Bynyr - Assignment

By Parth Khandelwal

Part 1:

My Observations and Reasons for updates:

- The first thing I noticed is that the queries(saving the product details and then updating the inventory) in the initial snippet were made consecutively with separate commits, without considering what would happen if the first query succeeded but the second failed (or vice versa). This would create inconsistencies in the database.
- This immediately raised a red flag about the lack of transaction handling .Both operations (product + inventory) should either succeed or fail .

- There is no validation of incoming request data if we actually are receiving something in JSON or it's just null. Directly accessing fields like `data['name']` can cause errors if the fields are missing.

- The snippet ties warehouse_id directly to product creation, but a product can exist in multiple warehouses. Inventory, not product, should carry information about which warehouse has how much quantity of product.

- SKU uniqueness is not enforced. Without this check, duplicate SKUs can exist across the system, leading to big business logic problems.

- Price validation is not there which could cause slip up of strings even negative values.

- Initial quantity handling is assumed mandatory, but I think it should be optional as sometimes we can just create a product without actually stocking it in any warehouse .

Corrected code :

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json()
    if not data:
        return jsonify({"error": "Invalid JSON"}), 400

    # Required field validation as it was not in the initial snippet which could cause issues where we are trying to access data but it actually have nothing in it .
    required_fields = ['name', 'sku', 'price']
    for field in required_fields:
        if field not in data:
            return jsonify({"error": f"Missing field: {field}"}), 400

   # validating price to avoid string or negative values to go through
    try:
        price = float(data['price'])
        if price < 0:
            raise ValueError()
    except (ValueError, TypeError):
        return jsonify({"error": "Invalid price format"}), 400

    # checking SKU uniqueness as multiple products cannot have same SKUs
    existing = Product.query.filter_by(sku=data['sku']).first()
    if existing:
        return jsonify({"error": "SKU already exists"}), 409

    # Creating product (without trying to stock in any warehouse directly) and not commit it at this point
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=price
    )
    db.session.add(product)

    try:
        db.session.flush()  # Get product.id without committing yet
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": f"Failed to create product: {str(e)}"}), 500
```

```
    # Optional inventory handling only when warehouse_id is provided as I already discussed it might not be necessary that we stock an item exactly when we have created it in products database

    if 'warehouse_id' in data and 'initial_quantity' in data:
        try:
            quantity = int(data['initial_quantity'])
            if quantity < 0:
                raise ValueError()
        except (ValueError, TypeError):
            db.session.rollback()
            return jsonify({"error": "Invalid initial quantity"}), 400

        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=quantity
        )
        db.session.add(inventory)

    try:
        db.session.commit()
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": f\"Database error during commit: {str(e)}\"}), 500

    return jsonify({
        "message": "Product created",
        "product_id": product.id
    }), 201
```

Assumptions for Part 1 :

1. `I am assuming to take` initial_quantity as optional param .
2. A product can exist in multiple warehouses (handled via `Inventory`).
3. SKU uniqueness is mandatory across the entire system ,so that we don't end up creating products with the same SKU.
4. Price must be non-negative and stored in a proper numeric/decimal field at the DB.
5. Product creation and inventory creation must happen atomically within one transaction.

Part 2

My Observations and Reasons for updates:

- The requirements seem to be clear about companies, warehouses, products, suppliers, inventory levels, and even bundled products but tracking can be a bit more to crack.
- Suppliers provide products, but is that one supplier per product, or many suppliers can supply the same product? In reality, it's many-to-many.

- If I go with flat inventory then we could directly trace the inventory movements but we often need a check that if the relation between suppliers and products is many-to-many then we need to trace which batch of products was supplied by which supplier,at which rate , on what date and etc.

- So what's missing is also how we can track changes in inventory: do we need full audit logs, reasons for changes (sale, purchase, adjustment), or just timestamps?

- For Bundles we can either bundle a product to its components' product (self-referencing relationship) or treat a bundle as a product in itself.

Database Design code :

```
        -- Companies
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT now()
);

-- Warehouses
CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
```

```
    company_id INT NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    location TEXT,
    created_at TIMESTAMP DEFAULT now()
);

-- Products
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) UNIQUE NOT NULL,
    price NUMERIC(10,2), -- base/list price (not necessarily purchase cost)
    low_stock_threshold INT DEFAULT 10,
    is_bundle BOOLEAN DEFAULT false,
    created_at TIMESTAMP DEFAULT now()
);

-- Suppliers
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    company_id INT NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    contact_email VARCHAR(255),
    contact_info JSONB,
    created_at TIMESTAMP DEFAULT now()
);

-- Supplier–Product Catalog
-- (represents that supplier can provide product, at typical price, with lead time)
CREATE TABLE supplier_products (
    id SERIAL PRIMARY KEY,
    supplier_id INT NOT NULL REFERENCES suppliers(id) ON DELETE CASCADE,
    product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    base_price NUMERIC(10,2), -- negotiated unit cost
    lead_time_days INT DEFAULT 7,
    UNIQUE (supplier_id, product_id)
);

  –  inventory
CREATE TABLE inventory (
    id SERIAL PRIMARY KEY,
    product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    warehouse_id INT NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
    total_quantity INT NOT NULL DEFAULT 0 CHECK (total_quantity >= 0),
```

```
    updated_at TIMESTAMP DEFAULT now(),
    UNIQUE (product_id, warehouse_id)
);



