CASE STUDY

Candidate: Yash Parshetty
Role: Backend Engineering Intern

Part 1: Code Review & Debugging

1.1 Indentify Issues :

- **No Input Validation**
- **Price Has Float (Not Decimal)**
- **Multiple DB commits**
- **No Transaction**
- **No Error Handling**
- **Uniqueness of SKU not checked**

1.2 Impact of Issues :

- **No Input Validation -**

If any field is missing it will throw KeyError which might cause problem at server side and client gets the some invalid data.

- **Price Has Float (Not Decimal)-**

It may create problem in rounding the prices ,which may give incorrect prices to client .

- **Multiple DB commits-**

In the code given the db is committed twice which leads to the low performance because the DB will have to work more and if anything breaks it will cause problem to all the things.

- **No Transaction-**

The “Product” is created before “Inventory” which will cause problem as if the “Inventory” commit fails , then the “Product” will not be stored.

- **No Error Handling-**

If there is any DB error then it leads to API server crash , which causes poor reliability.

- **Uniqueness of SKU not checked-**

SKU(Stock Keeping Unit) should be kept unique as it might get wrong data fetched which will further go to operational failures.

1.3 Fixes :

- **CODE :**

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

    name = data.get('name')
    sku = data.get('sku')
    price_raw = data.get('price')
    initial_quantity = data.get('initial_quantity', 0)
    company_id = data.get('company_id')

    if not name or not sku:
        return {"error": "name and sku are required"}

    try:
        price = Decimal(str(price_raw)) if price_raw is not None else None
        if price is not None and price < 0:
            return {"error": "price must be non-negative"},
    except (InvalidOperation, TypeError):
        return {"error": "invalid price format"},

    try:
        with db.session.begin():
            existing = Product.query.filter_by(sku=sku,
company_id=company_id).first()
            if existing:
                return {"error": "SKU already exists", "product_id": existing.id},

            product = Product(
                name=name,
                sku=sku,
                company_id=company_id,
                price=price
            )
            db.session.add(product)
            db.session.flush()

            if warehouse_id is not None:
                inv_q = Inventory.query.filter_by(product_id=product.id,
warehouse_id=warehouse_id)
                try:
                    inv = inv_q.with_for_update().first()
                except Exception:
                    inv = inv_q.first()

                if inv:
```

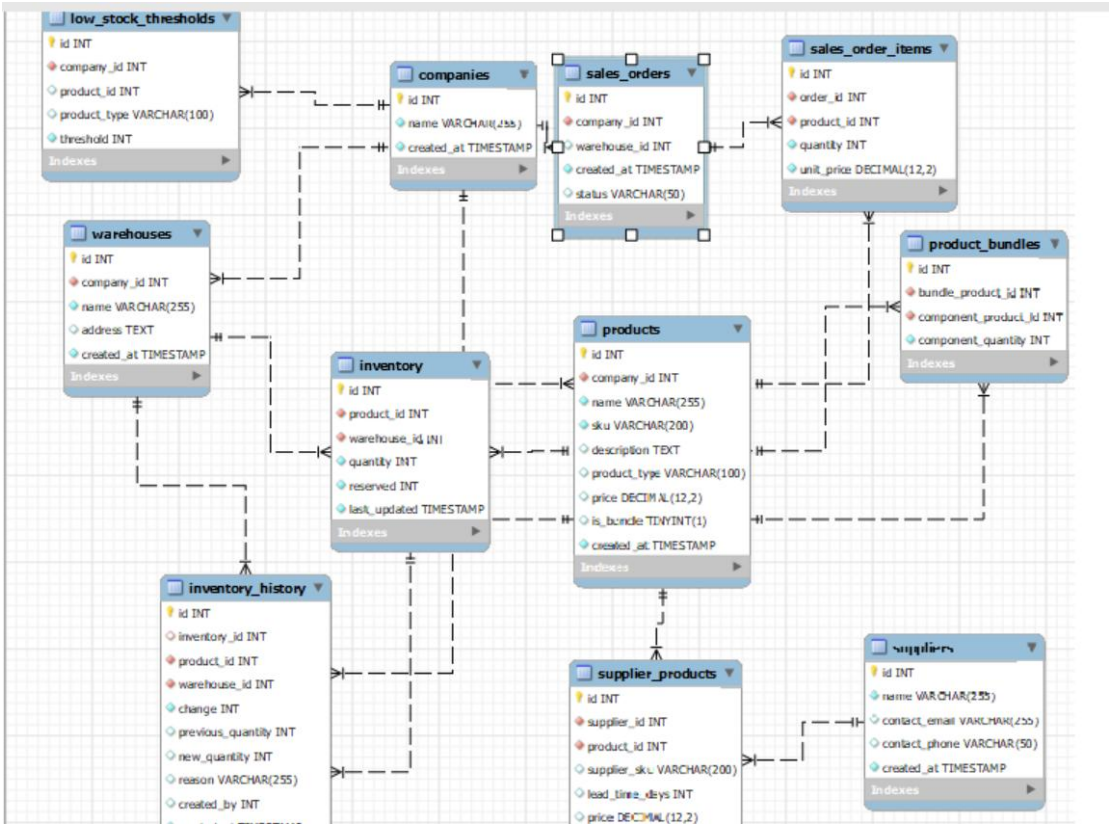
```
                inv.quantity = initial_quantity
            else:
                inv = Inventory(
                    product_id=product.id,
                    warehouse_id=warehouse_id,
                    quantity=initial_quantity
                )
                db.session.add(inv)

