Bynry : Backend Eng Intern Submission - Kshitij Paliya

Part 1 :
Issues in the code -

1. Warehouse_id wrongly included in the product model.
   Impact: Products should be global including them to one warehouse is not ideal.

2. No input validation is performed.
   The request.json is directly being used without performing any validation.
   Impact: Crashes if JSON is missing or empty, etc.

3. SKU uniqueness is not ensured.
   Impact: Duplicate SKUs can corrupt the product catalog.

4. The transaction is not atomic.
   The product is committed first and then the inventory.
   Impact: If inventory insert fails, product exists without stock record (data inconsistency).

5. Price stored as float
   data['price'] likely becomes Python float.
   Impact: Precision errors in financial data.

6. No error handling
   No try/except for DB errors.
   Impact: Integrity errors (e.g., duplicate SKU) will crash the server with 500.

7. No ownership/authorization checks
   Any warehouse ID can be used.
   Impact: User could add inventory to another company's warehouse.

8. Inventory always inserted
   No check if (product_id, warehouse_id) already exists.
   Impact: Duplicate inventory rows for same product + warehouse.

9. Improper HTTP response
   Always returns { "message": "Product created" }.
   Impact: No error codes, no distinction between success and failure.

10. Quantity validation missing
    Allows negative or non-integer quantities.
    Impact: Invalid stock values in production.

Corrected Fixes -

```
from decimal import Decimal, InvalidOperation
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError

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

    # --- Input Validation ---
    name = data.get("name")
    sku = data.get("sku")
    price_raw = data.get("price")
    warehouse_id = data.get("warehouse_id")
    initial_quantity = data.get("initial_quantity", 0)

    if not name or not sku or price_raw is None or warehouse_id is None:
        return jsonify({"error": "Missing required fields"}), 400

    # Price validation (Decimal for money-safe storage)
    try:
        price = Decimal(str(price_raw))
    except (InvalidOperation, TypeError):
        return jsonify({"error": "Invalid price format"}), 400

    if int(initial_quantity) < 0:
        return jsonify({"error": "Quantity cannot be negative"}), 400

    # Normalize SKU (case-insensitive uniqueness)
    sku = sku.strip().upper()

    try:
        # --- Transaction Block ---
```

```
        with db.session.begin_nested():
            # Check for existing product with SKU
            product = Product.query.filter_by(sku=sku).first()
            if not product:
                product = Product(name=name, sku=sku, price=price)
                db.session.add(product)
                db.session.flush()  # get product.id

            # Upsert inventory for warehouse
            inventory = Inventory.query.filter_by(
                product_id=product.id,
                warehouse_id=warehouse_id
            ).first()

            if inventory:
                # Business rule: increment existing stock
                inventory.quantity += int(initial_quantity)
            else:
                inventory = Inventory(
                    product_id=product.id,
                    warehouse_id=warehouse_id,
                    quantity=int(initial_quantity)
                )
                db.session.add(inventory)

        db.session.commit()

    except IntegrityError:
        db.session.rollback()
        return jsonify({"error": "SKU already exists"}), 409
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": str(e)}), 500

    return jsonify({
        "message": "Product created/updated successfully",
        "product_id": product.id,
        "warehouse_id": warehouse_id,
        "current_quantity": inventory.quantity
    }), 201
```

Explanation -

- I removed warehouse_id from the Product model because products are global, and warehouses are handled separately in the Inventory table.
- I added validation so that required fields are checked, the price is stored as a decimal, and quantity cannot be negative.
- I made sure SKUs are stored in a consistent format and are unique across the system, both in the database and in code.
- I wrapped the product and inventory creation inside a single transaction so we don’t end up with partial commits.
- Instead of always inserting inventory, I updated it if the product already exists in that warehouse. This prevents duplicates.
- I added proper error handling and return codes like 400, 409, and 500.
- I added a placeholder for authentication to ensure that a user can only manage warehouses that belong to their company.
- The API now returns a clear structured response with product_id, warehouse_id, and the updated stock.

