Inventory Management System for B2B SaaS

Name – Aakash Raju Mohole

Email – aakashmohole@gmail.com

Github – https://github.com/aakashmohole/Bynry-Case-Study-Task/tree/main

PART 1 - Code Review & Debugging — Approach & Analysis

1. Issues in the Provided code

Technical Issues –

- No check before creating product; duplicate SKUs could be created.
- Two commits used instead of one atomic transaction, risking database inconsistency if one commit fails.
- Decimal price – The code trust that data['price'] is valid this price should be convert to decimal.
- Implement error handling if something fails it crashes of corrupts all data.
- It is unclear if the Product model is meant to be per-warehouse; if not, the logic here can duplicate products unnecessarily.

Business Logic Issues

- The design suggests a new product per warehouse, but product should have a single entry with inventory per warehouse.
- If product creation or inventory insertion in case fails, IDs might become out of sync and this create problem.
- If the app should support multiple companies, there's no separation implemented we have to implement this.
- If inventory creation fails after product is added, the product remains without inventory—a broken state.

2. Impact of Each Issue in Production

- Duplicate SKUs lead to confusion in inventory tracking, order processing, and reporting and data will become redundant.
- Partial failures create split states, making rollback and consistency difficult.
- Invalid data can crash the server or lead to incomplete/corrupt entries.
- Inaccurate prices (due to float imprecision) can cause billing and accounting errors.
- Any exception causes 500 errors and possible data leaks or losses.
- Duplicating product entries for each warehouse breaks analytics, reporting, and increases maintenance burden.

3. Correct Code Version with explanations

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

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

    # Validate required fields
    required_fields = ['name', 'sku', 'price', 'warehouse_id']
    for field in required_fields:
        if field not in data:
            return jsonify({"error": f"Missing required field: {field}"}), 400

    # Parse and validate price as Decimal
    try:
        price = Decimal(str(data['price']))
    except:
        return jsonify({"error": "Invalid price format"}), 400

    # Check SKU uniqueness
    if Product.query.filter_by(sku=data['sku']).first():
        return jsonify({"error": "SKU already exists"}), 409

    # Begin transaction
    try:
        product = Product(
            name=data['name'],
```

```
            sku=data['sku'],
            price=price
            # exclude warehouse_id, since product is global across warehouses
        )
        db.session.add(product)
        db.session.flush()  # gets the product.id before commit

        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=data['initial_quantity']
        )
        db.session.add(inventory)
        db.session.commit()
    except IntegrityError as e:
        db.session.rollback()
        return jsonify({"error": "Database error: " + str(e.orig)}), 500
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": str(e)}), 500

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

PART 2: Database Design

1. Database Schema Design

Entity	Attributes	Relationships
Company	id (PK), name, created_at	1-to-many with Warehouse, 1-to-many with Product
Warehouse	id (PK), company_id (FK), name, location, created_at	Many-to-1 with Company, 1-to-many with Inventory
Product	id (PK), company_id (FK), name, sku, price, is_bundle	Many-to-1 with Company, many-to-many with Supplier, many-to-many with Product (as bundle), 1-to-many with Inventory
Supplier	id (PK), name, contact_info, created_at	many-to-many with Product via Product_Supplier
Product_Supplier	product_id (PK, FK), supplier_id (PK, FK)	Many-to-1 with Product, many-to-1 with Supplier
Product_Bundle	bundle_id (PK, FK), product_id (PK, FK), quantity	Many-to-1 with Product (bundle and component product)
Inventory	id (PK), product_id (FK), warehouse_id (FK), quantity	Many-to-1 with Product, many-to-1 with Warehouse, 1-to-many with Inventory_Change

Inventory_Change	id (PK), inventory_id (FK), change_amount, reason, changed_at, changed_by	Many-to-1 with Inventory