    except IntegrityError as e:
        db.session.rollback()
        return {"error": "database integrity error", "detail": str(e)},
    except Exception as e:
        db.session.rollback()
        return {"error": "server error", "detail": str(e)},

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

- **EXPLANATION:**

I. Used **data.get()** to validate the inputs which fixed the KeyError problem which was leading to the server crash on missing Keys.
II. **Converted Price from Float to Decimal** and also added prices validation which will ensure the accuracy/
III. **db.session.begin()** was used to handle the transaction as firstly the product and inventory firstly were created in two different commits which was causing problem of slow performance.
IV. **SKU uniqueness** is now checked before inserting as there should be a unique SKU across the company.
V. Added proper try/except error handling which will prevent the API crashes and will make debugging easier too.

Part 2: Database Design :

2.1 Design Schema :



- **TABLES and their RELATIONSHIP -**

I. Companies

- companies (1) → (N) warehouses via warehouses.company_id
- companies (1) → (N) products via products.company_id
- companies (1) → (N) sales_orders via sales_orders.company_id
- companies (1) → (N) low_stock_thresholds via low_stock_thresholds.company_id

II. Warehouses

- warehouses (N) ← (1) companies via warehouses.company_id
- warehouses (1) → (N) inventory via inventory.warehouse_id
- warehouses (1) → (N) inventory_history via inventory_history.warehouse_id
- warehouses (1) → (N) sales_orders via sales_orders.warehouse_id

III. Products

- products (N) ← (1) companies via products.company_id
- products (1) → (N) inventory via inventory.product_id
- products (1) → (N) inventory_history via inventory_history.product_id
- products (1) → (N) sales_order_items via sales_order_items.product_id
- products (1) → (N) supplier_products via supplier_products.product_id
- products (1) → (N) product_bundles (as bundle owner) via product_bundles.bundle_product_id
- products (1) → (N) product_bundles (as component) via product_bundles.component_product_id

IV. Inventory

- inventory (N) ← (1) products via inventory.product_id
- inventory (N) ← (1) warehouses via inventory.warehouse_id
- inventory (1) → (N) inventory_history via inventory_history.inventory_id

V. Inventory History

- inventory_history (N) ← (1) inventory via inventory_history.inventory_id
- inventory_history (N) ← (1) products via inventory_history.product_id
- inventory_history (N) ← (1) warehouses via inventory_history.warehouse_id

VI. Suppliers

- suppliers (1) → (N) supplier_products via supplier_products.supplier_id

VII. Supplier Products (junction table)

- supplier_products (N) ← (1) suppliers via supplier_products.supplier_id
- supplier_products (N) ← (1) products via supplier_products.product_id

VIII. Product Bundles (self-referential junction table)

- product_bundles (N) ← (1) products (as bundle owner) via product_bundles.bundle_product_id
- product_bundles (N) ← (1) products (as component) via product_bundles.component_product_id

IX. Sales Orders

- sales_orders (N) ← (1) companies via sales_orders.company_id
- sales_orders (N) ← (1) warehouses via sales_orders.warehouse_id
- sales_orders (1) → (N) sales_order_items via sales_order_items.order_id

