**Part 1: Code Review and Debugging**

**Previous Given Code:**

@app.route('/api/products', methods=['POST'])

def create\_product():

data = request.json

# Create new product

product = Product(

name=data['name'],

sku=data['sku'],

price=data['price'],

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

)

db.session.add(product)

db.session.commit()

# Update inventory count

inventory = Inventory(

product\_id=product.id,

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

quantity=data['initial\_quantity']

)

db.session.add(inventory)

db.session.commit()

return {"message": "Product created", "product\_id": product.id}

1. **Issues**
2. As mentioned in the additional context, SKUs must be unique across the platform but the above given code doesn’t check that if the same SKU already exists for another product.
3. Again as per the requirement a product can exists in multiple warehouses but in the above code, the Product model directly has a warehouse\_id.
4. No proper validation for the quantity(non – negative integer) and also not checked if initial\_quantity is provided.
5. Also two separate db commits are given which may cause data inconsistency if one of them fails. If second fails, you get a product without inventory.
6. Price is not validated properly as it is taken directly from JSON, it can be a string or integer and could cause type issues.
7. Requirement also says some fields might be optional but the code given assumes all are present.
8. No response validation provided will always return a success response, even if something do not work or fail.
9. **Impact**
10. No proper tracking or logging of inventory due to duplicate SKUs.
11. Risk of system getting corrupted due to unvalidated inputs.
12. Records might get lost or remain unreferenced due to separate commits.
13. No error handling and no response validation can lead to hard for clients to detect failures and poor reliability in production.
14. Bad user experience as API will fail if the optional field are ignored by the user.
15. **Fix ( My solution)**

@app.route('/api/products', methods=['POST'])

def create\_product():

data = request.get\_json()

# Validate required fields

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

missing\_fields = [field for field in required\_fields if field not in data]

if missing\_fields:

return {"error": f"Missing fields: {', '.join(missing\_fields)}"}, 400

# Ensure SKU is unique

existing\_product = Product.query.filter\_by(sku=data['sku']).first()

if existing\_product:

return {"error": "SKU must be unique"}, 409

try:

# Create new product (not tied to one warehouse directly)

product = Product(

name=data['name'],

sku=data['sku'],

price=float(data['price']) # ensure decimal conversion

)

db.session.add(product)

db.session.flush() # get product.id without committing yet

# Create inventory entry

inventory = Inventory(

product\_id=product.id,

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

quantity=max(0, int(data['initial\_quantity']))

)

db.session.add(inventory)

db.session.commit() # commit both in one transaction

return {

"message": "Successful Product Creation",

"product\_id": product.id

}, 201

except Exception as e:

db.session.rollback()

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

Why this fix :

1. Checks for missing fields
2. SKU Uniqueness
3. Commits only once for data atomicity
4. Rollback if any error occurs, avoiding partial updates
5. Validation and type conversion to prevent invalid values
6. Clear API responses and separated product from warehouse to align with multi-warehouse requirement.

**Part 2: Database Design**

**1.Design Schema**

1. Entities and Relationships

* Companies with multiple warehouses
* Products across multiple warehouses
* Inventory tracking over time
* Suppliers providing products
* Product bundles

1. Schema (SQL DDL):

-- Companies table

CREATE TABLE companies (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL

);

-- Warehouses belong to companies

CREATE TABLE warehouses (

id SERIAL PRIMARY KEY,

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

name VARCHAR(255) NOT NULL

);

-- Products table

CREATE TABLE products (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

sku VARCHAR(100) UNIQUE NOT NULL,

price DECIMAL(10,2) NOT NULL,

is\_bundle BOOLEAN DEFAULT FALSE

);

-- Inventory per warehouse

CREATE TABLE inventory (

id SERIAL PRIMARY KEY,

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

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

quantity INT NOT NULL,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(product\_id, warehouse\_id)

);

-- Inventory change history

CREATE TABLE inventory\_log (

id SERIAL PRIMARY KEY,

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

change\_amount INT NOT NULL,

change\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Suppliers provide products to companies

CREATE TABLE suppliers (

id SERIAL PRIMARY KEY,

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

name VARCHAR(255) NOT NULL

);

-- Link suppliers to products

CREATE TABLE product\_suppliers (

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

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

PRIMARY KEY (product\_id, supplier\_id)

);

-- Bundles: products containing other products

CREATE TABLE product\_bundles (

bundle\_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,

component\_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,

quantity INT NOT NULL,

PRIMARY KEY (bundle\_id, component\_id)

);

**2. Gaps**

* SKU scope: Should SKUs be unique globally across all companies, or only within a company? (I assumed global uniqueness)
* Bundles: Do bundles have their own SKUs? Should a bundle inventory automatically update when its component inventories changes, or is it stored separately?
* Suppliers: Can a product have multiple suppliers? (I assumed yes)
* Inventory History: Track who made the changes or just the change itself? How returns, damaged and expired products be handled I inventory?
* Warehouse: How do we need to cover the product movement at warehouse level of a company?

