StockFlow Engineering Case Study Response

Candidate: Atharva Gaikwad
Role: Backend Engineering Intern
Technology Stack: Python (Flask), PostgreSQL, SQLAlchemy

Part 1 – Code Review & Fixes

1. Issues Identified

#	Type	Problem	Risk in Production
1	Validation	data = request.json used unvalidated input	Runtime exceptions, invalid or malicious data
2	Atomicity	Separate db.session.commit() calls	If second fails, DB ends up in inconsistent state
3	Business Rule	SKU not checked for uniqueness	Duplicate product entries, inconsistent inventory
4	Data Model Design	price is a float	Rounding errors in financial computations
5	Error Handling	No try-except logic or error status codes	500s or silent failures
6	Assumptions	warehouse_id is used as a column in Product	Violates normalization; product not tied to just one warehouse

2. Impact

- System allows duplicate SKUs, violating uniqueness guarantee.
- Partial commits can leave products without inventory, or orphaned records.
- financial calculations based on float may yield incorrect totals.
- Missing status codes confuse frontend/error logs.
- Security issues (e.g., JSON injection) from lack of validation.

3. Corrected Implementation

Moved to transactional, fully validated, normalized version:
(Python)

```python
# validation schema
from marshmallow import Schema, fields, ValidationError
```

```
class ProductSchema(Schema):
   name = fields.String(required=True)
   sku = fields.String(required=True)
   price_cents = fields.Integer(required=True)  # storing as cents
   warehouse_id = fields.Integer(required=True)
   initial_quantity = fields.Integer(required=True)
```

Code:

```
# route: /api/products
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError
from http import HTTPStatus

@app.route('/api/products', methods=['POST'])
def create_product():
   try:
       data = ProductSchema().load(request.json)
   except ValidationError as err:
       return {"errors": err.messages}, HTTPStatus.UNPROCESSABLE_ENTITY

   try:
       with db.session.begin():
           # Ensure SKU is unique
           existing =
db.session.query(Product.id).filter_by(sku=data['sku']).first()
           if existing:
               return {"error": "SKU already exists"}, HTTPStatus.CONFLICT

           # Create product
           product = Product(name=data['name'], sku=data['sku'],
price_cents=data['price_cents'])
           db.session.add(product)
           db.session.flush()  # generate `product.id`

           # Check warehouse exists
           warehouse = Warehouse.query.get(data['warehouse_id'])
           if not warehouse:
               return {"error": "Warehouse not found"}, HTTPStatus.NOT_FOUND

           inventory = Inventory(
               product_id=product.id,
```

```
                warehouse_id=data['warehouse_id'],
                quantity=data['initial_quantity']
            )
            db.session.add(inventory)

        return {"message": "Product created", "product_id": product.id},
HTTPStatus.CREATED

    except IntegrityError:
        db.session.rollback()
        return {"error": "Integrity error"}, HTTPStatus.INTERNAL_SERVER_ERROR
```

Key Improvements

- All logic wrapped in db.session.begin() for atomicity.
- Uses Marshmallow schema validation and provides appropriate HTTP status codes.
- Price stored as integer (price_cents) to prevent float arithmetic bugs.
- Unique SKU check enforced both at app level and with DB constraint.

Part 2 – Database Design

Schema (PostgreSQL DDL Style)

```sql
CREATE TABLE companies (
 id SERIAL PRIMARY KEY,
 name TEXT NOT NULL
);

CREATE TABLE warehouses (
 id SERIAL PRIMARY KEY,
 company_id INTEGER NOT NULL REFERENCES companies(id),
 name TEXT NOT NULL,
 UNIQUE(company_id, name)
);

CREATE TABLE products (
 id SERIAL PRIMARY KEY,
 name TEXT NOT NULL,
 sku TEXT NOT NULL UNIQUE,
 price_cents INTEGER CHECK (price_cents >= 0),
```

```sql
 is_bundle BOOLEAN DEFAULT FALSE
);

CREATE TABLE inventories (
 product_id INTEGER REFERENCES products(id),
 warehouse_id INTEGER REFERENCES warehouses(id),
 quantity INTEGER DEFAULT 0,
 PRIMARY KEY (product_id, warehouse_id)
);

CREATE TABLE inventory_transactions (
 id BIGSERIAL PRIMARY KEY,
 product_id INTEGER NOT NULL,
 warehouse_id INTEGER NOT NULL,
 delta INTEGER NOT NULL,
 reason TEXT,
 tx_time TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE suppliers (
 id SERIAL PRIMARY KEY,
 name TEXT NOT NULL,
 contact_email TEXT
);

CREATE TABLE supplier_products (
 supplier_id INTEGER NOT NULL REFERENCES suppliers(id),
 product_id INTEGER NOT NULL REFERENCES products(id),
 lead_time_days INTEGER,
 PRIMARY KEY (supplier_id, product_id)
);

CREATE TABLE reorder_policies (
 product_id INTEGER PRIMARY KEY REFERENCES products(id),
 threshold_qty INTEGER NOT NULL
);

-- Bundle products (BOM)
CREATE TABLE product_components (
 parent_product_id INTEGER REFERENCES products(id),
 component_id INTEGER REFERENCES products(id),
 qty INTEGER NOT NULL,
 PRIMARY KEY (parent_product_id, component_id)
);
```

