STOCK FLOW CASE STUDY

OVERVIEW

STOCK FLOW IS A B2B INVENTORY MANAGEMENT PLATFORM ENABLING SMALL BUSINESSES TO MANAGE PRODUCTS ACROSS WAREHOUSES, TRACK INVENTORY, AND HANDLE SUPPLIER RELATIONSHIPS. THIS CASE STUDY INCLUDES THREE MAJOR PARTS:

- CODE REVIEW & DEBUGGING
- DATABASE DESIGN
- API IMPLEMENTATION

PART 1: CODE REVIEW & DEBUGGING

PROBLEMS FOUND

ISSUE	WHY IT MATTERS
MISSING VALIDATION	APP MAY CRASH IF KEY FIELDS LIKE SKU OR PRICE ARE MISSING
DUPLICATE SKUS ALLOWED	SKU SHOULD BE UNIQUE — NO CHECK BEFORE INSERT
TWO COMMIT() CALLS	PARTIAL SAVE IF THE FIRST SUCCEEDS BUT THE SECOND FAILS
NO ERROR HANDLING/ROLLBACK	RISK OF CORRUPT OR INCONSISTENT DB STATE
DECIMAL PRICE NOT HANDLED	USING FLOAT CAN CAUSE PRECISION ERRORS
NOT SCALABLE	HARDCODED TO A SINGLE WAREHOUSE; DOESN'T SUPPORT MULTI-LOCATION EXPANSION

FIXED CODE (FLASK + SQLALCHEMY)

```
from flask import Flask, request, jsonify
from decimal import Decimal, InvalidOperation
from sqlalchemy.exc import IntegrityError
from models import db, Product, Inventory  # assuming models are imported

app = Flask(__name__)

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

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

```
try:
    price = Decimal(str(data['price']))
except (InvalidOperation, ValueError):
    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

try:
    # Atomic transaction
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=price
    )
    db.session.add(product)
    db.session.flush()

    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=data['initial_quantity']
    )
    db.session.add(inventory)
    db.session.commit()

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

except IntegrityError:
    db.session.rollback()
    return jsonify({"error": "Database constraint violation"}), 500
except Exception as e:
    db.session.rollback()
    return jsonify({"error": str(e)}), 500
```

PART 2: DATABASE DESIGN

ENTITY TABLES

TABLE	DESCRIPTION
companies	Companies that own multiple warehouses
warehouses	Company-specific warehouse locations
products	Items with SKU, price, bundle flag
inventory	Stock per product per warehouse

TABLE	DESCRIPTION
inventory_history	Change log for auditing inventory
suppliers	External vendors
product_suppliers	Links suppliers to products
product_bundles	Maps component items to bundles
sales	Tracks product sales
product_thresholds	Custom thresholds for low stock alerts

SQL DDL SAMPLE (POSTGRESQL STYLE)

```
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE
);
CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    company_id INTEGER REFERENCES companies(id),
    name TEXT NOT NULL
);
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    sku TEXT UNIQUE NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE
);
CREATE TABLE inventory (
    id SERIAL PRIMARY KEY,
    warehouse_id INTEGER REFERENCES warehouses(id),
    product_id INTEGER REFERENCES products(id),
    quantity INTEGER NOT NULL,
    UNIQUE(product_id, warehouse_id)
);
CREATE TABLE inventory_history (
    id SERIAL PRIMARY KEY,
    product_id INTEGER REFERENCES products(id),
    warehouse_id INTEGER REFERENCES warehouses(id),
    changed_at TIMESTAMP DEFAULT now(),
    change INTEGER NOT NULL
);
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    contact_email TEXT
);
CREATE TABLE product_suppliers (
    id SERIAL PRIMARY KEY,
    product_id INTEGER REFERENCES products(id),
    supplier_id INTEGER REFERENCES suppliers(id)
);
CREATE TABLE product_bundles (
    id SERIAL PRIMARY KEY,
    bundle_id INTEGER REFERENCES products(id),
    component_product_id INTEGER REFERENCES
products(id),
    quantity INTEGER NOT NULL
);
```

