Inventory Management System for B2B SaaS

Part 1: Code Review & Debugging (30 minutes)

The issues in the given code were as follows:

1. Unsafe JSON parsing
   Issue: Uses request.json without checking if the request body is valid JSON.
   Impact: Crashes with 500 errors when JSON is missing.
   Fix: Used request.get_json(silent=True) and return 400 for invalid input.

2. Missing required field validation
   Issue: Directly accesses request fields without checking their presence.
   Impact: Causes KeyError and API crashes on bad requests.
   Fix: Validated required fields before using them.

3. Float-based price handling
   Issue: Price is accepted directly without precision or type validation.
   Impact: Leads to financial inaccuracies due to floating-point errors.
   Fix: Used decimal and validate numeric correctness.

4. Product tied to a single warehouse
   Issue: Product creation forces a warehouse_id.
   Impact: Breaks multi-warehouse support and limits scalability.
   Fix: Removed warehouse dependency from product creation.

5. No quantity validation
   Issue: Assumes initial_quantity is valid and present.
   Impact: Allows negative or invalid inventory values.
   Fix: Default quantity safely and rejected negative values.

6. Multiple database commits
   Issue: Commits product and inventory in separate transactions.
   Impact: Creates inconsistent data if one commit fails.
   Fix: Used a single transactional flow with flush() and one commit().

7. No rollback on failure
   Issue: Database session is not rolled back on errors.
   Impact: Session becomes unusable and breaks subsequent requests.
   Fix: Call db.session.rollback() on all exceptions.

8. No constraint error handling

Issue: Duplicate SKUs errors are not handled explicitly.
Impact: Clients receive 500 errors and retry unnecessarily.
Fix: Catched IntegrityError and return 409 Conflict.

CORRECTED CODE

```
from decimal import Decimal, InvalidOperation
from sqlalchemy.exc import IntegrityError

@app.route('/api/products', methods=['POST'])
def create_product():
    # fetching data
    data = request.get_json(silent=True)
    if not data:
        return {"error": "Invalid JSON body"}, 400                # if request.json is none
    # validate i/p data
    required = ["name", "sku", "price"]
    for field in required:
        if field not in data:
            return {"error": f"{field} is required"}, 400         # prevents missing or invalid data
    warehouse_id = data.get("warehouse_id")                       # products can be in multiple warehouses
    initial_quantity = data.get("initial_quantity", 0)            # takes initial quantity else default 0
    # check price value
    try:
        price = Decimal(str(data["price"]))
        if price < 0:                                             # price can't be negative
            raise ValueError
    except (InvalidOperation, ValueError):
        return {"error": "Invalid price value"}, 400

    # check quantity value
    if initial_quantity < 0:
        return {"error": "Quantity cannot be negative"}, 400

    # create product
    try:
        product = Product(
            name=data["name"],
            sku=data["sku"],
            price=price,
        )

        db.session.add(product)          # add() stages the object for insertion
        db.session.flush()               # flush() sends it to the database to generate the primary key
        db.session.commit()              # permanently saves all pending database changes
        return {
```

```
            "message": "Product created successfully",
            "product_id": product.id
        }, 201                                  # returns json response after successful creation

    except IntegrityError:
        db.session.rollback()
        return {
            "error": "SKU must be unique or warehouse does not exist"      # unique sku required
        }, 409

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

Part 2: Database Design (25 minutes)

SQL DDL used for the database schema as shown below followed by questions for the product team.
(The same is in the schema.sql in the repo)

```
-- Stores companies (customers) using the inventory management system
CREATE TABLE company (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Represents physical warehouses owned by a company
CREATE TABLE warehouse (
    id BIGSERIAL PRIMARY KEY,
    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 company(id)
        ON DELETE CASCADE
);

-- Stores product catalog information independent of warehouse stock
CREATE TABLE product (
    id BIGSERIAL PRIMARY KEY,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) NOT NULL,
    price DECIMAL(12, 2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_product_company
        FOREIGN KEY (company_id)
        REFERENCES company(id)
        ON DELETE CASCADE,

    CONSTRAINT uq_company_sku
        UNIQUE (company_id, sku)
);

-- Stores current inventory quantity of a product in a specific warehouse
CREATE TABLE inventory (
    id BIGSERIAL PRIMARY KEY,
    product_id BIGINT NOT NULL,
    warehouse_id BIGINT NOT NULL,
    quantity INTEGER NOT NULL DEFAULT 0,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

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

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

    CONSTRAINT uq_product_warehouse
        UNIQUE (product_id, warehouse_id)
);

-- Tracks every change made to inventory quantities for auditing and history
CREATE TABLE inventory_transaction (
    id BIGSERIAL PRIMARY KEY,
    product_id BIGINT NOT NULL,
    warehouse_id BIGINT NOT NULL,
    change_quantity INTEGER NOT NULL,
    reason VARCHAR(50) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_transaction_product
        FOREIGN KEY (product_id)
        REFERENCES product(id)
        ON DELETE CASCADE,

    CONSTRAINT fk_transaction_warehouse
        FOREIGN KEY (warehouse_id)
        REFERENCES warehouse(id)
        ON DELETE CASCADE
);