X. Sales Order Items

- sales_order_items (N) ← (1) sales_orders via sales_order_items.order_id
- sales_order_items (N) ← (1) products via sales_order_items.product_id

XI. Low Stock Thresholds

- low_stock_thresholds (N) ← (1) companies via low_stock_thresholds.company_id

2.2 Questions to Ask the Product Team About Missing Requirements :

1. Should SKU values be unique globally or only within a company?
2. How is “recent sales activity” defined for low-stock alerts?
3. Do low-stock thresholds apply per product, per product type, or per warehouse?
4. When a bundle product is sold, should inventory be reduced for each component item automatically?
5. Can different warehouses have different reorder thresholds for the same product?
6. Should supplier selection be automatic (based on lead time) or explicitly chosen by the user?
7. What should happen when a product has no inventory record in a warehouse?
8. Do we need to support returns/cancellations and reflect them in inventory history?

2.3 Decisions :

- **Products ↔ Inventory table**: Product has no warehouse_id. Inventory table models per-warehouse quantity. UNIQUE(product_id, warehouse_id) prevents duplicate inventory rows.

- **SKU unique constraint:** UNIQUE(company_id, sku) ensures uniqueness within company. If SKUs must be global, change to UNIQUE(sku).

- **Inventory history:** Audit trail for tracking who/when/why inventory changed (business requirement).

- **Suppliers & supplier_products table:** Allows multiple suppliers per product, stores supplier lead time and supplier-specific SKU & price for reorders.

- **Low_stock_thresholds**: Keep flexible — threshold can be defined per product or per product_type.

- **Sales tables:** Required to compute “recent sales activity” and average daily sales to estimate days until stockout.

- **Indexes**: Added on columns commonly used in filters (company_id, product_id, created_at) for performance.

