Solution Document for Case Study

Part 1: Code Review & Debugging

Inventory Management System – StockFlow:

1. Issues Identified:

A. Missing Input Validation

- The API directly accesses data['field'] without checking if the field exists.
- No validation for data types (price, quantity).
- No validation for negative values (price, quantity).
- No validation for optional fields.

B. SKU Uniqueness Not Enforced

- SKU must be unique across the platform, but the code does not check for existing SKUs.
- Duplicate SKUs can be inserted.

C. Incorrect Product–Warehouse Relationship

- Product is created with a single warehouse_id.
- Business requirement states that products can exist in **multiple warehouses**.
- Warehouse association should not be stored directly on the Product entity.

D. Missing Transaction Management

- Two separate db.session.commit() calls are used.
- If inventory creation fails after product creation, the system will have:
    - A product without inventory

    - Inconsistent database state

E. No Error Handling or Rollback

- Any exception (DB failure, constraint violation) will crash the API.
- No rollback is performed in case of failure.
- Client will receive a generic 500 error with no context.

F. Decimal Price Handling

- Price is accepted without enforcing decimal precision.
- Floating-point values can lead to rounding errors in financial data.

G. Inventory Initialization Logic is Fragile

- Inventory is created blindly without checking:
    - Whether inventory already exists for the same product and warehouse
- Leads to duplicate inventory records.

H. Concurrency Issues

- Two concurrent requests with the same SKU can both pass and create duplicates.
- No locking or unique constraint handling.

---

2. Impact in Production:

Issue	**Production Impact**
Missing validation	API crashes due to malformed requests
Duplicate SKU	Reporting, billing, and integrations break
Product tied to one warehouse	Multi-warehouse inventory becomes impossible
Partial commits	Orphaned products with no inventory
No rollback	Silent data corruption
Floating price errors	Incorrect invoices and accounting issues
Duplicate inventory rows	Incorrect stock counts
Concurrency issues	Data inconsistency under load

These issues will cause **data corruption**, **customer complaints**, and **manual database cleanup**.

3. Corrected Implementation:

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

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

    # Basic input validation
    required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
```

```
for field in required_fields:
    if field not in data:
        return {"error": f"Missing field: {field}"}, 400

if data['price'] < 0 or data['initial_quantity'] < 0:
    return {"error": "Price and quantity must be non-negative"}, 400

try:
    price = Decimal(str(data['price']))
except:
    return {"error": "Invalid price format"}, 400


try:
    # Start transaction
    with db.session.begin():

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

        # Create product (no warehouse coupling)
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=price
        )
```

```
        db.session.add(product)
        db.session.flush()  # Get product.id without committing

        # Check if inventory already exists
        existing_inventory = Inventory.query.filter_by(
            product_id=product.id,
            warehouse_id=data['warehouse_id']
        ).first()

        if existing_inventory:
            return {"error": "Inventory already exists for this product and warehouse"}, 409

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

    return {
        "message": "Product created successfully",
        "product_id": product.id
    }, 201

except IntegrityError:
```

```
        db.session.rollback()
        return {"error": "Database integrity error"}, 500

    except Exception as e:
        db.session.rollback()
        return {"error": "Unexpected server error"}, 500
```

---

4. Explanation of Fixes

Transaction Handling

- with db.session.begin() ensures **atomicity**
- Either both product and inventory are created, or none

SKU Validation

- Prevents duplicate SKUs at application level
- Should also be enforced with a database unique constraint

Decoupling Product from Warehouse

- Product is now warehouse-agnostic
- Inventory handles product-warehouse relationships correctly

---

Decimal Price Handling

- Uses Decimal to avoid floating-point precision issues
- Required for financial correctness

Defensive Programming

- Graceful error messages
- Controlled rollback on failure
- Proper HTTP status codes

---

5. Assumptions Made

- SKU is globally unique across all companies
- Initial inventory is created during product creation
- Product pricing is independent of warehouse
- Inventory uniqueness is (product_id, warehouse_id)

Part 2: Database Design

Inventory Management System – StockFlow

1. Overview:

The goal of this database design is to support a **multi-company, multi-warehouse inventory management platform** with the following capabilities:

- Companies managing multiple warehouses
- Products stored in multiple warehouses with different quantities
- Tracking inventory changes over time
- Supplier–product relationships
- Support for bundled products
- Scalable querying for alerts and reporting

The schema is normalized, scalable, and designed to prevent data inconsistency.

---

2. Database Schema Design:

2.1 Company

