Case Study :

Inventory Management System for B2B SaaS

Part 1: Code Review & Debugging

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

    # Create new product
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=data['price'],
        warehouse_id=data['warehouse_id']
    )

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

    # Update inventory count
    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=data['initial_quantity']
    )

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

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

Issues and Impact

API/ Input Error

1. **Issue:** Missing or invalid request data.
   Impact: The API may throw KeyError/TypeError causing server crashes.
   fixes:

```
# Type Error
if not request.is_json:
    return {"error": "Invalid, Missing"}, 400

data = request.get_json()

# key Error
required_fields = ["name", "sku", "price", "warehouse_id",
"initial_quantity"]

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

2. **Issue**: Required input fields are not validated using patterns or regular expressions.
   Impact: invalid fields will get accepted by the API.
   fixes:

```
# Type Validation
# for product name
if not isinstance(data.get("name"), str) or not
data["name"].strip():
    return {"error": "Invalid product name"}, 400
# for product id
if not isinstance(data.get("warehouse_id"), int) or
data["warehouse_id"] <= 0:
    return {"error": "Invalid warehouse_id"}, 400

# regex (pattern validation)
if not re.match(r"^[A-Z0-9_-]+$", data["sku"]):
    return {"error": "Invalid SKU format"}, 400
```

Database & Transaction Errors

1. **Issue:** Product and inventory creation are committed in separate transactions.
   Impact: If one operation succeeds and the other fails, incomplete or inconsistent data may be stored
   fixes : Use a single atomic transaction

```
try:
    product = Product(
        name=data["name"],
        sku=data["sku"],
        # remove warehouse id to normalize the data
        price=data["price"]
    )
    db.session.add(product)
    db.session.flush()  # without commit get product id preferred
    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data["warehouse_id"],
        quantity=data["initial_quantity"]
    )
    db.session.add(inventory)
# single atomic commit
    db.session.commit()
```

2. **Issue:** No database error handling such as abort, rollback, or transaction management.
   Impact: ACID properties are not preserved, leading to data inconsistency.
   fixes : added rollback for failures

```
except Exception:
    db.session.rollback()
    return {"error": "Database operation failed"}, 500
```

3. **Issue:** Database constraints such as UNIQUE and NOT NULL are not enforced.
   Impact: Duplicate or null values can be written to the database.
   Fixes: use unique nullable

```
# Example Unique constraint
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    sku = db.Column(db.String, unique=True, nullable=False)
```

4. **Issue** :Quantity constraints are not checked
   Impact: Invalid inventory quantities can be stored in the database

Integrity Issues

1. **Issue:** Product is directly associated with a single warehouse
   impact:The same product cannot exist in multiple warehouses, violating normalization principles.
   Fixes : use 3NF to normalize the database model
   ```
   # product
   class Product(db.Model):
       id = db.Column(db.Integer, primary_key=True)
       name = db.Column(db.String, nullable=False)
       sku = db.Column(db.String, unique=True, nullable=False)
   # inventory
   class Inventory(db.Model):
       id = db.Column(db.Integer, primary_key=True)
       product_id = db.Column(db.Integer,
   db.ForeignKey("product.id"), nullable=False)
       warehouse_id = db.Column(db.Integer,
   db.ForeignKey("warehouse.id"), nullable=False)
       quantity = db.Column(db.Integer, nullable=False)
   ```

2. **Issue:** Inventory quantity values are not validated.
   Impact: Can store negative values
   Fixes: give condition quantity >= 0
3. **Issue:** Foreign key references are not validated
   Impact: Inventory records may reference non-existent warehouse or product records.
   fixes : validate foreign key references
   ```
   warehouse = Warehouse.query.get(data["warehouse_id"])
   if not warehouse:
       return {"error": "Warehouse not found"}, 404
   ```

Part 2: Database Design

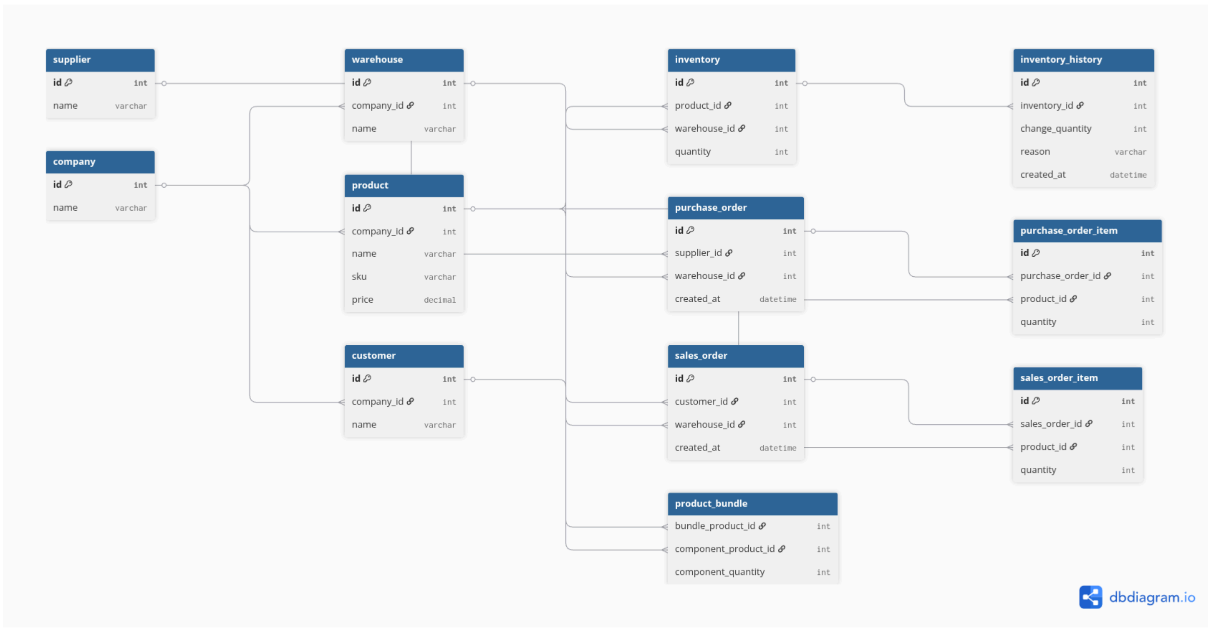
Given Requirements:

- Companies can have multiple warehouses
- Products can be stored in multiple warehouses with different quantities
- Track when inventory levels change
- Suppliers provide products to companies
- Some products might be "bundles" containing other products

Tasks:

- **Design Schema**: Create tables with columns, data types, and relationships
- **Identify Gaps**: List questions you'd ask the product team about missing requirements
- **Explain Decisions**: Justify your design choices (indexes, constraints, etc.)

DESIGN SCHEMA:



Relationships

1. Suppliers deliver products to warehouses through purchase orders (inbound).
2. Customers receive products from warehouses through sales orders (outbound).
3. Inventory tracks product quantities for each warehouse.
4. Inventory history records all inventory level changes.
5. Product bundles contain multiple products.

Gaps In the Requirements

1. SKU uniqueness is not clearly defined (whether it should be global or per company).
2. It is not specified whether a company can work with multiple suppliers.
3. The inventory lifecycle is unclear (RAW, WIP, FOR_SALE states are not defined).
4. The inventory flow between supplier → company and company → customer is not specified.
5. Inventory reservation rules and how inventory levels should be managed are not defined.
6. The location of products and how product bundles are stored or handled in inventory is not specified.
7. It is unclear whether product bundles consist of the same product or different products.

Design Decisions for gaps:

1. Enforced SKU uniqueness per company to handle unclear global uniqueness requirements.
2. Added supplier and purchase order tables to support multiple suppliers.
3. Introduced inventory lifecycle states (RAW, WIP, FOR_SALE).
4. Used purchase orders for supplier-to-company flow and sales orders for company-to-customer flow.
5. Added inventory history and reservations to track and control inventory levels.
6. Represented product bundles using relationships between products (product-to-product foreign keys).
7. Added indexes to improve query performance.

Part 3: API Implementation (35 minutes)

Expected Response Format:

```
{
  "alerts": [
    {
      "product_id": 123,
      "product_name": "Widget A",
      "sku": "WID-001",
      "warehouse_id": 456,
      "warehouse_name": "Main Warehouse",
      "current_stock": 5,
      "threshold": 20,
      "days_until_stockout": 12,
      "supplier": {
        "id": 789,
        "name": "Supplier Corp",
        "contact_email": "orders@supplier.com"
      }
    }
  ],
  "total_alerts": 1
}
```

Code :

```
from flask import Flask, request, jsonify
import sqlalchemy
import os
app = Flask(__name__)
DATABASE_URI = os.environ.get(
   'DATABASE_URL',
'postgresql+psycopg2://postgres:<user_password>@localhost:5432/inventor
y_db')

# used sqlalchemy to execute query using postgresql database its ordbms
and sqlalchemy is orm for python
engine = sqlalchemy.create_engine(DATABASE_URI)


@app.route('/api/companies/<int:company_id>/alerts/low-stock',
methods=['GET'])
def get_low_stock(company_id):
   # connect to the database using the engine
   with engine.connect() as connection:
       # --- SQL Query for Low Stock Products ---
       # This query joins the inventory, product, warehouse, and
supplier tables
       # to gather all necessary information for the low stock alerts.
       # just check if current_sstock is less than or equal to
threshold this will give low stock products
       # after that return the result by taking length of alerts
       query = """
       SELECT
           p.id AS product_id,
           p.name AS product_name,
           p.sku,
           w.id AS warehouse_id,
           w.name AS warehouse_name,
           i.quantity AS current_stock,
           i.threshold,
           -- Build a JSON object for supplier details to nest the
information.
           json_build_object(
               'id', s.id,
               'name', s.name,
               'contact_email', s.contact_email
           ) AS supplier
       FROM inventory i
       JOIN product p ON i.product_id = p.id
       JOIN warehouse w ON i.warehouse_id = w.id
       JOIN supplier s ON p.supplier_id = s.id
       WHERE w.company_id = :company_id AND i.quantity <= i.threshold
       """
       result = connection.execute(sqlalchemy.text(query), {
                                   "company_id": company_id})

       alerts = [dict(row._mapping) for row in result]

       return jsonify({'alerts': alerts, 'total_alerts': len(alerts)}),
200


if __name__ == '__main__':
   app.run(debug=True)
```