Stock Flow – Inventory Management System (Case Study)

Overview

This case study is based on building StockFlow, a B2B SaaS platform that helps small businesses manage their inventory across multiple warehouses and supplier networks. The project focuses on API design, debugging existing code, and database schema design to support real-world inventory workflows.

Part 1: Code Review & Debugging

Original Code -

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

    # Create new product
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=data['price'],
        warehouse_id=data['warehouse_id']
    )

    db.session.add(product)
    db.session.commit()

    # Update inventory count
    inventory = Inventory(
```

```
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=data['initial_quantity']
    )

    db.session.add(inventory)
    db.session.commit()
    return {"message": "Product created", "product_id": product.id}
```

Issues Identified

1. Business Logic Errors

- Products can exist in multiple warehouses, but the Product model stores warehouse_id (wrong one-to-one mapping).
- SKU uniqueness not enforced.
- Inventory tied to single warehouse.

2. Technical Issues

- initial_quantity may cause KeyError.
- Price handling should be Decimal.
- Multiple commits may cause partial saves.

Corrected Code -

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.json or {}
    required_fields = ['name', 'sku', 'price']
    for field in required_fields:
        if field not in data:
            return jsonify({"error": f"Missing field: {field}"}), 400
    try:
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=Decimal(str(data['price']))
        )
        db.session.add(product)
        db.session.flush()
        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)
        db.session.commit()
    except IntegrityError:
        db.session.rollback()
        return jsonify({"error": "SKU must be unique"}), 400
    return jsonify({"message": "Product created", "product_id":
product.id}), 201
```

Part 2: Database Design

Schema Design

```
-- Companies
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);
...
-- Inventory history
CREATE TABLE inventory_history (
    id SERIAL PRIMARY KEY,
    product_id INT REFERENCES products(id),
    warehouse_id INT REFERENCES warehouses(id),
    change INT NOT NULL,
    reason VARCHAR(255),
    changed_at TIMESTAMP DEFAULT NOW()
);
```

Questions to Product Team

- Should bundles allow nested bundles?
- Do we need product expiration dates?
- Is role-based access required?

Part 3: API Implementation – Low Stock Alerts

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

Implementation Example

```
@app.route('/api/companies/<int:company_id>/alerts/low-stock',
methods=['GET'])
def low_stock_alerts(company_id):
    alerts = []
    results = db.session.execute("""
        SELECT i.product_id, p.name AS product_name, p.sku,
               i.warehouse_id, w.name AS warehouse_name,
               i.quantity AS current_stock,
               p.threshold,
  s.id AS supplier_id, s.name AS supplier_name, s.contact_email
        FROM inventory i
        JOIN products p ON p.id = i.product_id
        JOIN warehouses w ON w.id = i.warehouse_id
        LEFT JOIN supplier_products sp ON sp.product_id = p.id
        LEFT JOIN suppliers s ON s.id = sp.supplier_id
        WHERE w.company_id = :company_id
          AND i.quantity < p.threshold
    """, {"company_id": company_id})
for row in results:
        avg_sales = get_avg_daily_sales(row.product_id,
company_id)
        days_until_stockout = (row.current_stock / avg_sales) if
avg_sales > 0 else None
        alerts.append({
            "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.contact_email
            }
        })

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

Conclusion -

This case study covered debugging a flawed API, designing a scalable database schema, and implementing a low-stock alert endpoint. The design emphasizes integrity, scalability, and flexibility for real-world inventory management.