-- Inventory Batches
-- (instead of one inventory row per product-warehouse,
--  we store per-batch stock with supplier info)
CREATE TABLE inventory_batches (
    id SERIAL PRIMARY KEY,
    product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    warehouse_id INT NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
    supplier_id INT NOT NULL REFERENCES suppliers(id) ON DELETE SET NULL,
    purchase_price NUMERIC(10,2) NOT NULL,
    quantity INT NOT NULL CHECK (quantity >= 0),
    received_at TIMESTAMP DEFAULT now(),
    expiry_date DATE, -- optional, for perishable goods
    UNIQUE (product_id, warehouse_id, supplier_id, received_at)
);


-- Inventory movements / history (for tracking changes over time)
CREATE TABLE inventory_movements (
    id SERIAL PRIMARY KEY,
    batch_id INT NOT NULL REFERENCES inventory_batches(id) ON DELETE CASCADE,
    product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    warehouse_id INT NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
    change_type VARCHAR(50) NOT NULL, -- 'INBOUND', 'SALE', 'TRANSFER',
'ADJUSTMENT'
    quantity_change INT NOT NULL,
    created_at TIMESTAMP DEFAULT now()
);


-- Product Bundles
-- (if product.is_bundle = true, links to its components)
CREATE TABLE product_bundles (
    id SERIAL PRIMARY KEY,
    bundle_product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    component_product_id INT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    quantity INT NOT NULL CHECK (quantity > 0),
    UNIQUE (bundle_product_id, component_product_id)
);
```

Assumptions for Part 2 :

1. Inventory is always tracked as whole units (`INT`).
2. `inventory` is a derived table (must always match sum of `inventory_batches`)
3. Suppliers should be company-specific (each company manages its own supplier list).
4. Every batch is tied to a specific supplier (`supplier_id`) and actual `purchase_price`
5. I assume the threshold applies to the total quantity in a warehouse, not per batch.
6. A bundle is just a special product flagged with is_bundle = TRUE.
7. Deletions cascade for simplicity, but in production we might prefer soft deletes.
8. No multi-currency or tax handling included (out of scope for now).

Question to ask for Part 2 :

1. How to actually treat a bundle (a product in itself or a group of other products).
2. Is supplier and product relationship for a company many-to-many or one-to-many ?

Part 3

My Observations :

- We now have `inventory` (flat table) that stores total quantity of a product per warehouse. This makes generating low-stock alerts much more efficient compared to summing batches.
- Threshold is stored at the product level (e.g., `products.low_stock_threshold`) but for future flexibility, we could be extend it to per-warehouse threshold because that's more practical .
- We need to check `inventory_movements` for a `SALE` movement within a configurable recent period (say last 30 days) otherwise we would be triggering alerts for inactive products .

- A product may have multiple suppliers so for simplicity, we'll return the most recently used supplier for that product in that warehouse .

Implementation through django REST

```
from .models import Product, Warehouse, Inventory, InventoryBatch, InventoryMovement, Supplier

@api_view(['GET'])
def low_stock_alerts(request, company_id):
    """
    GET /api/companies/{company_id}/alerts/low-stock
    Returns low stock alerts for products in a company's warehouses.
    """
    alerts = []
    today = now()
    sales_window_start = today - timedelta(days=30)

     warehouses = Warehouse.objects.filter(company_id=company_id)

    for warehouse in warehouses:

        low_stock_inventories = (
            Inventory.objects
            .filter(warehouse=warehouse, product__low_stock_threshold__isnull=False)
            .select_related("product")
        )

        for inv in low_stock_inventories:
            product = inv.product
            current_stock = inv.quantity
            threshold = product.low_stock_threshold

            if current_stock >= threshold:
                continue  # Stock is healthy, skip

                 recent_sales = (
                InventoryMovement.objects
                .filter(
                    product=product,
                    warehouse=warehouse,
                    movement_type="SALE",
                    created_at__gte=sales_window_start
```

```
    )
    .aggregate(total_sold=Sum("quantity"))
)

if not recent_sales["total_sold"]:
    continue  # No recent sales → skip alert


avg_daily_sales = recent_sales["total_sold"] / 30.0
if avg_daily_sales > 0:
    days_until_stockout = int(current_stock / avg_daily_sales)
else:
    days_until_stockout = None


latest_batch = (
    InventoryBatch.objects
    .filter(product=product, warehouse=warehouse)
    .select_related("supplier")
    .order_by("-received_date")
    .first()
)

supplier_info = None
if latest_batch and latest_batch.supplier:
    supplier_info = {
        "id": latest_batch.supplier.id,
        "name": latest_batch.supplier.name,
        "contact_email": latest_batch.supplier.contact_email,
    }


alerts.append({
    "product_id": product.id,
    "product_name": product.name,
    "sku": product.sku,
    "warehouse_id": warehouse.id,
    "warehouse_name": warehouse.name,
    "current_stock": current_stock,
    "threshold": threshold,
    "days_until_stockout": days_until_stockout,
    "supplier": supplier_info,
})
```

return Response({"alerts": alerts, "total_alerts": len(alerts)}, status=status.HTTP_200_OK)

Edge Cases

- If Product has stock but no supplier yet, Supplier field returned as `null`.
- If there is no recent sales then no alert, even if low stock.
- If threshold is missing for product then I would be skipping alert (unless business says default == 0).
- If we get negative stock (due to adjustment errors) we would still trigger an alert with stockout == 0.
- If multiple warehouses have low stock per product then we would alert each warehouse separately.

Assumptions for Part 3 :

- `inventory` is always consistent with sum of `inventory_batches`.
- Sales velocity is calculated based on `inventory_movements` of type `SALE` within last 30 days.
- Supplier per alert = most recent supplier that delivered stock (`inventory_batches`).
- Company filtering: We only consider warehouses belonging to the `company_id` in request.
- Performance consideration: Use pre-aggregated fields (`inventory`) for stock checks, not batch sums.