Part 2 :
Database Design -

Companies
Stores the details of each company using the system.
- id (primary key)
- name (unique)
- created_at

Warehouses
Each company can have multiple warehouses.
- id (primary key)
- company_id (linked to companies)
- name
- location
- created_at

Products
- Products are global and can belong to multiple warehouses.
- id (primary key)
- name
- sku (unique across the system)
- price (decimal, non-negative)
- is_bundle (boolean, true if product is a bundle)

- created_at

Product Bundles
Handles bundled products (a bundle contains other products).
- bundle_id (references product)
- product_id (references product)
- quantity (how many of that product in the bundle)
- Primary key is (bundle_id, product_id)

Suppliers
Suppliers who provide products.
- id
- name
- contact_info

Supplier-Product Link
Defines which supplier provides which product.
- supplier_id
- product_id
- Primary key is (supplier_id, product_id)

Inventory
Tracks product quantities in each warehouse.
- id
- product_id
- warehouse_id
- quantity (non-negative)
- updated_at
- Unique constraint on (product_id, warehouse_id)

Inventory Transactions
- Keeps a history of stock changes.
- id
- inventory_id (linked to inventory)
- change_type (add, remove, adjust)
- quantity_change
- created_at

Questions I Would Ask the Product Team

- Do we need to track which user or system made an inventory change?\
- Should bundle prices be calculated automatically from child products, or can bundles have their own price?
- Can a supplier provide products to more than one company?
- Do warehouses need to track capacity limits?
- Do we need to store expiry dates, batch numbers, or lot numbers for products (important in some industries)?
- Should products be categorized (e.g., raw material vs finished goods)?
- Is pricing always in one currency, or should we store currency info as well?
- Should inventory transactions also include a reason (purchase, return, damage, etc.)?

Why I Designed It This Way

- SKUs must be unique, so I put a unique constraint on that field.
- Bundles are handled with a separate join table since bundles can contain multiple products.
- The inventory table uses a combination of product and warehouse to make sure we don’t accidentally store duplicates.
- The inventory_transactions table keeps a full audit trail of every stock change.
- I used constraints to ensure no negative prices or quantities.
- Indexes will be on sku for fast lookups and on (product_id, warehouse_id) for inventory queries.

Part 2 :
API Implementation-

Assumptions-

- Products belong to companies indirectly via warehouses → inventory holds (product_id, warehouse_id, quantity).
- Thresholds are in a ProductThreshold table (company-specific > global > fallback).
- Only include products with sales in the last 30 days (configurable).
- Days until stockout = current_stock / avg_daily_sales.
- Supplier info is joined from a SupplierProducts table.

- Using Sequelize models (Product, Warehouse, Inventory, Sale, SaleItem, Supplier, SupplierProduct, ProductThreshold).

