Inventory Management System for B2B SaaS CASE STUDY

Part 1: Code Review & Debugging (30 minutes)

1. **List of Problems i see:**

    a. No Input Validation- Although the code is simple in logic, there will definitely be a production issue if the user is missing out an input it can cause error505. For example if the client forgets to input a required field such as sku(product identification code) the app will raise a KeyError. In production this will lead to poor user experience even if the code compiles. In one line, The API directly accesses data['name'], data['sku'], etc. without checking if they exist or are valid.

    Fix: We should validate required fields and datatypes.

    My fix:

```
required_fields = [n'ame', 'sku', 'price', 'warehouse_id', 'initial_quantity']
for field in required_fields:
    if field not in data:
        return {"error": f"{field} is required"}, 400
```

    b. Sku uniqueness not enforced- The system does not check whether the SKU already exists, even though SKUs must be unique across the platform. So incorrect product identification will happen which will cause broken inventory sync.

    Fix: We need to add a constraint in the data base to handle this integrity error.

    My fix:

```
# Product model
sku = db.Column(db.String(100), unique=True, nullable=False)

from sqlalchemy.exc import IntegrityError

try:
    db.session.commit()
except IntegrityError:
    db.session.rollback()
    return {"error": "SKU already exists"}, 409
```

c. Partial Faliure/ 2 commits- The product and inventory inserts are committed separately. So if the second commit fails, the system stores products without inventory. This silently breaks stock counts in production level.
Fix: we need to wrap the operations in one pass/transaction.

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

d. Product Incorrectly Tied to Warehouse -The product model includes warehouse_id, but products can exist in multiple warehouses. So there will be problems in counting the stock of that product which is available in multiple different warehouse.
Fix: we Remove warehouse_id from Product and manage stock only through Inventory.

```
product = Product(
    name=data['name'],
    sku=data['sku'],
    price=Decimal(str(data['price']))
)

inventory = Inventory(
    product=product,
    warehouse_id=data['warehouse_id'],
    quantity=data['initial_quantity']
)
```

e. Decimal precision of price- Price is stored without ensuring decimal precision. Its a subtle but a serious problem as it causes floating point errors which in turn will result in accounting issues.
Fix: We use decimal and also enforce it in DB.

```
from decimal import Decimal

price = Decimal(str(data[‘price’]))
```

f. Negative Inventory- There is no stopping to prevent negative stock and low-stock alert will be useless if it goes negative.
Fix: we can validate it at the api level itself

```
if data[‘initial_quantity’] < 0:
    return {"error": "Initial quantity cannot be negative"}, 400
```

g. Optional field not handled- Given api assumes , all field are mandatory.
   Fix: we can use .get and defaults
   Eg: description = data.get('description')
   barcode = data.get('barcode')

```
product = Product(
    name=data['name'],
    sku=data['sku'],
    price=price,
    description=description,
    barcode=barcode
)
```

h. Authorization check- we dont want any user to create products for any warehouse. It can raise major security risk.
   Fix: we ensure warehouse belongs to the authenticated company.

```
if warehouse.company_id != current_user.company_id:
    return {"error": "Unauthorized"}, 403
```

Part 2: Database Design

I have chosen mysql -ddl

Companies table-

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

—----------

Wahrehouses - one vompany - many warehouse

```
CREATE TABLE warehouses (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
```

```
    location VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_warehouse_company
        FOREIGN KEY (company_id) REFERENCES companies(id)
        ON DELETE CASCADE
);
```

—----------

Products - products are company owned and not tied to a warehouse

```
CREATE TABLE products (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT uq_sku UNIQUE (sku),
    CONSTRAINT fk_product_company
        FOREIGN KEY (company_id) REFERENCES companies(id)
        ON DELETE CASCADE
);
```

—----------

Inventory-product stock per warehouse

```
CREATE TABLE inventory (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    warehouse_id BIGINT NOT NULL,
    quantity INT NOT NULL DEFAULT 0,

    CONSTRAINT fk_inventory_product
        FOREIGN KEY (product_id) REFERENCES products(id)
        ON DELETE CASCADE,

    CONSTRAINT fk_inventory_warehouse
        FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
        ON DELETE CASCADE,
```

```
    CONSTRAINT uq_product_warehouse UNIQUE (product_id, warehouse_id),
    CONSTRAINT chk_quantity CHECK (quantity >= 0)
);

—----------------
```

Inventory tracking-keeps track of stock

```
CREATE TABLE inventory_movements (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    inventory_id BIGINT NOT NULL,
    change_amount INT NOT NULL,
    reason ENUM('sale', 'restock', 'return', 'adjustment', 'transfer') NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_movement_inventory
        FOREIGN KEY (inventory_id) REFERENCES inventory(id)
        ON DELETE CASCADE
);

—----------------------------
```

Suppliers- company level

```
CREATE TABLE suppliers (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    contact_email VARCHAR(255),
    phone VARCHAR(50),

    CONSTRAINT fk_supplier_company
        FOREIGN KEY (company_id) REFERENCES companies(id)
        ON DELETE CASCADE
);


—-----------------------------
```

