StockFlow – Backend Engineering Intern Assessment

Part 1:

1. Code Review & Debugging

1. **Issues Identified**
   While reviewing the provided API endpoint for product creation, I identified the following technical and business logic issues that could cause failures or inconsistencies in a production environment:
2. **Missing Input Validation**
   The API directly accesses values from request.json without checking whether required fields are present. Missing fields will result in runtime errors.
3. **SKU Uniqueness Not Enforced**
   Business rules specify that SKUs must be unique across the platform, but the code does not validate this before inserting a new product.
4. **Incorrect Product–Warehouse Relationship**
   The product model contains warehouse_id, but products can exist in multiple warehouses. This relationship should be handled via the inventory table.
5. **Lack of Transaction Management**
   Product creation and inventory creation are committed separately. If one operation fails, the database can end up in an inconsistent state.
6. **No Error or Exception Handling**
   Any database or runtime error will crash the API and return a generic server error without useful feedback.
7. **Price Data Type Not Properly Handled**
   Price values are accepted directly without enforcing decimal precision, which can lead to calculation inaccuracies.
8. **Optional Fields Not Safely Handled**
   Fields like warehouse_id and initial_quantity are assumed to always exist, even though they may be optional.

2. Impact in Production

- Missing validation can cause frequent API crashes due to malformed requests.
- Duplicate SKUs can result in incorrect product identification and reporting issues.
- Incorrect warehouse modeling limits scalability and prevents multi-warehouse support.
- Separate commits can lead to orphaned records and inconsistent inventory data.
- Lack of error handling makes debugging difficult and degrades user experience.
- Improper price handling may cause financial calculation errors.
- Optional fields missing from requests can cause unexpected failures.

3. Explanation of Fixes

- Added input validation to prevent runtime crashes
- Enforced SKU uniqueness to follow business rules
- Removed warehouse dependency from the product model
- Used a single transaction to ensure atomicity
- Used Decimal for accurate price handling
- Added exception handling for stability
- Safely handled optional fields

4. Corrected Implementation

```
from flask import request, jsonify
from decimal import Decimal
from sqlalchemy.exc import IntegrityError

@app.route('/api/products', methods=['POST'])
def create_product():
    try:
        data = request.json
        # Validate required fields
        required_fields = ['name', 'sku', 'price']
        for field in required_fields:
            if field not in data:
                return {"error": f"{field} is required"}, 400
        # Enforce SKU uniqueness
        if Product.query.filter_by(sku=data['sku']).first():
            return {"error": "SKU already exists"}, 409
        # Create product without warehouse dependency
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=Decimal(data['price'])
        )
        db.session.add(product)
        db.session.flush()  # Get product ID before commit
        # Create inventory only if warehouse data is provided
        if 'warehouse_id' in data and 'initial_quantity' in data:
            inventory = Inventory(
                product_id=product.id,
                warehouse_id=data['warehouse_id'],
                quantity=data['initial_quantity']
            )
            db.session.add(inventory)
        # Single transaction commit
        db.session.commit()
        return {
            "message": "Product created successfully",
            "product_id": product.id
        }, 201
    except IntegrityError:
        db.session.rollback()
        return {"error": "Database constraint violation"}, 400
    except Exception:
        db.session.rollback()
        return {"error": "Internal server error"}, 500
```

Part 2: Database Design

The goal of the database design is to support:

- Multiple companies
- Multiple warehouses per company
- Products stored in many warehouses
- Inventory tracking with history
- Supplier relationships
- Product bundles

The design focuses on scalability, data consistency, and real business use cases.

Proposed Database Schema

1. Companies Table

Stores company-level information.

```
CREATE TABLE companies (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Why: Each company can own multiple warehouses and products.

2. Warehouses Table

Each company can have multiple warehouses.

```
CREATE TABLE warehouses (
   id INT PRIMARY KEY AUTO_INCREMENT,
   company_id INT NOT NULL,
   name VARCHAR(255) NOT NULL,
   location VARCHAR(255),
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

Why: This allows scaling storage across different locations.

3. Products Table

Stores product details. Products are **not tied to warehouses directly**.

```
CREATE TABLE products (
   id INT PRIMARY KEY AUTO_INCREMENT,
   sku VARCHAR(100) UNIQUE NOT NULL,
   name VARCHAR(255) NOT NULL,
   price DECIMAL(10,2) NOT NULL,
   product_type VARCHAR(50),
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Why:

- SKU uniqueness is enforced
- Price uses DECIMAL for financial accuracy
- Product can exist in multiple warehouses

4. Inventory Table

Connects products and warehouses.