```
CREATE TABLE sales
 id SERIAL PRIMARY KEY,
   product_id INTEGER REFERENCES products(id),
   sold_at TIMESTAMP DEFAULT now(),
   quantity INTEGER NOT NULL
);
CREATE TABLE product_thresholds (
   id SERIAL PRIMARY KEY,
   product_id INTEGER REFERENCES products(id),
   threshold INTEGER NOT NULL
);
```

Questions for Product Team

Q1 Can a product have multiple suppliers?
Q2 Should bundles maintain separate inventory or rely on component availability?
Q3 Are sales tracked per warehouse or company-wide?
Q4 Can products be archived or marked inactive?
Q5 Should thresholds be user-configured or calculated automatically?
Q6 Should we track product returns/refunds?

Design Decisions

Decision	Rationale
Use of Decimal	Prevents rounding issues with prices
inventory table	Many-to-many support between warehouses and products
inventory_history	Enables traceability and analytics
sales table	Supports calculating "days_until_stockout"
product_suppliers	Supports many-to-many product-supplier relationships
Indexes	Ensures fast lookups for SKU and stock queries

Part 3: API Implementation – Low Stock Alerts

Why This Feature Matters

For companies managing multiple warehouses and hundreds of products, it's critical to stay ahead of inventory shortages. This API helps business teams:

1. Automatically detect when stock drops below predefined thresholds.
2. Include supplier contact info to speed up restocking.
3. Filter alerts by warehouse, company, or recent sales.
4. Estimate how many days of stock remain before running out.
5. It's a tool designed to help operations, procurement, and sales teams make smarter, faster decisions.

ENDPOINT

GET /api/companies/<company_id>/alerts/low-stock

SAMPLE RESPONSE


```
{
 "ALERTS": [
  {
   "PRODUCT_ID": 101,
   "PRODUCT_NAME": "USB CABLE",
   "SKU": "USB-101",
   "WAREHOUSE_ID": 4,
   "WAREHOUSE_NAME": "MUMBAI CENTRAL",
   "CURRENT_STOCK": 8,
   "THRESHOLD": 20,
   "DAYS_UNTIL_STOCKOUT": 4,
   "SUPPLIER": {
    "ID": 7,
    "NAME": "ABC ELECTRONICS",
    "CONTACT_EMAIL": "ORDERS@ABCELECTRONICS.COM"
   }
  }
 ],
 "TOTAL_ALERTS": 1
}
```


PYTHON + FLASK IMPLEMENTATION

```
from flask import Flask, jsonify, request
from sqlalchemy import func
from datetime import datetime, timedelta
from models import db, Product, Inventory, Warehouse, ProductThreshold, Supplier, Sales,
ProductSupplier

@app.route("/api/companies/<int:company_id>/alerts/low-stock")
def low_stock_alerts(company_id):
   days = int(request.args.get("days", 30))
```

```
limit = int(request.args.get("limit", 50))
warehouse_id = request.args.get("warehouse_id", None)

since_date = datetime.utcnow() - timedelta(days=days)

# Subquery: total recent sales per product per warehouse
sales_subq = (
    db.session.query(
        Sales.product_id,
        Sales.warehouse_id,
        func.sum(Sales.quantity).label("total_sold"),
        func.avg(Sales.quantity).label("avg_daily_sales")
    )
    .filter(Sales.sold_at >= since_date)
    .group_by(Sales.product_id, Sales.warehouse_id)
    .subquery()
)

# Main query: join inventory + thresholds + sales + warehouse + supplier
query = (
    db.session.query(
        Inventory.product_id,
        Product.name.label("product_name"),
        Product.sku,
        Inventory.quantity.label("current_stock"),
        Inventory.warehouse_id,
        Warehouse.name.label("warehouse_name"),
        ProductThreshold.threshold,
        sales_subq.c.avg_daily_sales,
        Supplier.id.label("supplier_id"),
        Supplier.name.label("supplier_name"),
        Supplier.contact_email
    )
    .join(Product, Product.id == Inventory.product_id)
    .join(Warehouse, Warehouse.id == Inventory.warehouse_id)
    .join(ProductThreshold, ProductThreshold.product_id == Product.id)
    .outerjoin(sales_subq,
            (sales_subq.c.product_id == Inventory.product_id) &
            (sales_subq.c.warehouse_id == Inventory.warehouse_id))
    .outerjoin(ProductSupplier, ProductSupplier.product_id == Product.id)
    .outerjoin(Supplier, Supplier.id == ProductSupplier.supplier_id)
    .filter(Warehouse.company_id == company_id)
)

IF WAREHOUSE_ID:
    QUERY = QUERY.FILTER(INVENTORY.WAREHOUSE_ID == WAREHOUSE_ID)
```

```
RESULTS = QUERY.LIMIT(LIMIT).ALL()

ALERTS = []
FOR ROW IN RESULTS:
  IF ROW.CURRENT_STOCK < ROW.THRESHOLD:
    # ESTIMATE STOCKOUT DAYS IF SALES DATA EXISTS
    IF ROW.AVG_DAILY_SALES AND ROW.AVG_DAILY_SALES > 0:
      DAYS_UNTIL_STOCKOUT = ROUND(ROW.CURRENT_STOCK / ROW.AVG_DAILY_SALES)
    ELSE:
      DAYS_UNTIL_STOCKOUT = 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
      } IF ROW.SUPPLIER_ID ELSE NONE
    })

RETURN JSONIFY({
  "ALERTS": ALERTS,
  "TOTAL_ALERTS": LEN(ALERTS)
})
```