-- Stores suppliers that provide products to a company
CREATE TABLE supplier (
    id BIGSERIAL PRIMARY KEY,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    contact_info TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

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

-- Maps suppliers to the products they provide, including cost details
CREATE TABLE supplier_product (
    id BIGSERIAL PRIMARY KEY,
```

```
    supplier_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    supplier_sku VARCHAR(100),
    cost_price DECIMAL(12, 2),

    CONSTRAINT fk_supplier_product_supplier
        FOREIGN KEY (supplier_id)
        REFERENCES supplier(id)
        ON DELETE CASCADE,

    CONSTRAINT fk_supplier_product_product
        FOREIGN KEY (product_id)
        REFERENCES product(id)
        ON DELETE CASCADE,

    CONSTRAINT uq_supplier_product
        UNIQUE (supplier_id, product_id)
);

-- Defines bundled products made up of multiple component products
CREATE TABLE product_bundle (
    id BIGSERIAL PRIMARY KEY,
    bundle_product_id BIGINT NOT NULL,
    component_product_id BIGINT NOT NULL,
    quantity INTEGER NOT NULL,

    CONSTRAINT fk_bundle_product
        FOREIGN KEY (bundle_product_id)
        REFERENCES product(id)
        ON DELETE CASCADE,

    CONSTRAINT fk_component_product
        FOREIGN KEY (component_product_id)
        REFERENCES product(id)
        ON DELETE CASCADE,

    CONSTRAINT chk_bundle_not_self
        CHECK (bundle_product_id <> component_product_id)
);
CREATE INDEX idx_product_sku ON product (sku);
CREATE INDEX idx_inventory_product_warehouse ON inventory (product_id, warehouse_id);
CREATE INDEX idx_inventory_transaction_created_at ON inventory_transaction (created_at);
CREATE INDEX idx_supplier_product_product_id ON supplier_product (product_id);
```

Questions for the product's company:

1. Should SKU uniqueness be scoped per company?
2. Can a product belong to multiple suppliers?
3. Do we need to track who/what caused inventory changes?
4. Are inventory transfers between warehouses supported?
5. Do bundled products have fixed prices or derived prices?
6. Should deleted products and warehouses be hard-deleted?

Part 3: API Implementation (35 minutes)

Implementation of an endpoint that returns low-stock alerts for a company.

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

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
    """
    Returns low-stock alerts for a company across all warehouses.
    """
    alerts = []

    # configurable business rule
    RECENT_SALES_DAYS = 30
    since_date = datetime.utcnow() - timedelta(days=RECENT_SALES_DAYS)

    # Example threshold mapping by product type
    THRESHOLD_BY_TYPE = {
        "fast_moving": 20,
        "regular": 10,
        "slow_moving": 5
    }

    # fetch all inventory for the company
    inventories = db.session.execute("""
        SELECT
            p.id AS product_id,
            p.name AS product_name,
            p.sku,
            p.product_type,
            w.id AS warehouse_id,
            w.name AS warehouse_name,
            i.quantity
        FROM inventory i
        JOIN product p ON p.id = i.product_id
        JOIN warehouse w ON w.id = i.warehouse_id
        WHERE p.company_id = :company_id
    """, {"company_id": company_id}).fetchall()

    for row in inventories:
        threshold = THRESHOLD_BY_TYPE.get(row.product_type, 10)

        # skip if stock is not low
        if row.quantity >= threshold:
            continue

        # check recent sales activity
        recent_sales = db.session.execute("""
```

```
            SELECT 1
            FROM inventory_transaction
            WHERE product_id = :product_id
              AND warehouse_id = :warehouse_id
              AND change_quantity < 0
              AND created_at >= :since_date
            LIMIT 1
        """, {
            "product_id": row.product_id,
            "warehouse_id": row.warehouse_id,
            "since_date": since_date
        }).fetchone()

        if not recent_sales:
            continue

        # fetch supplier info
        supplier = db.session.execute("""
            SELECT s.id, s.name, s.contact_email
            FROM supplier s
            JOIN supplier_product sp ON sp.supplier_id = s.id
            WHERE sp.product_id = :product_id
            LIMIT 1
        """, {"product_id": row.product_id}).fetchone()

        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.quantity,
            "threshold": threshold,
            "days_until_stockout": None,  # calculated below
            "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 jsonify({
        "alerts": alerts,
        "total_alerts": len(alerts)
    }), 200
```

My Assumptions:

1. Each product belongs to a single company.

2. Inventory is tracked per product per warehouse.
3. Low-stock threshold is defined based on product type.
4. Recent sales activity means at least one inventory reduction in the last 30 days.
5. Inventory changes are recorded in an inventory transaction table.
6. Each product has at least one supplier for reordering.

Edge cases to consider:

1. A company has no warehouses or no inventory records.
2. A product exists but has no supplier linked.
3. Inventory quantity is exactly equal to the threshold.
4. A product has low stock but no recent sales activity.
5. A product has recent sales but stock is not low.
6. Multiple warehouses have different stock levels for the same product.
7. Inventory quantity is zero.
8. Product type is missing or unknown (threshold not defined).
9. Inventory transactions exist but are older than the “recent” window.
10. Supplier contact details are missing or incomplete.