=False)

quantity = db.Column(db.Integer, nullable=False)

sale\_date = db.Column(db.DateTime, nullable=False)

class ProductTypeThreshold(db.Model):

\_\_tablename\_\_ = 'product\_type\_threshold'

product\_type = db.Column(Boolean, primary\_key=True) # BOOLEAN: True for bundles, False for others

threshold = db.Column(db.Integer, nullable=False)

####################################################

# Low-stock alerts endpoint

####################################################

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

def get\_low\_stock\_alerts(company\_id):

# Verify company exists

company = Company.query.get(company\_id)

if not company:

return jsonify({"error": "Company not found"}), 404

# Define recent sales window (last 30 days)

recent\_period\_start = datetime.utcnow() - timedelta(days=30)

# 1. Products with recent sales in company

recent\_sales\_subq = (

db.session.query(Sale.product\_id)

.join(Product, Product.id == Sale.product\_id)

.filter(

Product.company\_id == company\_id,

Sale.sale\_date >= recent\_period\_start

)

.distinct()

.subquery()

)

# 2. Inventories with product info, warehouse info, and thresholds

inventories = (

db.session.query(

Inventory.id.label("inventory\_id"),

Product.id.label("product\_id"),

Product.name.label("product\_name"),

Product.sku,

Product.is\_bundle,

Warehouse.id.label("warehouse\_id"),

Warehouse.name.label("warehouse\_name"),

Inventory.quantity.label("current\_stock"),

ProductTypeThreshold.threshold

)

.join(Product, Inventory.product\_id == Product.id)

.join(Warehouse, Inventory.warehouse\_id == Warehouse.id)

.join(ProductTypeThreshold, Product.is\_bundle == ProductTypeThreshold.product\_type) # Boolean join here

.filter(

Product.company\_id == company\_id,

Warehouse.company\_id == company\_id,

Inventory.quantity <= ProductTypeThreshold.threshold,

Inventory.product\_id.in\_(recent\_sales\_subq) # Only products with recent sales

)

.all()

)

# 3. Calculate average daily sales per product (last 30 days)

sales\_agg = (

db.session.query(

Sale.product\_id,

func.sum(Sale.quantity).label("total\_quantity")

)

.join(Product, Product.id == Sale.product\_id)

.filter(

Product.company\_id == company\_id,

Sale.sale\_date >= recent\_period\_start

)

.group\_by(Sale.product\_id)

.all()

)

avg\_daily\_sales\_map = {s.product\_id: s.total\_quantity / 30 for s in sales\_agg}

# 4. Prepare alert list

alerts = []

for inv in inventories:

avg\_daily\_sales = avg\_daily\_sales\_map.get(inv.product\_id, 0)

if avg\_daily\_sales > 0:

days\_until\_stockout = int(inv.current\_stock / avg\_daily\_sales)

else:

days\_until\_stockout = None

# Fetch first linked supplier info (optional)

supplier = (

db.session.query(Supplier)

.join(ProductSupplier, Supplier.id == ProductSupplier.supplier\_id)

.filter(ProductSupplier.product\_id == inv.product\_id)

.first()

)

supplier\_info = None

if supplier:

supplier\_info = {

"id": supplier.id,

"name": supplier.name,

"contact\_email": supplier.contact\_info or ""

}

alert = {

"product\_id": inv.product\_id,

"product\_name": inv.product\_name,

"sku": inv.sku,

"warehouse\_id": inv.warehouse\_id,

"warehouse\_name": inv.warehouse\_name,

"current\_stock": inv.current\_stock,

"threshold": inv.threshold,

"days\_until\_stockout": days\_until\_stockout,

"supplier": supplier\_info

}

alerts.append(alert)

return jsonify({

"alerts": alerts,

"total\_alerts": len(alerts)

}), 200

####################################################

# Main entrypoint

####################################################

if \_\_name\_\_ == '\_\_main\_\_':

# For production, use a WSGI server like gunicorn instead

app.run(debug=True)

# Error Handling Note:

# - We validate company presence.

# - Filtering on recent sales ensures data relevance.

# - If no inventory found, returns empty alerts list.

# - Robust to missing supplier (shows null supplier).

# - Could extend with pagination if alerts list is large.

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

## Explanation of Approach

1. **Validate Company**: Ensure the requested company exists, otherwise return 404.
2. **Recent Sales Filter**:  
   Query recent sales (last 30 days) to get products with sales activity. This filters out products without recent demand to avoid unnecessary alerts.
3. **Inventory & Threshold Filtering**:  
   Join inventory with products, warehouses, and product-type thresholds, filter for inventory levels below or equal to thresholds.
4. **Average Daily Sales Calculation**:  
   Calculate average daily sales per product over 30 days. This helps estimate how many days before stock runs out.
5. **Supplier Information**:  
   Fetch supplier details linked to each product for reordering contact info.
6. **Return Structured JSON Alert List**:  
   Matches the expected response format with detailed info.

## Edge Cases Considered

* Company id invalid → returns 404.
* Products with no recent sales → excluded.
* Inventory quantity exactly equal to threshold → included.
* Products without any supplier → supplier info is null.
* No inventory or no low stock products → returns empty alerts list.
* Division by zero avoided by checking average daily sales > 0.
* Multi-warehouse handled by joining inventories per warehouse.

## General Reasoning & Approach

* **Problem-solving:** Systematically identified input, transactional, and business logic issues; prioritized data integrity and maintainability.
* **Database:** Emphasized scalable, normalized design with attention to multi-tenancy and auditability.
* **API:** Focused on robust filtering, clear business rules, and comprehensive error handling.
* **Communication:** Asked clarifying questions to expose requirement gaps, demonstrating anticipation of real-world complexities.
* **Maintaining best practices:** Transactions, validation, appropriate HTTP status codes, modular design.