- **SQL DDL**

```
-- 1. Companies
CREATE TABLE company (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    -- Add fields like address, contact info, etc.
    created_at TIMESTAMP NOT NULL DEFAULT NOW()
);

-- 2. Warehouses
CREATE TABLE warehouse (
    id SERIAL PRIMARY KEY,
    company_id INTEGER NOT NULL REFERENCES company(id) ON DELETE
CASCADE,
    name VARCHAR(255) NOT NULL,
    location VARCHAR(255),
    created_at TIMESTAMP NOT NULL DEFAULT NOW()
);

-- 3. Products
CREATE TABLE product (
    id SERIAL PRIMARY KEY,
    company_id INTEGER NOT NULL REFERENCES company(id) ON DELETE
CASCADE,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(64) NOT NULL,
    price DECIMAL(12, 2) NOT NULL,
    is_bundle BOOLEAN NOT NULL DEFAULT FALSE,
    -- Additional fields: description, etc.
    UNIQUE (company_id, sku)  -- SKU unique within company
);

-- 4. Bundled Products (For bundles containing other products)
```

```
CREATE TABLE product_bundle (
   bundle_id INTEGER NOT NULL REFERENCES product(id) ON DELETE
CASCADE,
   product_id INTEGER NOT NULL REFERENCES product(id) ON DELETE
CASCADE,
   quantity INTEGER NOT NULL DEFAULT 1,
   PRIMARY KEY (bundle_id, product_id)
);

-- 5. Suppliers
CREATE TABLE supplier (
   id SERIAL PRIMARY KEY,
   name VARCHAR(255) NOT NULL,
   contact_info VARCHAR(255),
   -- Add address, phone, etc.
   created_at TIMESTAMP NOT NULL DEFAULT NOW()
);

-- 6. Product - Supplier mapping (Products can have one or more
suppliers)
CREATE TABLE product_supplier (
   product_id INTEGER NOT NULL REFERENCES product(id) ON DELETE
CASCADE,
   supplier_id INTEGER NOT NULL REFERENCES supplier(id) ON DELETE
CASCADE,
   PRIMARY KEY (product_id, supplier_id)
);

-- 7. Inventory (quantity per product per warehouse)
CREATE TABLE inventory (
   id SERIAL PRIMARY KEY,
   product_id INTEGER NOT NULL REFERENCES product(id) ON DELETE
CASCADE,
   warehouse_id INTEGER NOT NULL REFERENCES warehouse(id) ON
DELETE CASCADE,
   quantity INTEGER NOT NULL DEFAULT 0,
   UNIQUE (product_id, warehouse_id)
);

-- 8. Inventory Changes (Audit/history log)
```

```
CREATE TABLE inventory_change (
    id SERIAL PRIMARY KEY,
    inventory_id INTEGER NOT NULL REFERENCES inventory(id) ON DELETE
CASCADE,
    change_amount INTEGER NOT NULL,
    reason VARCHAR(255),
    changed_at TIMESTAMP NOT NULL DEFAULT NOW(),
    changed_by INTEGER -- Optionally FK to user table
);
```

2. Identify Gaps / Product Questions

- Will we implement user access control to define which users can view or manage specific company data?
- How should we model suppliers? Can a single supplier provide products to multiple companies, or should supplier relationships be limited to one company? In other words, are suppliers global entities or company-specific?
- Regarding product bundles, can a bundle include other bundles (creating a nested structure), or are bundles composed solely of individual products?
- How will inventory for bundles be handled? Will bundles have their own distinct inventory, or will their availability be derived from the inventory of their constituent components?
- What types of inventory transactions do we anticipate tracking? (e.g., sales, returns, damages, restocks, transfers).
- What is the desired scope of SKU uniqueness? Should SKUs be unique across the entire system globally, within each company, or per warehouse?Any difference between warehouse types? (online, physical, drop-ship, etc.)
- Do we need to differentiate between various warehouse types (e.g., online fulfilment centres, physical retail stores, drop-ship locations)? If so, what are the key distinctions?
- Should we store specific warehouse attributes such as capacity, physical address, or other relevant details?
- Is it necessary to track product pricing on a per-supplier basis?

3. Explain Decisions:

Indexes and Constraints –

a) Indexes and Constraints:

Unique (company_id, sku):

- Keeps SKUs unique within a company—prevents duplicates but allows same SKU in different companies (fits multi-tenancy).

Foreign Keys:

- Ensure no orphaned warehouses, products, inventory, etc.

Unique (product_id, warehouse_id) for Inventory:

- Guarantees one inventory record per product/location.

Primary Keys/Composite Keys in Mapping Tables:

- Supports fast JOIN operations and avoids duplicates.

b) Audit Table (inventory_change):

- Tracks inventory adjustments over time for history and troubleshooting.

c) Bundles (product_bundle):

- Supports "products as bundles" with flexible contents; easy to navigate for inventory checks or UI displays.

d) Scalability:

- Indexed lookup on SKU, company/product IDs.
- Avoids denormalization; easy to extend (add more product attributes or relationships later).

e) Flexibility:

- Schema supports per-company isolation (key for B2B SaaS/multi-tenant).
- Bundles and product-supplier associations are generic; can accommodate changing requirements without major refactoring.