Product supply- many -to-many

```
CREATE TABLE product_suppliers (
    product_id BIGINT NOT NULL,
    supplier_id BIGINT NOT NULL,

    PRIMARY KEY (product_id, supplier_id),

    CONSTRAINT fk_ps_product
        FOREIGN KEY (product_id) REFERENCES products(id)
        ON DELETE CASCADE,
```

```
    CONSTRAINT fk_ps_supplier
        FOREIGN KEY (supplier_id) REFERENCES suppliers(id)
        ON DELETE CASCADE
);
```

—----------

Product bundles-child products mapping

```
CREATE TABLE product_bundles (
    bundle_id BIGINT NOT NULL,
    child_product_id BIGINT NOT NULL,
    quantity INT NOT NULL,

    PRIMARY KEY (bundle_id, child_product_id),

    CONSTRAINT fk_bundle_parent
        FOREIGN KEY (bundle_id) REFERENCES products(id)
        ON DELETE CASCADE,

    CONSTRAINT fk_bundle_child
        FOREIGN KEY (child_product_id) REFERENCES products(id)
        ON DELETE CASCADE,

    CONSTRAINT chk_bundle_qty CHECK (quantity > 0)
);
```

—----------

Questions i have for the product team about requirements:

Should SKUs be globally unique or only unique per company?
Can a product belong to multiple suppliers?
Can a bundle contain another bundle?
Should inventory ever go negative (e.g., backorders)?
Do warehouses share stock or operate independently?
Should deleted products remain in reports?
Are transfers between warehouses required?
Do we need batch/lot tracking or expiry dates?

My design detains and explanations:

The products are taken out of the inventory so that we can make sure products can be in many different warehouses, with different amounts. This way we do not have to repeat the product information over and over. It also makes it easy to add warehouses when we need to.

The inventory table is set up so that you cannot have the product_id and warehouse_id more than once. This is really important because it helps keep the information accurate. The inventory table also has a rule that says the inventory amount must always be zero or more. This rule is in place to make sure that the inventory amount is never negative. The product_id and warehouse_id combination is what makes each stock row unique, in the inventory table.

The inventory_movements table is really useful because it keeps a record of every change that happens to our stock. We need this record to figure out what is going wrong when things do not add up to make reports or to plan for things we want to do in the future like trying to guess what we will need or catching people who are doing something they should not be doing with our inventory_movements table and stock.

At the company level the suppliers are modeled. We use a table called product_suppliers to connect products with suppliers. This table lets us have products, from one supplier and many suppliers for one product. This is how it works in the world so we do it this way too. The product_suppliers table gives us the flexibility we need because products often have suppliers.

Bundles use a table that refers back to itself which is called the product bundles table. This table fully supports bundles that're complex and it makes sure the rules about quantity are followed. This design works with big catalogs and it checks to make sure there are no circular dependencies. The bundles system is designed to handle a number of products and it validates the relationships, between them to prevent any problems. The product bundles table is a part of this system because it allows bundles to be set up in a way that is flexible and easy to manage.

Inventory systems are data-integrity–critical, so indexes and constraints are used heavily. Unique keys prevent duplication, foreign keys prevent orphan records, and DECIMAL is used for price to avoid financial rounding errors.

Part 3: API Implementation

Let me give my assumptions for my api:

1. Each product has a low_stock_threshold column.
2. “Recent sales” means at least one sale in the last 30 days.
3. Stock is tracked per warehouse in the inventory table.
4. A product can have multiple suppliers, but the primary supplier is returned.
5. Days until stockout is calculated using average daily sales from the last 30 days.
6. Bundles do not directly maintain stock (stock belongs to child products).
7. Warehouses belong to one company.

My overview/flow- First, we validate the companies existence then we identify products with recent sales. Then we calculate a 30 day sales per product then join inventory for warehouse then filter where stock is less than threshold then attach warehouse and supplier and four then compute the estimated days till stock out and then we return a structured response.

```
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):

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

    recent_window = datetime.utcnow() - timedelta(days=30)


    recent_sales = db.session.query(
        Sale.product_id,
        func.sum(Sale.quantity).label("total_sold")
    ).join(Warehouse).filter(
        Warehouse.company_id == company_id,
        Sale.created_at >= recent_window
    ).group_by(Sale.product_id).subquery()


    results = db.session.query(
        Product,
        Inventory,
        Warehouse,
        Supplier,
        recent_sales.c.total_sold
    ).join(Inventory, Product.id == Inventory.product_id)\
     .join(Warehouse, Warehouse.id == Inventory.warehouse_id)\
     .join(product_suppliers, product_suppliers.c.product_id == Product.id)\
     .join(Supplier, Supplier.id == product_suppliers.c.supplier_id)\
     .join(recent_sales, recent_sales.c.product_id == Product.id)\
     .filter(
        Warehouse.company_id == company_id,
        Inventory.quantity < Product.low_stock_threshold
     ).all()

    alerts = []
```

```
for product, inventory, warehouse, supplier, total_sold in results:

    avg_daily_sales = total_sold / 30 if total_sold else 0
    days_left = int(inventory.quantity / avg_daily_sales) if avg_daily_sales else None

    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": product.low_stock_threshold,
        "days_until_stockout": days_left,
        "supplier": {
            "id": supplier.id,
            "name": supplier.name,
            "contact_email": supplier.contact_email
        }
    })

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