**1) Code review and debugging**

**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.json or {}

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

for field in required\_fields:

if field not in data:

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

try:

product = Product(

name=data['name'],

sku=data['sku'],

price=Decimal(str(data['price'])) # ensure decimal

)

db.session.add(product)

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

inventory = Inventory(

product\_id=product.id,

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

quantity=data['initial\_quantity']

)

db.session.add(inventory)

db.session.commit()

except IntegrityError:

db.session.rollback()

return jsonify({"error": "SKU already exists"}), 400

except Exception as e:

db.session.rollback()

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

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

**Explanation -**

* **Removed warehouse\_id from Product** - products are independent of warehouses, inventory links them.
* **Added db.session.flush()** - gets the product ID before commit, so we can use it in inventory.
* **Single commit()** - prevent partial saves if one insert fails.
* **IntegrityError handling** - catches duplicate SKU issues.
* **Decimal type** - makes sure we can store fractional prices.
* **Validation loop** - avoids server errors when required data is missing.

**2) Database Design -**

*Company and Warehouses*

CREATE TABLE Company (

company\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL

);

CREATE TABLE Warehouse (

warehouse\_id SERIAL PRIMARY KEY,

company\_id INT NOT NULL,

name VARCHAR(100) NOT NULL,

FOREIGN KEY (company\_id) REFERENCES Company(company\_id)

);

*Suppliers and Products*

CREATE TABLE Supplier (

supplier\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL

);

CREATE TABLE Product (

product\_id SERIAL PRIMARY KEY,

supplier\_id INT,

name VARCHAR(100) NOT NULL,

is\_bundle BOOLEAN DEFAULT FALSE,

FOREIGN KEY (supplier\_id) REFERENCES Supplier(supplier\_id)

);

*Product bundles*

CREATE TABLE BundleItem (

bundle\_id INT NOT NULL,

product\_id INT NOT NULL,

quantity INT NOT NULL CHECK (quantity > 0),

PRIMARY KEY (bundle\_id, product\_id),

FOREIGN KEY (bundle\_id) REFERENCES Product(product\_id),

FOREIGN KEY (product\_id) REFERENCES Product(product\_id)

);

CREATE TABLE Inventory (

warehouse\_id INT NOT NULL,

product\_id INT NOT NULL,

quantity INT NOT NULL DEFAULT 0 CHECK (quantity >= 0),

PRIMARY KEY (warehouse\_id, product\_id),

FOREIGN KEY (warehouse\_id) REFERENCES Warehouse(warehouse\_id),

FOREIGN KEY (product\_id) REFERENCES Product(product\_id)

);

CREATE TABLE InventoryChange (

log\_id SERIAL PRIMARY KEY,

warehouse\_id INT NOT NULL,

product\_id INT NOT NULL,

change\_amount INT NOT NULL,

reason VARCHAR(100),

changed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (warehouse\_id) REFERENCES Warehouse(warehouse\_id),

FOREIGN KEY (product\_id) REFERENCES Product(product\_id)

);

**Justifications** -

### **Company**

* company\_id SERIAL PRIMARY KEY  
  - I used SERIAL so the database automatically assigns unique IDs. Company names might not be unique (two companies can have similar names), so a generated numeric ID is safer.

### **Warehouse**

* warehouse\_id SERIAL PRIMARY KEY  
  - Each warehouse needs its own unique identifier.
* company\_id INT FOREIGN KEY  
  - Warehouses belong to companies, so I made company\_id a foreign key. This makes sure that no warehouse exists without a parent company.

### **Supplier**

* supplier\_id SERIAL PRIMARY KEY  
  - Similar reasoning as company\_id. Suppliers might share names, so numeric IDs avoid conflicts.

### **Product**

* product\_id SERIAL PRIMARY KEY  
  - Products need a unique ID for easy referencing.
* supplier\_id INT FOREIGN KEY  
  - A product usually comes from a supplier. Making this a foreign key enforces valid supplier references. I allowed it to be nullable since the requirement didn’t say every product must have a supplier.
* is\_bundle BOOLEAN  
  - I added this so we can quickly check if a product is a bundle or a normal product.

### **BundleItem (junction table for bundles)**

* PRIMARY KEY (bundle\_id, product\_id)  
  - Composite primary key ensures the same product isn’t added twice to the same bundle.
* bundle\_id and product\_id are both foreign keys to Product  
   - This enforces that only valid products can be part of a bundle, and bundles themselves are also just products.

### **Inventory**

* PRIMARY KEY (warehouse\_id, product\_id)  
   - Each warehouse should only have one row per product. The composite key guarantees uniqueness of that pair.
* quantity INT CHECK (quantity >= 0)  
   - Prevents invalid data like negative inventory.

### **InventoryChange**

* log\_id SERIAL PRIMARY KEY  
  - Each change event needs a unique identifier for audit/history.
* warehouse\_id, product\_id as foreign keys  
   - A change must relate to an actual product and an actual warehouse.
* change\_amount INT  
   - Positive values for restocks, negative values for sales or removals.
* changed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
   - Automatically records when the change happened, so no manual input is needed.

**3) API Implementation -**

from flask import Flask, jsonify

from datetime import datetime, timedelta

from sqlalchemy import func

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

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

def low\_stock\_alerts(company\_id):

"""

Assumptions:

- Each product has a 'threshold' column that defines its low stock level.

- InventoryChange stores stock changes (positive for restock, negative for sale).

- "Recent sales" = sales recorded in the last 30 days.

"""

try:

alerts = []

thirty\_days\_ago = datetime.utcnow() - timedelta(days=30)

warehouses = Warehouse.query.filter\_by(company\_id=company\_id).all()

if not warehouses:

return jsonify({"alerts": [], "total\_alerts": 0}), 200

for warehouse in warehouses:

products = (

db.session.query(

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

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

Product.sku,

Product.threshold,

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

Supplier.id.label("supplier\_id"),

Supplier.name.label("supplier\_name"),

Supplier.contact\_info.label("supplier\_email")

)

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

.outerjoin(Supplier, Supplier.id == Product.supplier\_id)

.filter(Inventory.warehouse\_id == warehouse.id)

.all()

)

for row in products:

sales = (

db.session.query(func.sum(InventoryChange.change\_amount))

.filter(

InventoryChange.product\_id == row.product\_id,

InventoryChange.warehouse\_id == warehouse.id,

InventoryChange.reason == "sale",

InventoryChange.changed\_at >= thirty\_days\_ago

)

.scalar()

)

sales = abs(sales) if sales else 0

if sales == 0:

continue # skip products without recent sales

if row.current\_stock < row.threshold:

avg\_daily\_sales = sales / 30 if sales > 0 else 1

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": warehouse.id,

"warehouse\_name": warehouse.name,

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

"threshold": row.threshold,

"days\_until\_stockout": int(days\_until\_stockout) if days\_until\_stockout else None,

"supplier": {

"id": row.supplier\_id,

"name": row.supplier\_name,

"contact\_email": row.supplier\_email

}

})

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

except Exception as e:

# Rollback in case of query issues

db.session.rollback()

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

**Assumptions**

* Each product has a column threshold (different thresholds possible).
* Recent sales = last 30 days.
* Sales are logged in InventoryChange with reason="sale".
* A product always belongs to at least one warehouse, but might not have a supplier.

**Edge cases**

* Company has no warehouses - return empty alerts list.
* No recent sales - product skipped.
* Division by zero - handled by defaulting avg daily sales to 1.
* Missing supplier info - allowed with outerjoin (supplier fields can be null).
* Errors in DB queries - caught in try/except, rollback applied.