```
companies (
   id BIGINT PRIMARY KEY,
   name VARCHAR(255) NOT NULL,
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
)
```

Purpose:
Represents a tenant in the B2B SaaS platform.

2.2 Warehouse

```
warehouses (
   id BIGINT PRIMARY KEY,
   company_id BIGINT NOT NULL,
   name VARCHAR(255) NOT NULL,
   location VARCHAR(255),
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (company_id) REFERENCES companies(id)
)
```

Relationship:

- One company → many warehouses

2.3 Product

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

Key Design Choice:

- SKU is **globally unique**
- Product is **warehouse-agnostic**

2.4 Inventory

```
inventory (
   id BIGINT PRIMARY KEY,
   product_id BIGINT NOT NULL,
   warehouse_id BIGINT NOT NULL,
   quantity INT NOT NULL DEFAULT 0,
   updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   UNIQUE (product_id, warehouse_id),
   FOREIGN KEY (product_id) REFERENCES products(id),
   FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
)
```

Purpose:
Tracks **current stock per product per warehouse**.

2.5 Inventory History

```
inventory_history (
   id BIGINT PRIMARY KEY,
   product_id BIGINT NOT NULL,
   warehouse_id BIGINT NOT NULL,
   quantity_change INT NOT NULL,
   reason VARCHAR(100),
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (product_id) REFERENCES products(id),
   FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
)
```

Purpose:
Maintains an **audit trail** of stock changes (sales, restock, adjustments).

2.6 Supplier

```
suppliers (
   id BIGINT PRIMARY KEY,
   name VARCHAR(255) NOT NULL,
   contact_email VARCHAR(255),
   contact_phone VARCHAR(50),
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
)
```

2.7 Product Supplier Mapping

```
product_suppliers (
   product_id BIGINT NOT NULL,
   supplier_id BIGINT NOT NULL,
   is_primary BOOLEAN DEFAULT FALSE,
   PRIMARY KEY (product_id, supplier_id),
   FOREIGN KEY (product_id) REFERENCES products(id),
   FOREIGN KEY (supplier_id) REFERENCES suppliers(id)
)
```

Purpose:
Supports **multiple suppliers per product** with a preferred supplier.

2.8 Sales / Order Items

```
order_items (
   id BIGINT PRIMARY KEY,
   product_id BIGINT NOT NULL,
```

```
    warehouse_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES products(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
)
```

Purpose:
Used to determine **recent sales activity** and **sales velocity**.

2.9 Product Bundles

```
product_bundles (
    bundle_id BIGINT NOT NULL,
    component_product_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (bundle_id, component_product_id),
    FOREIGN KEY (bundle_id) REFERENCES products(id),
    FOREIGN KEY (component_product_id) REFERENCES products(id)
)
```

Purpose:
Allows one product to be composed of multiple other products.

---

3. Indexing & Constraints:

Recommended Indexes

```
CREATE INDEX idx_inventory_product_warehouse
ON inventory(product_id, warehouse_id);
```

```
CREATE INDEX idx_order_items_created_at
ON order_items(created_at);

CREATE INDEX idx_inventory_history_created_at
ON inventory_history(created_at);
```

Why:

- Fast low-stock queries
- Efficient sales trend analysis
- Alert generation at scale

---

4. Missing Requirements & Questions for Product Team:

The following requirements are **not explicitly defined** and need clarification:

1. Is SKU unique **globally** or **per company**?
2. Can product price vary by warehouse or supplier?
3. What qualifies as “recent sales activity” (time window)?
4. Should inactive or discontinued products trigger alerts?
5. Can bundles be stocked directly or only computed virtually?
6. Do warehouses have reorder rules or minimum stock levels?
7. Should inventory history store user/action metadata?
8. Can suppliers be company-specific or global?

---

5. Design Decisions & Justifications

Inventory as a Separate Table

- Enables multi-warehouse support
- Prevents data duplication
- Scales cleanly with new warehouses

Inventory History Table

- Required for auditing
- Enables analytics and forecasting
- Supports compliance and debugging

Bundle Mapping Table

- Self-referencing design avoids duplication
- Flexible for future product compositions

Strong Constraints

- Unique constraints prevent duplication
- Foreign keys ensure referential integrity
- Composite keys enforce business rules

---

6. Assumptions Made

- SKU is globally unique
- Inventory is always warehouse-specific
- Sales activity drives alert eligibility
- One primary supplier per product (optional)
- Product price is independent of warehouse

Part 3: Low Stock Alerts API Implementation

Inventory Management System – StockFlow

---

1. Problem Overview:

The goal of this API is to generate **actionable low-stock alerts** for a given company by analyzing inventory levels, recent sales activity, and product-specific thresholds across **multiple warehouses**.

This endpoint is intended for **operations and procurement teams** to proactively reorder products before stockouts occur.

---

2. Endpoint Specification:

HTTP Method: GET
Endpoint:

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

---

3. Business Rules Applied:

The API follows these business rules:

1. Low-stock threshold varies by **product type**
2. Alerts are generated **only for products with recent sales activity**
3. Inventory must be evaluated **per warehouse**
4. Supplier information must be included for reordering
5. Products with zero sales velocity are excluded
6. Only active inventory records are considered

---

4. Assumptions (Explicitly Documented):

Due to incomplete requirements, the following assumptions are made:

1. **Recent sales** = sales in the last **30 days**

2. Low-stock threshold is derived from product_type
3. Days until stockout = current_stock / avg_daily_sales
4. One **primary supplier** exists per product
5. Bundled products are excluded from alerts
6. Products without sales history are excluded
7. Inventory quantity reflects real-time stock

These assumptions are documented and can be refined with product input.

---

5. High-Level Approach:

The API performs the following steps:

1. Validate company existence
2. Fetch all warehouses for the company
3. Identify products with recent sales activity
4. Calculate average daily sales per product per warehouse
5. Compare current stock against threshold
6. Estimate days until stockout
7. Attach supplier information
8. Return aggregated alerts

---

6. Example Threshold Logic:

