# PART 1: Code Review & Debugging (30 minutes)

**@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}**

**Identified Issues:**

* The code commits once after creating the product and again after creating the inventory record. If the second commit fails, the product is still inserted but without a matching inventory entry. This causes inconsistent data in production.
* The code assumes all required keys (name, sku, price, warehouse\_id, initial\_quantity) are present in the request. If any field is missing or invalid, it will throw a KeyError.
* If any exception occurs after the first commit, the product is already stored in the database, leaving orphaned records.
* Proper rollback handling is missing.
* The inventory record is always created with the product, but no check is done to ensure the warehouse exists or that the SKU is unique.
* Returning a plain dictionary works with Flask, but using jsonify is recommended for proper headers and consistency.

**Corrected Code:**

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

def create\_product():

data = request.get\_json()

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:

# Single transaction for product + inventory

product = Product(

name=data['name'],

sku=data['sku'],

price=data['price'],

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

)

db.session.add(product)

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

inventory = Inventory(

product\_id=product.id,

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

quantity=data['initial\_quantity']

)

db.session.add(inventory)

db.session.commit()

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

except Exception as e:

db.session.rollback()

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

# PART 2: Database Schema Design

-- Companies

CREATE TABLE Companies (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL UNIQUE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Warehouses

CREATE TABLE Warehouses (

id SERIAL PRIMARY KEY,

company\_id INT NOT NULL REFERENCES Companies(id),

name VARCHAR(255) NOT NULL,

location VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Suppliers

CREATE TABLE Suppliers (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL UNIQUE,

contact\_info TEXT

);

-- Products

CREATE TABLE Products (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

sku VARCHAR(100) NOT NULL UNIQUE,

price DECIMAL(10,2) NOT NULL,

is\_bundle BOOLEAN DEFAULT FALSE,

supplier\_id INT REFERENCES Suppliers(id),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- ProductBundles (many-to-many self reference for bundled products)

CREATE TABLE ProductBundles (

bundle\_id INT NOT NULL REFERENCES Products(id),

product\_id INT NOT NULL REFERENCES Products(id),

quantity INT NOT NULL DEFAULT 1,

PRIMARY KEY (bundle\_id, product\_id)

);

-- Inventory (current stock levels)

CREATE TABLE Inventory (

id SERIAL PRIMARY KEY,

product\_id INT NOT NULL REFERENCES Products(id),

warehouse\_id INT NOT NULL REFERENCES Warehouses(id),

quantity INT NOT NULL DEFAULT 0,

last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

UNIQUE (product\_id, warehouse\_id)

);

-- InventoryHistory (audit trail of stock changes)

CREATE TABLE InventoryHistory (

id SERIAL PRIMARY KEY,

product\_id INT NOT NULL REFERENCES Products(id),

warehouse\_id INT NOT NULL REFERENCES Warehouses(id),

change\_amount INT NOT NULL,

reason VARCHAR(100),

changed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

ERD-Style Description

Companies (1) ───< (N) Warehouses

Warehouses (1) ───< (N) Inventory >───(1) Products

Suppliers (1) ───< (N) Products

Products (N) ───< (N) ProductBundles (bundles can contain products)

Inventory (1) ───< (N) InventoryHistory (tracks changes)

Questions / Gaps in Requirements

* Can products have **multiple suppliers**, or only one?
* Should **price** be global, or vary per company/warehouse?
* Should **bundles** allow nested bundles (a bundle inside another bundle)?
* Do we need to track **who performed the inventory change** (user ID)?
* Should there be **soft deletes** (active flag) for products and warehouses instead of hard deletes?
* Should we explicitly model **transfers between warehouses**, or just record them as two stock changes?

Design Decisions & Justifications

* **Transactions**: Separate Inventory (current stock) and InventoryHistory (audit log) to maintain both real-time view and traceability.
* **Constraints**: Unique constraint on (product\_id, warehouse\_id) ensures no duplicate inventory rows.
* **Indexes**:  
  + Index on Products.sku for quick lookups.
  + Index on (product\_id, warehouse\_id) in Inventory for fast stock queries.
  + Index on InventoryHistory.changed\_at for chronological reports.
* **Extensibility**: is\_bundle flag + ProductBundles table allows flexible handling of bundles.
* **Normalization**: Keeps suppliers, products, warehouses independent to avoid redundant data.

# Part 3: API Implementation – Low Stock Alerts

Assumptions about Schema and Business Logic

* Each Product has a defined **low stock threshold** (threshold column assumed in Products table).
* Recent sales activity is logged in a Sales or Orders table (assume we only consider products sold in the past 30 days).
* Stock levels are stored in Inventory (linked to both Products and Warehouses).
* Suppliers are linked to products via Products.supplier\_id.
* Calculation for days\_until\_stockout = current\_stock / avg\_daily\_sales. (If sales data unavailable, set this to null).

Python + Flask Implementation

from flask import Flask, jsonify, request

from sqlalchemy import func

from models import db, Product, Warehouse, Inventory, Supplier, Sales # assumed ORM models

import datetime

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

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

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

"""

Returns low-stock alerts for a company across all its warehouses.

Business Rules:

- Only products below their defined threshold are included.

- Only consider products with recent sales (last 30 days).

- Include supplier info for reordering.

"""

# Define recent sales cutoff

cutoff\_date = datetime.datetime.utcnow() - datetime.timedelta(days=30)

# Query for products with recent sales activity

recent\_sales = (

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

.filter(Sales.company\_id == company\_id, Sales.date >= cutoff\_date)

.distinct()

.subquery()

)

# Join Inventory, Products, Warehouses, Suppliers

results = (

db.session.query(

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

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

Product.sku,

Product.threshold,

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

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

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

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

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

Supplier.contact\_info.label("supplier\_contact"),

)

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

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

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

.filter(Warehouse.company\_id == company\_id)

.filter(Product.id.in\_(recent\_sales)) # only products with recent sales

.filter(Inventory.quantity < Product.threshold) # low stock check

.all()

)

alerts = []

for row in results:

# Estimate days until stockout (naive: current stock / avg daily sales)

avg\_sales = (

db.session.query(func.avg(Sales.quantity))

.filter(Sales.product\_id == row.product\_id, Sales.date >= cutoff\_date)

.scalar()

)

days\_until\_stockout = None

if avg\_sales and avg\_sales > 0:

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

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.supplier\_contact

}

})

return jsonify({

"alerts": alerts,

"total\_alerts": len(alerts)

})

Edge Cases Considered

1. **No recent sales** → Products without sales in last 30 days are excluded.
2. **Zero sales volume** → If average sales = 0, days\_until\_stockout is set to null.
3. **Missing supplier info** → If supplier data is missing, either exclude from results or return null fields.
4. **Multiple warehouses** → Each product may appear multiple times if stock is low in multiple warehouses.
5. **Concurrent updates** → If stock changes while fetching, results may be slightly stale, but still valid for alerts.

Approach Explained

* **Filtering by recent sales** ensures we don’t spam alerts for products that aren’t moving.
* **Threshold comparison** (Inventory.quantity < Product.threshold) drives the core logic.
* **Supplier join** provides reorder information in the same response.
* **Days until stockout** gives a predictive signal to prioritize urgent reorders.
* **Separation of concerns**: Inventory shows current levels, while Sales provides consumption rate.