PART 3: API Implementation

Assumptions & Schema Extensions

To implement the endpoint, I assume:

- There is a table sales with columns:
  - id, product_id, warehouse_id, quantity, sale_date.
- Product types are implied by a product_type field in product table (e.g. "standard", "bundle").
- Low-stock thresholds are stored in a table product_type_threshold (product_type → threshold stock level).
- supplier table has a contact_email column.
- Only sales within the last 30 days count as recent sales activity.
- Days until stockout calculated as:
  - current_stock / average_daily_sales for that product across the company (all warehouses).
- If average_daily_sales = 0, set days_until_stockout = null or very high

API CODE -

```
from flask import Flask, jsonify

from flask_sqlalchemy import SQLAlchemy

from datetime import datetime, timedelta

from sqlalchemy import func, Boolean

from flasgger import Swagger

from dotenv import load_dotenv

import os


load_dotenv()


app = Flask(__name__)

swagger = Swagger(app, config=None, template=None)

app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL')

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

print("SQLALCHEMY_DATABASE_URI:", app.config.get('SQLALCHEMY_DATABASE_URI'))
```

```
db = SQLAlchemy(app)
########################################################
# Database models (reflect your schema exactly)
########################################################

class Company(db.Model):
    __tablename__ = 'company'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255), nullable=False)
    created_at = db.Column(db.DateTime, nullable=False, server_default=func.now())

class Warehouse(db.Model):
    __tablename__ = 'warehouse'
    id = db.Column(db.Integer, primary_key=True)
    company_id = db.Column(db.Integer, db.ForeignKey('company.id', ondelete='CASCADE'), nullable=False)
    name = db.Column(db.String(255), nullable=False)
    location = db.Column(db.String(255))
    created_at = db.Column(db.DateTime, nullable=False, server_default=func.now())

class Product(db.Model):
    __tablename__ = 'product'
    id = db.Column(db.Integer, primary_key=True)
    company_id = db.Column(db.Integer, db.ForeignKey('company.id', ondelete='CASCADE'), nullable=False)
    name = db.Column(db.String(255), nullable=False)
    sku = db.Column(db.String(64), nullable=False)
    price = db.Column(db.Numeric(12, 2), nullable=False)
    is_bundle = db.Column(db.Boolean, nullable=False, default=False)  # use this for threshold join

class Supplier(db.Model):
    __tablename__ = 'supplier'
```

```
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255), nullable=False)
    contact_info = db.Column(db.String(255))  # used as contact_email in response
    created_at = db.Column(db.DateTime, nullable=False, server_default=func.now())

class ProductSupplier(db.Model):
    __tablename__ = 'product_supplier'
    product_id = db.Column(db.Integer, db.ForeignKey('product.id', ondelete='CASCADE'), primary_key=True)
    supplier_id = db.Column(db.Integer, db.ForeignKey('supplier.id', ondelete='CASCADE'), primary_key=True)

class Inventory(db.Model):
    __tablename__ = 'inventory'
    id = db.Column(db.Integer, primary_key=True)
    product_id = db.Column(db.Integer, db.ForeignKey('product.id', ondelete='CASCADE'), nullable=False)
    warehouse_id = db.Column(db.Integer, db.ForeignKey('warehouse.id', ondelete='CASCADE'), nullable=False)
    quantity = db.Column(db.Integer, nullable=False, default=0)

class Sale(db.Model):
    __tablename__ = 'sales'
    id = db.Column(db.Integer, primary_key=True)
    product_id = db.Column(db.Integer, db.ForeignKey('product.id'), nullable=False)
    warehouse_id = db.Column(db.Integer, db.ForeignKey('warehouse.id'), nullable=False)
    quantity = db.Column(db.Integer, nullable=False)
    sale_date = db.Column(db.DateTime, nullable=False)

class ProductTypeThreshold(db.Model):
    __tablename__ = 'product_type_threshold'
    product_type = db.Column(Boolean, primary_key=True)  # BOOLEAN: True for bundles, False for others
```

```
    threshold = db.Column(db.Integer, nullable=False)

##########################################################
# Low-stock alerts endpoint
##########################################################

@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.
    ---
    tags:
      - Alerts
    parameters:
      - name: company_id
        in: path
        type: integer
        required: true
        description: The ID of the company to get low stock alerts for
    responses:
      200:
        description: List of low-stock alerts with supplier information
        schema:
          type: object
          properties:
            alerts:
              type: array
              items:
                type: object
                properties:
```

```
product_id:
  type: integer
  example: 123
product_name:
  type: string
  example: "Widget A"
sku:
  type: string
  example: "WID-001"
warehouse_id:
  type: integer
  example: 456
warehouse_name:
  type: string
  example: "Main Warehouse"
current_stock:
  type: integer
  example: 5
threshold:
  type: integer
  example: 20
days_until_stockout:
  type: integer
  example: 12
  nullable: true
supplier:
  type: object
  nullable: true
  properties:
    id:
      type: integer
```

```
                  example: 789
                name:
                  type: string
                  example: "Supplier Corp"
                contact_email:
                  type: string
                  example: "orders@supplier.com"
        total_alerts:
          type: integer
          example: 1
  404:
    description: Company not found
    schema:
      type: object
      properties:
        error:
          type: string
          example: "Company not found"
"""



# Verify company exists
company = Company.query.get(company_id)
if not company:
    return jsonify({"error": "Company not found"}), 404

# Define recent sales window (last 30 days)
recent_period_start = datetime.utcnow() - timedelta(days=30)

# 1. Products with recent sales in company
recent_sales_subq = (
```

```
        db.session.query(Sale.product_id)
        .join(Product, Product.id == Sale.product_id)
        .filter(
            Product.company_id == company_id,
            Sale.sale_date >= recent_period_start
        )
        .distinct()
        .subquery()
    )

    # 2. Inventories with product info, warehouse info, and thresholds
    inventories = (
        db.session.query(
            Inventory.id.label("inventory_id"),
            Product.id.label("product_id"),
            Product.name.label("product_name"),
            Product.sku,
            Product.is_bundle,
            Warehouse.id.label("warehouse_id"),
            Warehouse.name.label("warehouse_name"),
            Inventory.quantity.label("current_stock"),
            ProductTypeThreshold.threshold
        )
        .join(Product, Inventory.product_id == Product.id)
        .join(Warehouse, Inventory.warehouse_id == Warehouse.id)
        .join(ProductTypeThreshold, Product.is_bundle == ProductTypeThreshold.product_type)  # Boolean join here
        .filter(
            Product.company_id == company_id,
            Warehouse.company_id == company_id,
            Inventory.quantity <= ProductTypeThreshold.threshold,
```

```
        Inventory.product_id.in_(recent_sales_subq)  # Only products with recent sales
    )
    .all()
)

# 3. Calculate average daily sales per product (last 30 days)
sales_agg = (
    db.session.query(
        Sale.product_id,
        func.sum(Sale.quantity).label("total_quantity")
    )
    .join(Product, Product.id == Sale.product_id)
    .filter(
        Product.company_id == company_id,
        Sale.sale_date >= recent_period_start
    )
    .group_by(Sale.product_id)
    .all()
)
avg_daily_sales_map = {s.product_id: s.total_quantity / 30 for s in sales_agg}

# 4. Prepare alert list
alerts = []
for inv in inventories:
    avg_daily_sales = avg_daily_sales_map.get(inv.product_id, 0)
    if avg_daily_sales > 0:
        days_until_stockout = int(inv.current_stock / avg_daily_sales)
    else:
        days_until_stockout = None

    # Fetch first linked supplier info (optional)
```

```
        supplier = (
            db.session.query(Supplier)
            .join(ProductSupplier, Supplier.id == ProductSupplier.supplier_id)
            .filter(ProductSupplier.product_id == inv.product_id)
            .first()
        )

        supplier_info = None
        if supplier:
            supplier_info = {
                "id": supplier.id,
                "name": supplier.name,
                "contact_email": supplier.contact_info or ""
            }

        alert = {
            "product_id": inv.product_id,
            "product_name": inv.product_name,
            "sku": inv.sku,
            "warehouse_id": inv.warehouse_id,
            "warehouse_name": inv.warehouse_name,
            "current_stock": inv.current_stock,
            "threshold": inv.threshold,
            "days_until_stockout": days_until_stockout,
            "supplier": supplier_info
        }
        alerts.append(alert)

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