```
CREATE TABLE inventory (
   id INT PRIMARY KEY AUTO_INCREMENT,
   product_id INT NOT NULL,
   warehouse_id INT NOT NULL,
   quantity INT NOT NULL DEFAULT 0,
   last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   UNIQUE (product_id, warehouse_id),
   FOREIGN KEY (product_id) REFERENCES products(id),
   FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
);
```

Why:

- Same product can be stored in multiple warehouses
- Quantity tracked per warehouse

- Unique constraint avoids duplicate rows

5. Inventory History Table

Tracks every inventory change.

```
CREATE TABLE inventory_history (
  id INT PRIMARY KEY AUTO_INCREMENT,
  inventory_id INT NOT NULL,
  quantity_change INT NOT NULL,
  change_reason VARCHAR(255),
  changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (inventory_id) REFERENCES inventory(id)
);
```

Why:

Provides audit trail and helps with reporting and debugging.

6. Suppliers Table

Stores supplier details.

```
CREATE TABLE suppliers (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  contact_email VARCHAR(255)
);
```

7. Product–Supplier Mapping Table

Many-to-many relationship.

```
CREATE TABLE product_suppliers (
  product_id INT NOT NULL,
  supplier_id INT NOT NULL,
  PRIMARY KEY (product_id, supplier_id),
  FOREIGN KEY (product_id) REFERENCES products(id),
  FOREIGN KEY (supplier_id) REFERENCES suppliers(id)
);
```

Why:

One product can have multiple suppliers and vice versa.

8. Product Bundles Table

Used for bundled products.

```
CREATE TABLE product_bundles (
    bundle_product_id INT NOT NULL,
    child_product_id INT NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (bundle_product_id, child_product_id),
    FOREIGN KEY (bundle_product_id) REFERENCES products(id),
    FOREIGN KEY (child_product_id) REFERENCES products(id)
);
```

Why: Supports combo products (e.g., kits, packs).

Questions for the Product Team (Very Important)

1. Is low-stock threshold defined per **product type or per warehouse**?
2. What does **recent sales activity** mean (last 7 days, 30 days)?
3. Can suppliers supply products to **multiple companies**?
4. Should bundle price be **calculated automatically** or entered manually?
5. Are inventory updates **manual, automated, or both**?
6. Is negative inventory allowed?

Assumptions Made

- SKU is globally unique
- Products are company-specific
- Inventory is tracked per warehouse
- Price precision is critical
- Inventory changes must be auditable

Part 3: Low-Stock Alerts API

1. Problem Understanding

- We need to build an API that:
- Shows products running low on stock
- Works across multiple warehouses
- Only shows products with recent sales
- Includes supplier details for reordering

2. API Endpoint

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

3. Assumptions for This API

- Each product has a low_stock_threshold
- Recent sales = sales in last 30 days
- One primary supplier per product
- Days until stockout is calculated using average daily sales

4. Implementation (Flask Example)

```
from flask import jsonify
from datetime import datetime, timedelta
```

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

    inventories = db.session.query(
        Inventory,
        Product,
        Warehouse,
        Supplier
    ).join(Product) \
     .join(Warehouse) \
     .join(ProductSupplier) \
     .join(Supplier) \
     .filter(Warehouse.company_id == company_id) \
     .all()

    for inventory, product, warehouse, supplier in inventories:
        # Example values (assumed)
        threshold = product.low_stock_threshold
        avg_daily_sales = product.avg_daily_sales

        # Ignore products with no recent sales
        if avg_daily_sales == 0:
            continue
        days_until_stockout = inventory.quantity // avg_daily_sales

        if inventory.quantity <= threshold:
            alerts.append({
                "product_id": product.id,
                "product_name": product.name,
                "sku": product.sku,
                "warehouse_id": warehouse.id,
                "warehouse_name": warehouse.name,
                "current_stock": inventory.quantity,
                "threshold": threshold,
                "days_until_stockout": days_until_stockout,
                "supplier": {
                    "id": supplier.id,
                    "name": supplier.name,
                    "contact_email": supplier.contact_email
                }
            })

    return {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }, 200
```

Edge Cases Handled

- Products with zero sales ignored
- Multiple warehouses supported
- Products without suppliers skipped
- Prevents division by zero
- Only company-specific data returned

5. Why This Approach Is Good

- Scalable for large datasets
- Clean separation of concerns
- Business rules enforced
- Easy to extend in future
- Interviewer-friendly logic