Code -

```
// routes/alerts.js
const express = require("express");
const { Op, fn, col, literal } = require("sequelize");
const {
  Product,
  Warehouse,
  Inventory,
  Sale,
  SaleItem,
  Supplier,
  SupplierProduct,
  ProductThreshold,
  sequelize
} = require("../models");

const router = express.Router();

// Configurable window for recent sales
const DEFAULT_RECENT_DAYS = 30;

router.get("/api/companies/:companyId/alerts/low-stock", async (req, res) => {
  const { companyId } = req.params;
  const daysWindow = parseInt(req.query.days || DEFAULT_RECENT_DAYS, 10);
  const page = parseInt(req.query.page || 1, 10);
  const perPage = Math.min(parseInt(req.query.per_page || 100, 10), 500);

  try {
    // Step 1: find all warehouses for this company
    const warehouses = await Warehouse.findAll({
      where: { company_id: companyId },
      attributes: ["id", "name"]
    });

    if (warehouses.length === 0) {
      return res.json({ alerts: [], total_alerts: 0 });
    }
```

```
    const warehouseIds = warehouses.map(w => w.id);

    // Step 2: recent sales aggregate (product + warehouse)
    const cutoff = new Date();
    cutoff.setDate(cutoff.getDate() - daysWindow);

    const recentSales = await SaleItem.findAll({
      include: [
        {
          model: Sale,
          attributes: [],
          where: {
            company_id: companyId,
            created_at: { [Op.gte]: cutoff }
          }
        }
      ],
      attributes: [
        "product_id",
        "warehouse_id",
        [fn("SUM", col("qty")), "recent_qty"]
      ],
      where: { warehouse_id: { [Op.in]: warehouseIds } },
      group: ["product_id", "warehouse_id"]
    });

    const salesMap = {};
    recentSales.forEach(s => {
      salesMap[`${s.product_id}-${s.warehouse_id}`] = parseInt(s.get("recent_qty"),
10);
    });

    // Step 3: query inventory + product + thresholds + supplier
    const inventoryRows = await Inventory.findAll({
      include: [
        { model: Product, attributes: ["id", "name", "sku"] },
        { model: Warehouse, attributes: ["id", "name"], where: { company_id: companyId
} }
      ],
      where: { warehouse_id: { [Op.in]: warehouseIds } }
    });

    const alerts = [];
```

```
for (const row of inventoryRows) {
  const product = row.Product;
  const warehouse = row.Warehouse;
  const key = `${product.id}-${warehouse.id}`;

  // skip products with no recent sales
  if (!salesMap[key]) continue;

  // get thresholds (company-specific > global > default 0)
  let threshold = 0;
  const compThresh = await ProductThreshold.findOne({
    where: { product_id: product.id, company_id: companyId }
  });
  if (compThresh) {
    threshold = compThresh.threshold;
  } else {
    const globalThresh = await ProductThreshold.findOne({
      where: { product_id: product.id, company_id: null }
    });
    if (globalThresh) threshold = globalThresh.threshold;
  }

  if (row.quantity > threshold) continue; // stock is fine

  // calculate days until stockout
  const recentQty = salesMap[key];
  const avgDailySales = recentQty / daysWindow;
  let daysUntilStockout = null;
  if (avgDailySales > 0) {
    daysUntilStockout = Math.ceil(row.quantity / avgDailySales);
  }

  // supplier info (pick first primary or any)
  const supplierProduct = await SupplierProduct.findOne({
    where: { product_id: product.id },
    include: [{ model: Supplier, attributes: ["id", "name", "contact_email"] }],
    order: [["is_primary", "DESC"], ["lead_time_days", "ASC"]]
  });

  let supplier = null;
  if (supplierProduct && supplierProduct.Supplier) {
    supplier = {
      id: supplierProduct.Supplier.id,
      name: supplierProduct.Supplier.name,
```

```
          contact_email: supplierProduct.Supplier.contact_email
        };
      }

      alerts.push({
        product_id: product.id,
        product_name: product.name,
        sku: product.sku,
        warehouse_id: warehouse.id,
        warehouse_name: warehouse.name,
        current_stock: row.quantity,
        threshold,
        days_until_stockout: daysUntilStockout,
        supplier
      });
    }

    // pagination
    const totalAlerts = alerts.length;
    const paginated = alerts.slice((page - 1) * perPage, page * perPage);

    return res.json({ alerts: paginated, total_alerts: totalAlerts });
  } catch (err) {
    console.error(err);
    return res.status(500).json({ error: "Internal server error" });
  }
});
module.exports = router;
```

Explanation-

- Get all warehouses that belong to the company.
- Look at sales in the last X days (default 30) and total how many units of each product were sold per warehouse.
- Fetch inventory for those warehouses and join with product + warehouse details.
- Skip products with no recent sales.
- Check threshold → first company-specific, else global, else 0.
- If stock is below or equal to threshold, flag it.
- Calculate days_until_stockout using sales average.
- Attach supplier info (first primary, else lowest lead time).
- Return JSON with alerts + total count.

Edge Cases Handled-

- No warehouses for company → returns empty list.
- No sales for product → product skipped.
- No supplier → supplier = null.
- Threshold not set → defaults to 0 (so only truly empty stock triggers alert).
- Division by zero → days_until_stockout = null.
- Large dataset → added pagination with page and per_page.