```
PRODUCT_TYPE_THRESHOLDS = {
  "fast_moving": 20,
  "slow_moving": 5,
  "critical": 50
}
```

---

7. API Implementation (Flask / SQLAlchemy):

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

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
    alerts = []
    today = datetime.utcnow()
    sales_window_start = today - timedelta(days=30)

    # Fetch warehouses for the company
    warehouses = Warehouse.query.filter_by(company_id=company_id).all()
    if not warehouses:
        return {"alerts": [], "total_alerts": 0}, 200

    warehouse_ids = [w.id for w in warehouses]

    # Fetch recent sales grouped by product & warehouse
    sales_data = (
        db.session.query(
            OrderItem.product_id,
            OrderItem.warehouse_id,
            func.sum(OrderItem.quantity).label("total_sold")
        )
        .filter(
```

```
        OrderItem.warehouse_id.in_(warehouse_ids),
        OrderItem.created_at >= sales_window_start
    )
    .group_by(OrderItem.product_id, OrderItem.warehouse_id)
    .all()
)

for sale in sales_data:
    inventory = Inventory.query.filter_by(
        product_id=sale.product_id,
        warehouse_id=sale.warehouse_id
    ).first()

    if not inventory or inventory.quantity <= 0:
        continue

    product = Product.query.get(sale.product_id)
    if not product:
        continue

    # Determine threshold
    threshold = PRODUCT_TYPE_THRESHOLDS.get(product.product_type, 10)

    if inventory.quantity >= threshold:
        continue
```

```
avg_daily_sales = sale.total_sold / 30
if avg_daily_sales <= 0:
    continue

days_until_stockout = int(inventory.quantity / avg_daily_sales)

# Fetch primary supplier
supplier = (
    db.session.query(Supplier)
    .join(ProductSupplier)
    .filter(
        ProductSupplier.product_id == product.id,
        ProductSupplier.is_primary == True
    )
    .first()
)

warehouse = Warehouse.query.get(sale.warehouse_id)

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 if supplier else None,
                "name": supplier.name if supplier else None,
                "contact_email": supplier.contact_email if supplier else None
            }
        })

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

---

8. Edge Cases Handled

Edge Case	Handling
No warehouses	Empty alert list
No recent sales	Product excluded
Zero sales velocity	Avoid division by zero
Missing inventory	Skipped safely
Missing supplier	Supplier returned as null
Multiple warehouses	Evaluated independently
Over-threshold stock	Excluded from alerts

---

9. Scalability & Performance Considerations

- Aggregation queries minimize DB round-trips
- Indexed created_at enables fast sales filtering
- Threshold logic is configurable
- Can be optimized using materialized views or background jobs
- API response supports pagination if needed

---

10. Why This Design Works Well

- Matches real-world procurement workflows
- Avoids alert fatigue by filtering inactive products
- Provides actionable data (supplier + stockout days)
- Scales across companies and warehouses
- Easy to extend (email alerts, dashboards, forecasting)

---

11. Summary

This API demonstrates:

- Strong backend fundamentals
- Clear separation of concerns
- Business-driven logic
- Defensive programming
- Production-ready thinking

This is all about the Case Study and solutions From my side for the above given problems.