### Backend Intern Case Study Solution

### Issues Identified

| **Type** | **Problem** | | |
| --- | --- | --- | --- |
| Technical | | price could be float/decimal — potential precision issues |
| Technical | | No validation for missing fields (e.g., sku, price, quantity) |
| Technical | | No error handling (try-except) — one failure crashes whole flow |
| Business | | SKU should be **unique** — no uniqueness check |
| Business | | Product tied to a single warehouse — violates multi-warehouse logic |
| Business | | initial\_quantity field used without verifying its presence |

**Impact in Production**

* **Duplicate SKUs**: Inventory chaos due to multiple entries with the same SKU.
* **Missing fields**: Crashes API with 500 errors.
* **No rollback on failure**: Partial data (product saved but not inventory).
* **Wrong modeling**: Product can't be associated with multiple warehouses in future.

**Part 1: Fixed Code with Explaination**

from flask import request, jsonify

from sqlalchemy.exc import IntegrityError

from decimal import Decimal, InvalidOperation

import re

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

def create\_product():

try:

# Input validation

data = request.get\_json()

if not data:

return jsonify({"error": "No JSON data provided"}), 400

# Validate required fields

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

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

if missing\_fields:

return jsonify({

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

}), 400

# Validate data types and business rules

errors = []

# Name validation

if not isinstance(data['name'], str) or len(data['name'].strip()) == 0:

errors.append("Product name must be a non-empty string")

# SKU validation (alphanumeric, dashes, max 50 chars)

sku = data['sku'].strip()

if not re.match(r'^[A-Za-z0-9-\_]+$', sku) or len(sku) > 50:

errors.append("SKU must be alphanumeric with dashes/underscores, max 50 characters")

# Price validation

try:

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

if price < 0:

errors.append("Price cannot be negative")

except (InvalidOperation, ValueError):

errors.append("Price must be a valid decimal number")

# Warehouse validation

if not isinstance(data['warehouse\_id'], int) or data['warehouse\_id'] <= 0:

errors.append("Warehouse ID must be a positive integer")

# Initial quantity validation (optional field)

initial\_quantity = data.get('initial\_quantity', 0)

if not isinstance(initial\_quantity, (int, float)) or initial\_quantity < 0:

errors.append("Initial quantity must be a non-negative number")

if errors:

return jsonify({"errors": errors}), 400

# Check if warehouse exists

warehouse = Warehouse.query.get(data['warehouse\_id'])

if not warehouse:

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

# Check SKU uniqueness

existing\_product = Product.query.filter\_by(sku=sku).first()

if existing\_product:

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

# Start database transaction

try:

# Create new product

product = Product(

name=data['name'].strip(),

sku=sku,

price=price,

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

)

db.session.add(product)

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

# Create inventory record if initial quantity provided

if initial\_quantity > 0:

inventory = Inventory(

product\_id=product.id,

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

quantity=initial\_quantity

)

db.session.add(inventory)

# Commit both operations together

db.session.commit()

return jsonify({

"message": "Product created successfully",

"product": {

"id": product.id,

"name": product.name,

"sku": product.sku,

"price": str(product.price)

}

}), 201

except IntegrityError as e:

db.session.rollback()

return jsonify({"error": "Database constraint violation"}), 409

except Exception as e:

db.session.rollback()

return jsonify({"error": "Internal server error"}), 500

except Exception as e:

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

**Part 2: Database Design**

**Schema Design (SQL DDL Style)**

CREATE TABLE companies (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL

);

CREATE TABLE warehouses (

id SERIAL PRIMARY KEY,

company\_id INT REFERENCES companies(id),

name VARCHAR(255) NOT NULL

);

CREATE TABLE suppliers (

id SERIAL PRIMARY KEY,

name VARCHAR(255),

contact\_email VARCHAR(255)

);

CREATE TABLE products (

id SERIAL PRIMARY KEY,

name VARCHAR(255),

sku VARCHAR(50) UNIQUE,

price DECIMAL(10, 2),

supplier\_id INT REFERENCES suppliers(id),

is\_bundle BOOLEAN DEFAULT FALSE

);

CREATE TABLE product\_warehouse\_inventory (

id SERIAL PRIMARY KEY,

product\_id INT REFERENCES products(id),

warehouse\_id INT REFERENCES warehouses(id),

quantity INT NOT NULL,

last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE inventory\_changes (

id SERIAL PRIMARY KEY,

product\_id INT REFERENCES products(id),

warehouse\_id INT REFERENCES warehouses(id),

change\_type VARCHAR(50), -- e.g. 'sale', 'restock'

quantity\_change INT,

changed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- For bundled products

CREATE TABLE product\_bundles (

bundle\_id INT REFERENCES products(id),

component\_id INT REFERENCES products(id),

quantity INT,

PRIMARY KEY (bundle\_id, component\_id)

);

**Questions to Ask Product Team**

1. What defines "recent" sales activity (last 7 days? 30 days?)?
2. Do bundled products affect inventory of components?
3. Are there different thresholds for different warehouse locations?
4. Should inventory changes be reversible (e.g. returns)?
5. Should warehouse transfers be tracked?

