StockFlow Technical Assessment Submission

By: Suraj Hanumant Takbhate

Part 1: Code Review & Debugging

My Observations:

When I looked at this code, I noticed some important issues that could cause problems in a production environment.

1. **The Commit Problem:** The code uses db. session. commit () twice. If the product is saved successfully but an error occurs while saving the inventory, it can lead to data inconsistency in the database. We need an “all or nothing” approach.
2. **Missing Validation:** The data received from request. Json is not validated. If a user does not send required fields like SKU or name, it can cause a KeyError in Python and the API may crash.
3. **SKU Uniqueness:** The business rule says that the SKU must be unique, but the code does not check for this.

My Fixed Version:

I have improved this code by using a try-except block and proper validation.

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

def create_product():

   # Getting the request data

   incoming_data = request.json


   # 1. Manually checking if required fields are present

   # (Student's practical approach to avoid KeyError)

   required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']

   for field in required_fields:

      if field not in incoming_data:

         return {"error": f"Opps! '{field}' is missing."}, 400


   try:

      # 2. Checking if SKU is already used

      check_sku = Product.query.filter_by(sku=incoming_data['sku']).first()

      if check_sku:
            return {"error": "This SKU is already taken, try a different one."}, 400

        # 3. Handling DB session in one go to keep data safe
        new_prod = Product(
            name=incoming_data['name'],
            sku=incoming_data['sku'],
            price=incoming_data['price']
        )
        db.session.add(new_prod)

        # Using flush here because I need the product.id for the next table
        db.session.flush()

        # Adding initial stock to the warehouse
        new_inv = Inventory(
            product_id=new_prod.id,
            warehouse_id=incoming_data['warehouse_id'],
            quantity=incoming_data['initial_quantity']
        )
        db.session.add(new_inv)

        # Now committing everything at once
        db.session.commit()

        return {"message": "Great! Product and stock added.", "id": new_prod.id}, 201

    except Exception as e:
        db.session.rollback() # Undo if something breaks
        return {"error": "Database error", "msg": str(e)}, 500
```

Part 2: Database Design

I tried to keep the database design **normalized** while creating it.

- **Warehouses:** It contains a `company id` so that we can identify which company the warehouse belongs to.

- **Inventory:** This table connects the **Product** and **Warehouse**. I added a **low_stock_threshold** column because each product can have a different minimum stock limit.

- **Bundles:** For this, I suggest a Bundle Items table where there will be a relationship using parent id and child id.

Questions I have for the Product Team:

- **How should bundle product stock be calculated?**
  (If any component runs out, should the bundle be shown as *Out of Stock*?)

- **When stock becomes low, should an email notification be sent?**

Part 3: API for Low Stock Alerts

Logic used: I have joined three tables here and used a simple formula:
Current Stock / Average Daily Sales, which helps us understand how many days the stock will last.

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

def get_low_stock_alerts(company_id):

  # Assumption: Every product has a 'low_stock_threshold' defined in Inventory table.


  low_stock_list = []


  # Query to get items where stock is less than or equal to threshold

  items = db.session.query(Inventory, Product, Warehouse, Supplier)\

    .join(Product, 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(Inventory.quantity <= Inventory.low_stock_threshold).all()

    for inv, prod, wh, sup in items:
        # A simple logic to skip items that are not selling (no activity)
        avg_sales = get_daily_sales_rate(prod.id)

        if avg_sales > 0:
            days_remaining = int(inv.quantity / avg_sales)

            low_stock_list.append({
                "product_name": prod.name,
                "sku": prod.sku,
                "warehouse": wh.name,
                "current_stock": inv.quantity,
                "threshold": inv.low_stock_threshold,
                "days_until_stockout": days_remaining,
                "supplier": {
                    "name": sup.name,
                    "email": sup.contact_email
                }
            })

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

# Placeholder function for sales logic
def get_daily_sales_rate(product_id):
    # I'm assuming we'll calculate this from the 'Orders' table for last 30 days
    return 1.5
```