Part 3: API Implementation

3.1 CODE -

```
from flask import Blueprint, jsonify
from sqlalchemy import func
from math import ceil
from datetime import datetime, timedelta

from app import db
from models import (
    Company, Warehouse, Product, Inventory,
    SalesOrder, SalesOrderItem, Supplier, SupplierProducts, LowStockThreshold
)

alerts_bp = Blueprint('alerts', __name__)

# Configuration: can be adjusted by product/ops team
RECENT_DAYS = 30                    # Lookback window for “recent sales”
DEFAULT_THRESHOLD = 10              # Default low-stock threshold if none specified

@alerts_bp.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
    """
    Computes low-stock alerts for all products belonging to a company.
    Logic:
      - Only include products that had sales in the past RECENT_DAYS.
      - Combine inventory data with product info and thresholds.
      - Compare current stock with the appropriate threshold.
      - Estimate stockout time based on average daily sales.
      - Attach supplier information to help with reordering.
    """

    # Step 1: Check that the company exists to avoid invalid requests
    company = db.session.get(Company, company_id)
    if not company:
        return jsonify({"error": "Company not found"}), 404

    # Step 2: Determine the date range for "recent sales"
    end_date = datetime.utcnow()
    start_date = end_date - timedelta(days=RECENT_DAYS)

    # Step 3: Aggregate total quantity sold per product within the recent window
    recent_sales_rows = (
        db.session.query(
```

```
            SalesOrderItem.product_id,
            func.sum(SalesOrderItem.quantity).label('qty_sold')
        )
        .join(SalesOrder, SalesOrder.id == SalesOrderItem.order_id)
        .filter(
            SalesOrder.company_id == company_id,
            SalesOrder.created_at >= start_date,
            SalesOrder.created_at <= end_date,
            SalesOrder.status.in_(('placed', 'shipped', 'completed'))
        )
        .group_by(SalesOrderItem.product_id)
        .all()
)

# Convert sales rows into a usable lookup structure
sales_map = {row.product_id: float(row.qty_sold) for row in recent_sales_rows}

# If there were no recent sales, there cannot be any low-stock alerts
if not sales_map:
    return jsonify({"alerts": [], "total_alerts": 0})

product_ids = list(sales_map.keys())

# Step 4: Fetch inventory for all warehouses containing these products
# Includes product & warehouse names for response readability
inventories = (
    db.session.query(
        Inventory.product_id,
        Inventory.warehouse_id,
        Inventory.quantity,
        Product.name.label('product_name'),
        Product.sku.label('sku'),
        Product.product_type.label('product_type'),
        Warehouse.name.label('warehouse_name')
    )
    .join(Product, Product.id == Inventory.product_id)
    .join(Warehouse, Warehouse.id == Inventory.warehouse_id)
    .filter(
        Product.company_id == company_id,
        Inventory.product_id.in_(product_ids)
    )
    .all()
)

alerts = []

# Step 5: Evaluate each (product, warehouse) pair for low-stock
for inv in inventories:
    pid = inv.product_id
    wid = inv.warehouse_id
```

```
        current_stock = int(inv.quantity or 0)

        # Compute average daily sales from total recent sales
        qty_sold = sales_map.get(pid, 0.0)
        avg_daily_sales = qty_sold / RECENT_DAYS

        # Zero or negative average daily sales indicates no meaningful activity
        if avg_daily_sales <= 0:
            continue

        # Step 6: Determine the applicable threshold
        # Priority:
        #     1. Product-specific threshold
        #     2. Product-type threshold
        #     3. Default system threshold
        threshold_row = (
            db.session.query(LowStockThreshold)
            .filter(
                LowStockThreshold.company_id == company_id,
                LowStockThreshold.product_id == pid
            )
            .first()
        )

        if threshold_row:
            threshold = int(threshold_row.threshold)
        else:
            if inv.product_type:
                type_row = (
                    db.session.query(LowStockThreshold)
                    .filter(
                        LowStockThreshold.company_id == company_id,
                        LowStockThreshold.product_type == inv.product_type
                    )
                    .first()
                )
                threshold = int(type_row.threshold) if type_row else
DEFAULT_THRESHOLD
            else:
                threshold = DEFAULT_THRESHOLD

        # Skip if stock level is adequate
        if current_stock >= threshold:
            continue

        # Step 7: Choose the supplier with the shortest lead time
        supplier_entry = (
            db.session.query(Supplier, SupplierProducts)
            .join(SupplierProducts, SupplierProducts.supplier_id == Supplier.id)
            .filter(SupplierProducts.product_id == pid)
```

```
                .order_by(func.coalesce(SupplierProducts.lead_time_days,
999999).asc())
                .first()
            )

            supplier_info = None
            if supplier_entry:
                supplier = supplier_entry[0]
                supplier_info = {
                    "id": supplier.id,
                    "name": supplier.name,
                    "contact_email": supplier.contact_email
                }

            # Step 8: Estimate when product will run out of stock
            try:
                days_until_stockout = ceil(current_stock / avg_daily_sales)
            except Exception:
                days_until_stockout = None

            # Step 9: Construct an alert entry for this item
            alert = {
                "product_id": pid,
                "product_name": inv.product_name,
                "sku": inv.sku,
                "warehouse_id": wid,
                "warehouse_name": inv.warehouse_name,
                "current_stock": current_stock,
                "threshold": threshold,
                "days_until_stockout": days_until_stockout,
                "supplier": supplier_info
            }
            alerts.append(alert)

        # Step 10: Return summary result
        return jsonify({"alerts": alerts, "total_alerts": len(alerts)})
```

3.1 Edge Cases -

1. Division by Zero in Average Daily Sales

If RECENT_DAYS is set incorrectly (e.g., 0) or total sales is 0,
avg_daily_sales = qty_sold / RECENT_DAYS will fail.

Impact: API crashes, no alerts returned.
Fix: Validate RECENT_DAYS >= 1 and skip products with zero sales safely.

2. No Recent Sales but Stock Critically Low

The business rule says "only alert for products with recent sales activity."
Some items may be essential but have no recent orders.

Impact: System misses critical low-stock items.
Fix: Make “recent-sales-only” alerting configurable, or add absolute minimum alerts.

3. Missing or Incomplete Inventory Rows

Some products may not have an inventory record for a warehouse yet.

Impact: Crashes when accessing fields; incorrect assumption of stock.
Fix: Treat missing inventory as 0 or auto-create inventory rows on product creation.

4. Supplier Data Missing or Lead Time NULL

A product may have no supplier assigned or NULL lead times.

Impact: No reorder info in alerts; wrong supplier chosen.
Fix: Use COALESCE(lead_time_days, large_number) and allow null-safe sorting; optionally enforce at least one supplier per active product.

5. Cross-Tenant Data Leakage Due to Missing company_id Filters

If a query forgets to filter by company, alerts may show products/inventory from another company**.**

Impact: Severe security violation.
Fix: Always include Product.company_id == company_id and warehouse scoping in every query.