**Design Choices**

* Used product\_warehouse\_inventory to support multiple warehouses.
* Indexed sku, product\_id, warehouse\_id for fast lookups.
* inventory\_changes logs help calculate sales trends and audit.
* Separate product\_bundles table models bundles without duplication.

**Part 3: API Implementation**

**Endpoint**

GET /api/companies/{company\_id}/alerts/low-stock

**Implementation in Flask**

from flask import jsonify

from sqlalchemy import and\_, func

from datetime import datetime, timedelta

from decimal import Decimal

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

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

"""

Get low stock alerts for a company across all warehouses.

Only includes products with recent sales activity.

"""

try:

# Verify company exists

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

if not company:

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

# Date range for "recent" sales activity (last 30 days)

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

# Query for products with low stock and recent sales

alerts\_query = db.session.query(

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

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

Product.sku,

Product.low\_stock\_threshold.label('threshold'),

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

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

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

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

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

Supplier.contact\_email.label('supplier\_email'),

func.avg(Sales.quantity\_sold).label('avg\_daily\_sales')

).select\_from(

Product

).join(

Inventory, Product.id == Inventory.product\_id

).join(

Warehouse, Inventory.warehouse\_id == Warehouse.id

).join(

Sales, and\_(

Sales.product\_id == Product.id,

Sales.warehouse\_id == Warehouse.id,

Sales.sale\_date >= thirty\_days\_ago

)

).outerjoin(

ProductSupplier, and\_(

ProductSupplier.product\_id == Product.id,

ProductSupplier.is\_primary == True

)

).outerjoin(

Supplier, ProductSupplier.supplier\_id == Supplier.id

).filter(

and\_(

Product.company\_id == company\_id,

Product.is\_active == True,

Inventory.quantity > 0, # Don't alert for already empty stock

Inventory.quantity <= Product.low\_stock\_threshold

)

).group\_by(

Product.id, Product.name, Product.sku, Product.low\_stock\_threshold,

Warehouse.id, Warehouse.name, Inventory.quantity,

Supplier.id, Supplier.name, Supplier.contact\_email

).having(

func.count(Sales.id) > 0 # Only products with recent sales

)

results = alerts\_query.all()

alerts = []

for row in results:

# Calculate days until stockout based on average daily sales

avg\_daily\_sales = float(row.avg\_daily\_sales) if row.avg\_daily\_sales else 0

days\_until\_stockout = None

if avg\_daily\_sales > 0:

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

alert = {

"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\_email

} if row.supplier\_id else None

}

alerts.append(alert)

# Sort by urgency (days until stockout, then by current stock level)

alerts.sort(key=lambda x: (

x["days\_until\_stockout"] if x["days\_until\_stockout"] is not None else 999,

x["current\_stock"]

))

return jsonify({

"alerts": alerts,

"total\_alerts": len(alerts)

})

except Exception as e:

# Log the error in production

app.logger.error(f"Error generating low stock alerts for company {company\_id}: {str(e)}")

return jsonify({"error": "Internal server error"}), 500

**Edge Cases Handled**

1. **Company Not Found**: Return 404 with appropriate message
2. **No Recent Sales**: Only products with sales activity in last 30 days
3. **Division by Zero**: Handle cases where average daily sales is 0
4. **Missing Supplier**: Handle products without primary suppliers
5. **Database Errors**: Proper error handling and logging
6. **Empty Results**: Return empty array with total\_alerts: 0
7. **Performance**: Optimized query version for large datasets

**API Design Decisions**

* **RESTful URL Structure**: Clear resource hierarchy
* **Proper HTTP Status Codes**: 200 for success, 404 for not found, 500 for errors
* **Consistent Response Format**: Structured JSON with metadata
* **Error Handling**: Comprehensive error catching with appropriate responses
* **Performance Considerations**: Optimized queries and indexing strategy
* **Extensibility**: Easy to add filters, pagination, or sorting parameters