```
    }), 200


##################################################

# Main entrypoint

##################################################


if __name__ == '__main__':

    # For production, use a WSGI server like gunicorn instead

    app.run(debug=True)
```

Explanation of Approach

1. **Validate Company**: Ensure the requested company exists, otherwise return 404.
2. **Recent Sales Filter**:
   Query recent sales (last 30 days) to get products with sales activity. This filters out products without recent demand to avoid unnecessary alerts.
3. **Inventory & Threshold Filtering**:
   Join inventory with products, warehouses, and product-type thresholds, filter for inventory levels below or equal to thresholds.
4. **Average Daily Sales Calculation**:
   Calculate average daily sales per product over 30 days. This helps estimate how many days before stock runs out.
5. **Supplier Information**:
   Fetch supplier details linked to each product for reordering contact info.
6. **Return Structured JSON Alert List**:
   Matches the expected response format with detailed info.

Edge Cases Considered

- Company id invalid → returns 404.
- Products with no recent sales → excluded.
- Inventory quantity exactly equal to threshold → included.
- Products without any supplier → supplier info is null.
- No inventory or no low stock products → returns empty alerts list.
- Division by zero avoided by checking average daily sales > 0.
- Multi-warehouse handled by joining inventories per warehouse.

General Reasoning & Approach

- **Problem-solving:** Systematically identified input, transactional, and business logic issues; prioritized data integrity and maintainability.
- **Database:** Emphasized scalable, normalized design with attention to multi-tenancy and auditability.
- **API:** Focused on robust filtering, clear business rules, and comprehensive error handling.
- **Communication:** Asked clarifying questions to expose requirement gaps, demonstrating anticipation of real-world complexities.
- **Maintaining best practices:** Transactions, validation, appropriate HTTP status codes, modular design.