1. **Why this design choice**

Minimal Schema : only the requirements given, but still scalable for future additions.

1. Companies and Warehouses

Foreign key: ensure every warehouse belongs to valid company

ON DELET CASCADE: Remove the warehouse if its parent company is removed to avoid lost or unreferenced records.

1. Products

SKU as UNIQUE: will ensure each product can be uniquely identified across the system.

is\_bundle: to differentiate between the bundle from regular product.

1. Inventory

Composite unique constraints (product id and warehouse id): prevents duplicate entries for the same product in the same warehouse.

Foreign key: ensure only valid products are stored in valid warehouse

TIMESTAMP: allows us to know when the stock was last updated

1. Inventory logs

Separate table for logs to keep the main inventory table efficient while allowing undefinite history tracking.

change\_time: allows chronological tracking of stock changes for reporting and surveys.

1. Suppliers and Products

Many to Many relation-ship between product and supplier as a supplier can have supply multiple products or a product can have multiple suppliers.

Composite Primary Key (product id and supplier id): to avoid duplicate relationships.

1. Product Bundles

Self-referencing many to many relation-ship: a product can be a bundle containing other products

Quantity: ensure the system knows how many units of a component are included in the bundle.

**Part 3: Low-Stock Alerts API**

1. **My API Implementation (Flask)**

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

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

try:

# Step 1: Fetch all products and their stock per warehouse

query = db.session.execute("""

SELECT p.id AS product\_id, p.name AS product\_name, p.sku,

w.id AS warehouse\_id, w.name AS warehouse\_name,

i.quantity AS current\_stock,

p.low\_stock\_threshold AS threshold,

s.id AS supplier\_id, s.name AS supplier\_name, s.contact\_email

FROM inventory i

JOIN products p ON i.product\_id = p.id

JOIN warehouses w ON i.warehouse\_id = w.id

LEFT JOIN product\_suppliers ps ON p.id = ps.product\_id

LEFT JOIN suppliers s ON ps.supplier\_id = s.id

WHERE w.company\_id = :company\_id

AND i.quantity < p.low\_stock\_threshold

""", {"company\_id": company\_id})

alerts = []

for row in query:

# Step 2: Check recent sales activity

sales = db.session.execute("""

SELECT COALESCE(AVG(sq.daily\_sales), 0) AS avg\_daily\_sales

FROM (

SELECT DATE(s.sale\_date) AS day, SUM(s.quantity) AS daily\_sales

FROM sales\_log s

WHERE s.product\_id = :pid

AND s.warehouse\_id = :wid

AND s.sale\_date >= NOW() - INTERVAL '30 days'

GROUP BY day

) sq

""", {"pid": row.product\_id, "wid": row.warehouse\_id}).first()

avg\_daily\_sales = sales.avg\_daily\_sales or 0

if avg\_daily\_sales == 0:

continue # Skip products with no recent sales

# Step 3: Calculate days until stockout

days\_until\_stockout = (

row.current\_stock / avg\_daily\_sales if avg\_daily\_sales > 0 else None

)

alerts.append({

"product\_id": row.product\_id,

"product\_name": row.product\_name,

"sku": row.sku,

"warehouse\_id": row.warehouse\_id,

"warehouse\_name": row.warehouse\_name,

"current\_stock": row.current\_stock,

"threshold": row.threshold,

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

"supplier": {

"id": row.supplier\_id,

"name": row.supplier\_name,

"contact\_email": row.contact\_email

} if row.supplier\_id else None

})

return {"alerts": alerts, "total\_alerts": len(alerts)}, 200

except Exception as e:

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

1. **Assumptions made to implement the above code**
2. Threshold is stored per product.
3. Days until stockout = current stock / average daily sales (using sales logs).
4. If no supplier exists for a product, supplier filed will be null.
5. Database Tables needed:

* Products
* Warehouses
* Inventory
* Product and Suppliers
* Sales\_log

1. **Handling Edge cases**

* Return “supplier”: null if no suppliers linked.
* No recent sales will exclude the product from the alerts.
* If threshold is missing a default of fallback is set (e.g., 10 units).
* Skipping products with no sales handle division by zero.

1. **Approach**

Started by joining the inventory, products and warehouses tables to get each product’s stock levels per warehouse for the given company and filtered for the cases where the stocks of a product are lower than the required threshold.

I check the recent sales activity from a sales\_log table, to calculate the average daily sales over the last 30 days, which allows me to compute the days until the stockout of the particular product.

At last, I structured the response to match the expected JSON format, handling edge cases like missing suppliers, thresholds and preventing division by zero.

This approach makes sure that the API gives company the actionable alert with enough information while staying aligned and efficient with the requirements.