Case Study Response

Name: Rakesh Hokrani
Role: Backend Engineering Intern
Date: 29 Dec 2025

Part 1: Code Review & Debugging

Problem Overview

The given API endpoint is responsible for creating a new product and initializing its inventory. Although the code compiles, it has multiple technical and business-logic issues that can cause failures in production.

Issues Identified

1. No input validation (KeyError if field missing)
2. SKU uniqueness not checked
3. No transaction handling (partial commits possible)
4. Decimal price not handled safely
5. Assumes product exists in only one warehouse
6. No error handling / rollback
7. Business rule violation: inventory tied at creation
8. Initial quantity not validated (negative values)

Impact in Production

- Duplicate SKUs → data corruption
- Partial failures → product created without inventory
- App crashes on bad input
- Incorrect pricing due to float issues
- Cannot scale to multi-warehouse use

Fixed Code (Flask + SQLAlchemy)

```
from decimal import Decimal
from sqlalchemy.exc import IntegrityError

@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json()

    required = ['name', 'sku', 'price', 'warehouse_id']
    for field in required:
        if field not in data:
            return {"error": f"{field} is required"}, 400
```

```python
try:
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=Decimal(str(data['price']))
    )

    db.session.add(product)
    db.session.flush()  # get product.id without commit

    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=max(0, data.get('initial_quantity', 0))
    )

    db.session.add(inventory)
    db.session.commit()

    return {"message": "Product created", "product_id": product.id}, 201

except IntegrityError:
    db.session.rollback()
    return {"error": "SKU must be unique"}, 409

except Exception:
    db.session.rollback()
    return {"error": "Internal server error"}, 500
```

Explanation of Fix

The updated solution validates inputs, enforces SKU uniqueness, uses decimal-safe pricing, and wraps all database operations in a single atomic transaction to ensure data integrity.

Part 2: Database Design

Requirements Summary

The system must support multiple companies, warehouses, suppliers, bundled products, and inventory tracking across locations.

Proposed Schema (Tables list or SQL DDL)

- Company
- Warehouse
- Product
- Inventory
- InventoryLog
- Supplier
- ProductSupplier
- Bundle

Core Tables

```
Company(id, name)
Warehouse(id, company_id, name)

Product(id, name, sku UNIQUE, price, type)
Supplier(id, name, contact_email)

Inventory(id, product_id, warehouse_id, quantity)
InventoryLog(id, inventory_id, change, reason, created_at)

ProductSupplier(product_id, supplier_id)

Bundle(id, parent_product_id, child_product_id, quantity)
```

Key Design Decisions

- Inventory as junction table → supports multi-warehouse
- InventoryLog → audit & analytics
- Bundle self-referencing → nested products
- Index on (product_id, warehouse_id)
- SKU globally unique

Missing Questions for Product Team

- Can SKU vary per company?
- Can bundles contain bundles?

- Sales velocity calculation window?
- Multiple suppliers priority?
- Soft delete required?

Part 3: Low-Stock Alerts API

Endpoint

```
GET /api/companies/{company_id}/alerts/low-stock
```

Business Logic

- Low-stock threshold varies by product type
- Only products with recent sales activity are considered
- Inventory is checked across all warehouses
- Supplier details are included for reordering

Assumptions

- sales_last_30_days column exists
- Threshold stored per product type
- One primary supplier per product

Implementation (Flask)

```
@app.route('/api/companies/<int:company_id>/alerts/low-stock')
def low_stock_alerts(company_id):
    alerts = []

    inventories = db.session.query(Inventory)\
        .join(Product)\
        .join(Warehouse)\
        .filter(Warehouse.company_id == company_id)\
        .filter(Product.sales_last_30_days > 0)\
        .all()

    for inv in inventories:
        threshold = inv.product.low_stock_threshold
        if inv.quantity < threshold:
            days_left = inv.quantity // max(1, inv.product.avg_daily_sales)
        supplier = inv.product.suppliers[0] if inv.product.suppliers else None

        alerts.append({
            "product_id": inv.product.id,
            "product_name": inv.product.name,
            "sku": inv.product.sku,
            "warehouse_id": inv.warehouse.id,
            "warehouse_name": inv.warehouse.name,
            "current_stock": inv.quantity,
            "threshold": threshold,
            "days_until_stockout": days_left,
            "supplier": {
                "id": supplier.id,
                "name": supplier.name,
                "contact_email": supplier.contact_email
            } if supplier else None
        })

    return {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }
```

Edge Cases Handled

- No recent sales → ignored
- Zero sales → division safe
- Missing supplier → null-safe
- Multiple warehouses → supported

Explanation of every part in short

I approached this case study by first focusing on correctness and real-world behavior.

In Part 1, I reviewed the existing product-creation API and identified issues such as missing input validation, lack of SKU uniqueness checks, unsafe handling of decimal prices and absence of transaction management. These issues could cause duplicate products, partial database writes and production crashes. I fixed them by validating inputs, enforcing unique constraints, using proper decimal handling for price and wrapping database operations in a single safe transaction with rollback support to ensure data integrity.

In Part 2, I designed a scalable database schema that supports companies with multiple warehouses and products stored across them. I separated products, warehouses and inventory into distinct tables to handle many-to-many relationships, added an inventory log table to track stock changes over time and introduced junction tables for suppliers and bundled products. This design ensures flexibility, auditability and efficient querying while leaving room for future growth.

In Part 3, I implemented a low-stock alert API that aggregates inventory across all warehouses for a company, applies product-specific low-stock thresholds and filters alerts to only products with recent sales activity. The response includes supplier details to support reordering decisions. Throughout the solution, I documented assumptions where requirements were unclear and focused on clean API design, edge-case handling and maintainability.