Design Justification

- Composite PK in inventories gives O(1) lookup by product-warehouse.
- Normalized schema for flexibility: products separate from inventory.
- product_components supports bundles as multi-product compositions.
- inventory_transactions mirrors accounting ledgers, preserving audit trails.
- Indexes on (product_id), (warehouse_id) and SKU.

Questions to Product Team (Gaps)

1. Do we need lot/serial tracking (e.g. for perishable goods or recalls)?
2. Should we support multi-unit measures (e.g., 1 case = 12 items)?
3. Do products ever have multiple suppliers/tiered costs?
4. Should inventory_transactions track user/actions who made the change?

Part 3 – Low-Stock Alert API

Assumptions

- inventory_transactions capture all stock movements with +/- delta values.
- "Recent sales activity" means transactions in the last 30 days.
- Low stock = quantity < threshold in reorder_policies.
- Supplier link via supplier_products, sorted by lead_time_days.

Implementation (Flask + SQL)

```
python

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
   query = text("""
   WITH recent_sales AS (
       SELECT product_id, warehouse_id, SUM(-delta) AS sales_30d
       FROM inventory_transactions
       WHERE reason = 'sale' AND tx_time >= now() - INTERVAL '30 days'
```

```
        GROUP BY product_id, warehouse_id
    ),
    thresholded AS (
        SELECT i.product_id, i.warehouse_id, i.quantity, rp.threshold_qty,
               COALESCE(rs.sales_30d, 0) AS sales_30d
        FROM inventories i
        JOIN reorder_policies rp ON rp.product_id = i.product_id
        LEFT JOIN recent_sales rs ON rs.product_id = i.product_id AND
rs.warehouse_id = i.warehouse_id
    ),
    supplier_info AS (
        SELECT sp.product_id,
               sp.supplier_id,
               s.name AS supplier_name,
               s.contact_email,
               ROW_NUMBER() OVER (PARTITION BY sp.product_id ORDER BY
sp.lead_time_days) = 1 AS first_choice
        FROM supplier_products sp
        JOIN suppliers s ON s.id = sp.supplier_id
    )
    SELECT p.id AS product_id,
           p.name AS product_name,
           p.sku,
           w.id AS warehouse_id,
           w.name AS warehouse_name,
           t.quantity AS current_stock,
           t.threshold_qty AS threshold,
           CASE WHEN t.sales_30d = 0 THEN NULL
                ELSE CEIL(t.quantity / (t.sales_30d / 30.0)) END AS
days_until_stockout,
           jsonb_build_object(
              'id', si.supplier_id,
              'name', si.supplier_name,
              'contact_email', si.contact_email
           ) AS supplier
    FROM thresholded t
    JOIN products p ON p.id = t.product_id
    JOIN warehouses w ON w.id = t.warehouse_id AND w.company_id = :company_id
    LEFT JOIN supplier_info si ON si.product_id = p.id AND si.first_choice
    WHERE t.quantity < t.threshold_qty
    ORDER BY t.quantity ASC
    LIMIT 100
    """)

    res = db.session.execute(query, {"company_id": company_id})
    alerts = [dict(row._mapping) for row in res]
```

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

Edge Case Handling

- Zero sales? → days_until_stockout is null (can’t extrapolate).
- Missing supplier? → supplier = null handled gracefully.
- No alerts if stock > threshold or no recent sales.
- Only alerts for the given company_id.

Improvements If Time Allowed

- Paging with limit/offset via request.args.
- Caching with Redis/materialized view for performance.
- Precompute daily aggregates of sales to reduce query cost.

Final Notes

Key Assumptions Made

- price handled as cents (int) for accuracy.
- Time window for sales = 30 days.
- Fastest supplier: lowest lead_time_days.
- Bundles are treated as separate product entries (no real-time BOM unpacking).
- No multi-currency, taxes, or UoM conversions.

Alternative Considerations

- Use SQL views or data warehouse (e.g., BigQuery) for complex analytics/reporting.
- Logic for bundle alerts could recursively compute dependency trees but was omitted for simplicity.

Prepared by: Atharva Gaikwad
atharva.gaikwad211@gmail.com | 7301321111