Bynry Inc.

Backend Intern Assessment Report

Date: 02 August 2025

By: Vijay Bahadur Vishwakarma

Approach and Methodology

To complete the Backend Intern Assessment, I followed a systematic approach broken into the following phases:

1. Understanding the Problem Statement

- Carefully reviewed all three parts of the assessment: Code Debugging, Database Design, and API Implementation.
- Noted business constraints like multi-warehouse support, supplier-product relationships, and dynamic inventory levels.

2. Identifying Gaps and Making Assumptions

- Where the specification was incomplete (e.g., how bundle products are handled or if product thresholds are static), I documented open questions and made reasonable assumptions.
- Assumed normalized schema and realistic business behaviours (e.g., average sales to estimate stockout days).

3. Problem Solving and Implementation

- Broke each section into core issues or goals.
- Used templates and best practices for code review (atomicity, exception handling, schema design).
- For the API, wrote optimized SQL logic with proper joins, grouping, and conditional aggregation to meet the exact JSON format required.

4. Error Handling and Scalability Consideration

- Applied try/except/finally blocks to ensure fault tolerance in API logic.
- Used groupings and aggregate functions in SQL to ensure that the system could scale to multiple warehouses and suppliers efficiently.

5. Review and Structuring

- Structured the report for clarity: Problem → Analysis → Fix/Design → Result.
- Ensured formatting matched expectations, and returned expected outputs in structured JSON.

Section 1. Code Review & Debugging

Assumptions:- Assuming *all the data to this step is in correct format and validated at the time of Input*. And we have currently, two data tables in our database as listed below:

1. Product (name, sku, price, warehouse_id) [INT(PK), VARCHAR(7), DECIMAL(12,2), INT(FK)]
2. Inventory (product_id, warehouse_id, quantity) [INT(FK), INT(FK), INT]

By our assumption, what we solved:

- Price can be decimal.
- SKUs will be unique while filling data.

Issue with the code,

Issues	Impact Analysis	Fixes

Wrong Business Logic for Inventory	Impact: Critical
	
	Explanation: Instead of Updating the Inventory, whenever a Product is added, a Product is initialized with a new Inventory without taking any consideration about pre-existence of Product in the Inventory.
		Template:
		
		Product_id = get_from_query(data[name])
		If product_id in Inventory:
		quantity = Inventory[quantity] + Product[quantity]
		Inventory[quantity] = quantity
		Else:
		inventory = Inventory(
		....same code used to initiate inventory but with change quantity = Product[quantity]....)

Incorrect Quantity Counting Logic	Impact: Critical Explanation: The quantity inside the Update Inventory Count part is calculated incorrectly and is just replaced by the new quantity, irrespective to the previous presence of item in inventory.	Template: inventory = Inventory(....same code used to initiate inventory but with change quantity = Product[quantity]....)
Lack of Atomicity	Impact: High	Template: {

	Explanation: Multiple session commit is made in the same code. So, if there occurs some error in between such that the second db.session.commit() fails then the data becomes inconsistent.	..Create new product section as given.. ..Remove the db.session add and commit.. ..Update Inventory count.. ..this will be the final and only db.session add and commit.. }
No Input Validation & Exception Handling	Impact: Medium **Explanation:** Nothing to make sure that the data is surely a JSON format-type input. And we didn't handle any exception that might occur like "No JSON found"	**Template:** Try: Data = request.json Except e as error: Print("Error occurred") {..other part of code..}

Section 2. Database Design

As per the growing requirements, we need to update our assumed database appropriately.

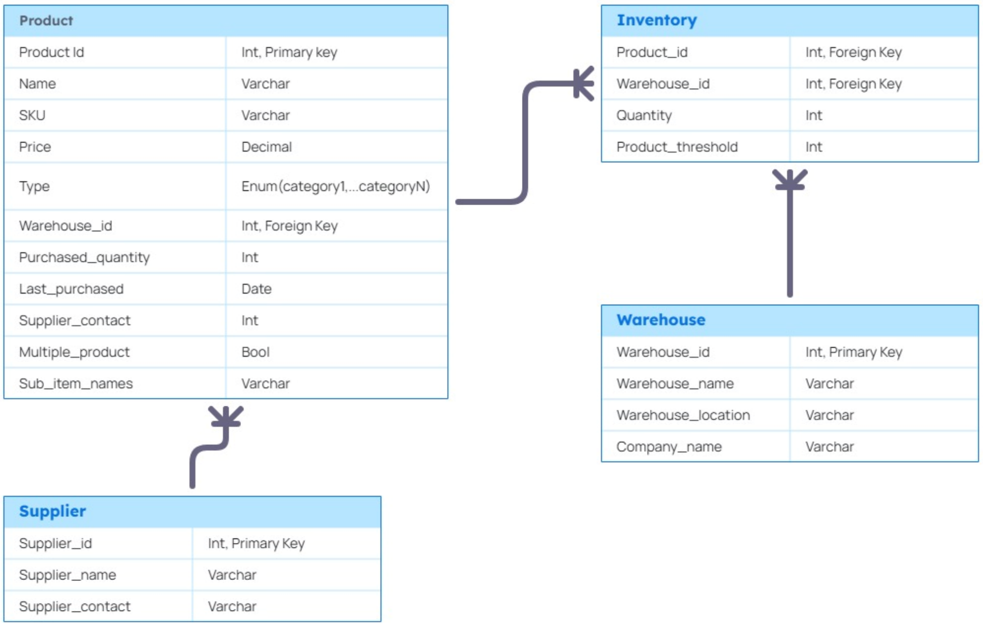
Designed Schemas for Database:

1. Product
    - *a.* Product_Id *(INT[PK])*
    - *b.* Name *(VARCHAR(35))*
    - *c.* SKU *(VARCHAR(7))*
    - *d.* Price *(DECIMAL(12,2))*
    - e. Type *(ENUM(Catg1, Categ2,…))*
    - f. Warehouse_id *(INT[FK])*
    - *g.* Purchased_quantity *(INT)*
    - *h.* Last_purchase *(DATE)*
    - *i.* Supplier_contact *(INT(10))*
    - *j.* Multiple_product *(BOOL)*
    - *k.* Sub_item_name *(VARCHAR(65))*
2. Inventory
    - *a.* Product_id *(INT[FK])*
    - b. Warehouse_id *(INT[FK])*
    - c. Quantity *(INT)*

    d. Product_threshold *(INT)*

3. Warehouse [Company Name with registered Warehouses]
    a. Warehouse_id *(INT[PK])*
    b. Name *(VARCHAR(30))*
    c. Company_name *(VARCHAR(30))*
    d. Warehouse_location *(VARCHAR(40))*
4. Supplier
    a. Supplier_id *(INT[PK])*
    b. Supplier_Name *(VARCHAR(27))*
    c. Contact *(INT(10))*

ER-Diagram



Product	
Product Id	Int, Primary key
Name	Varchar
SKU	Varchar
Price	Decimal
Type	Enum(category1,...categoryN)
Warehouse_id	Int, Foreign Key
Purchased_quantity	Int
Last_purchased	Date
Supplier_contact	Int
Multiple_product	Bool
Sub_item_names	Varchar

Inventory	
Product_id	Int, Foreign Key
Warehouse_id	Int, Foreign Key
Quantity	Int
Product_threshold	Int

Warehouse	
Warehouse_id	Int, Primary Key
Warehouse_name	Varchar
Warehouse_location	Varchar
Company_name	Varchar

Supplier	
Supplier_id	Int, Primary Key
Supplier_name	Varchar
Supplier_contact	Varchar

Gaps:

1. Can we count bundle as a single product, because it might be used to sell as a single product like gifts?
2. Will the product_threshold be static or dynamic (that is can be change) with respect to time for a warehouse?
3. For a company, with multiple warehouses, can a product have multiple product_threshold with respect to warehouses.

Decision:

It is typical to explain why the data schema is made this specific. All the data types and constraints are chosen in order to achieve the following:

1. Less Redundancy [unique ids]
2. Correlation [common attributes become foreign key for other]
3. Space Efficient [occupy less space to be efficient]

Special Consideration:

1. Size allotted in VARCHAR is sufficient for data.
2. Enum for “Type” will contain categorical data like “Beverage”, “Fitness”, “Edible”, etc.
3. One additional column we can add in Product Table is “Bill Ref.” of “BLOB” type date item, this can store a screenshot of bill.

Section 3. API Implementation (on Python / Flask)

Implementation:

```
@app.route(‘/api/companies/<int: company_id>/alert/lowstock’, method=[“GET”])

def get_allert(company_id):

  # Try/Except/Final format to handle error

  try:


    # Establishing Connection to run Query and Fetch required detail at Backend


    connection=pymysql.connect(….Connection Details….)

    connection=connection.cursor(pymysql.cursors.DictCursor)


    # Query to Fetch Data from Database


    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,
       p.threshold,
       s.id AS supplier_id,
       s.name AS supplier_name,
       s.contact_email,
       -- Estimate days_until_stockout (simplified logic)
       CASE
           WHEN AVG(sa.quantity_sold) IS NULL OR AVG(sa.quantity_sold) = 0 THEN NULL
           ELSE ROUND(i.quantity / AVG(sa.quantity_sold))
       END AS days_until_stockout
     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_contact = s.contact_email
     LEFT JOIN sales sa ON sa.product_id = p.id AND sa.sale_date >= CURDATE() - INTERVAL 30 DAY
     WHERE w.company_id = %s AND i.quantity < p.threshold
     GROUP BY p.id, w.id
     """

    # Execute Query
     cursor.execute(query, (company_id,))

    # Get items in results
```

```
   results = cursor.fetchall()

   # Create alerts from the results row

    alerts = []
    for row in results:
        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["current_stock"],
            "threshold": row["threshold"],
            "days_until_stockout": row["days_until_stockout"],
            "supplier": {
                "id": row["supplier_id"],
                "name": row["supplier_name"],
                "contact_email": row["contact_email"]
            }
        })

response = {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }
```

```
 # Returns result

    return jsonify(response), 200

# Exception Handling for any Error

except Exception as e:
    print("Error:", e)
    return jsonify({"error": "Internal server error"}), 500

# Finally close the established connection

finally:
    cursor.close()
    connection.close()
```

~Thanking You~