Name: Omkar Deshpande
Github Repo Link: Backend_Engineering_Intern

Part 1: Code Review & Debugging

Answer:

Code Issues:

1. SKU uniqueness not enforced: The code does not check if the SKU already exists, violating the business rule that SKUs must be unique platform-wide.
2. Potential race condition on SKU: With concurrent requests, two products with the same SKU may be created due to lack of uniqueness constraint and checking.
3. No error handling: If any required field is missing or invalid, or if DB commit fails, the code doesn't handle these cases gracefully.
4. Duplicate commit calls: **db.session.commit()** is called twice unnecessarily within one request; this could be optimized.
5. Missing default/optional field handling: Some fields are optional but no defaults or validations are shown.
6. Price decimal handling: Price should be stored and handled as a decimal, not float, for accuracy.
7. Inventory creation tied immediately to product creation: If inventory creation fails, the product is already committed leading to inconsistent state.
8. No transaction management: Creating product and inventory should be atomic.

Impact:

- Duplicate SKUs causing inconsistent inventory/product data.
- Partial product creation without inventory leading to out-of-sync records.
- Crashes caused by missing required fields or bad data.
- Unexpected financial inconsistency due to price float handling.

Corrected Version:

```
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError
from decimal import Decimal
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.json or {}
    # Validate required fields
    required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
    missing_fields = [f for f in required_fields if f not in data]
    if missing_fields:
```

```
        return jsonify({"error": f"Missing fields: {', '.join(missing_fields)}"}),
400

    # Check SKU uniqueness (assumes unique constraint on sku column)
    existing = Product.query.filter_by(sku=data['sku']).first()
    if existing:
        return jsonify({"error": "SKU already exists"}), 409
    try:
        # Convert price to Decimal
        price = Decimal(str(data['price']))
    except:
        return jsonify({"error": "Invalid price format"}), 400
    try:
        # Atomic transaction
        with db.session.begin_nested():
            product = Product(
                name=data['name'],
                sku=data['sku'],
                price=price,
                warehouse_id=data['warehouse_id']  # Consider if warehouse_id
should be in product or only inventory
            )
            db.session.add(product)
            db.session.flush()  # Flush to get product.id for inventory FK

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

        db.session.commit()
      return jsonify({"message": "Product created", "product_id": product.id}), 201
    except IntegrityError:
        db.session.rollback()
```

```
        return jsonify({"error": "Database integrity error"}), 500
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": str(e)}), 500
```

Explanation:

- Added validation for required fields and proper error responses.
- Enforced SKU uniqueness with a pre-check and rely on DB unique constraint.
- Used Decimal for price.
- Wrapped product and inventory additions in a transaction (begin_nested) to keep atomicity.
- Single commit at end.
- Proper error handling with rollback.
- Converted quantity to int for safety.
- Returned appropriate HTTP status codes (400 for bad request, 409 for conflict, 201 for created).

Part 2: Database Design

Answer:

Design Schema:

```
-- Company owns warehouses and suppliers
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE
);

CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    company_id INT NOT NULL REFERENCES companies(id),
    name VARCHAR(255) NOT NULL,
    UNIQUE (company_id, name)
);

CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    company_id INT NOT NULL REFERENCES companies(id),
    name VARCHAR(255) NOT NULL,
```

```
    contact_email VARCHAR(255)
);

-- Products table
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    sku VARCHAR(100) NOT NULL UNIQUE,
    name VARCHAR(255) NOT NULL,
    price NUMERIC(12, 2) NOT NULL,
    type VARCHAR(50) NOT NULL DEFAULT 'standard'  -- e.g. 'standard' or 'bundle'
);

-- To represent bundles containing other products
CREATE TABLE product_bundles (
    bundle_product_id INT NOT NULL REFERENCES products(id),
    component_product_id INT NOT NULL REFERENCES products(id),
    quantity INT NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (bundle_product_id, component_product_id)
);

-- Inventory per product per warehouse, different quantities
CREATE TABLE inventory (
    id SERIAL PRIMARY KEY,
    product_id INT NOT NULL REFERENCES products(id),
    warehouse_id INT NOT NULL REFERENCES warehouses(id),
    quantity INT NOT NULL CHECK (quantity >= 0),
    UNIQUE (product_id, warehouse_id)
);

-- Track all inventory changes (additions, removals)
CREATE TABLE inventory_changes (
    id SERIAL PRIMARY KEY,
    inventory_id INT NOT NULL REFERENCES inventory(id),
    change INT NOT NULL,           -- positive or negative quantity change
    change_date TIMESTAMPTZ NOT NULL DEFAULT NOW(),
```

```
    reason VARCHAR(255)            -- optional description, e.g. sales, restock
);


-- Products supplied by suppliers
CREATE TABLE product_suppliers (
    product_id INT NOT NULL REFERENCES products(id),
    supplier_id INT NOT NULL REFERENCES suppliers(id),
    PRIMARY KEY (product_id, supplier_id)
);


-- Sales activity could be tracked separately (not specified)
CREATE TABLE sales (
    id SERIAL PRIMARY KEY,
    product_id INT NOT NULL REFERENCES products(id),
    warehouse_id INT NOT NULL REFERENCES warehouses(id),
    quantity INT NOT NULL,
    sale_date TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Gaps/Questions for Product Team:

- How to identify product types and define bundle logic? What properties does a bundle have?
- How should we handle supplier-product relationships? Single supplier per product or multiple?
- What attributes define low-stock thresholds per product type?
- Should warehouses have locations or other metadata?
- How detailed should sales activity tracking be? What defines “recent” sales?
- Permissions/security: who can access/modify?
- What optional product fields are required?

Design Explanation:

1. Unique constraints on SKU, company warehouse names to avoid duplicates.
2. Separate product_bundles table to model many-to-many bundle components.
3. Inventory table with combination of product and warehouse unique for accurate stock tracking.
4. Inventory_changes table for audit trail of stock changes aligned with tracking requirement.
5. Numeric price with decimals for monetary values.
6. Sales table to help determine recent sales activity.
7. Product-supplier link allows multiple suppliers per product.
8. Timestamps and constraints to enforce data integrity and enable queries.

Part 3: API Implementation

Answer:

Flask Implementation:

```
from flask import jsonify
from datetime import datetime, timedelta
from sqlalchemy import func

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
    # Assumptions:
    # - Low stock threshold stored in Product.low_stock_threshold (numeric)
    # - Recent sales within last 30 days
    # - Days until stockout = current_stock / avg_daily_sales (if avg_daily_sales >
0 else None)

    try:
        today = datetime.utcnow()
        recent_period = today - timedelta(days=30)

        # Subquery: recent sales aggregated per product per warehouse
        recent_sales_subq = (
            db.session.query(
                Sales.product_id,
                Sales.warehouse_id,
                func.sum(Sales.quantity).label('total_sold')
            )
            .join(Warehouse, Warehouse.id == Sales.warehouse_id)
            .filter(Sales.sale_date >= recent_period, Warehouse.company_id ==
company_id)
            .group_by(Sales.product_id, Sales.warehouse_id)
            .subquery()
        )

        # Query inventory joined with recent sales and supplier info
        alerts_query = (
            db.session.query(
                Inventory.product_id,
                Product.name.label('product_name'),
                Product.sku,
                Inventory.warehouse_id,
                Warehouse.name.label('warehouse_name'),
                Inventory.quantity.label('current_stock'),
                Product.low_stock_threshold.label('threshold'),
                Supplier.id.label('supplier_id'),
                Supplier.name.label('supplier_name'),
                Supplier.contact_email,
                recent_sales_subq.c.total_sold
            )
            .join(Product, Product.id == Inventory.product_id)
```

```
            .join(Warehouse, Warehouse.id == Inventory.warehouse_id)
            .join(product_suppliers, product_suppliers.c.product_id == Product.id)
            .join(Supplier, Supplier.id == product_suppliers.c.supplier_id)
            .join(recent_sales_subq,
                  (recent_sales_subq.c.product_id == Inventory.product_id) &
                  (recent_sales_subq.c.warehouse_id == Inventory.warehouse_id))
            .filter(Warehouse.company_id == company_id)
            .filter(Inventory.quantity <= Product.low_stock_threshold)
        ).all()

        alerts = []
        for row in alerts_query:
            avg_daily_sales = row.total_sold / 30 if row.total_sold else None
            days_until_stockout = (row.current_stock / avg_daily_sales) if
avg_daily_sales and avg_daily_sales > 0 else None

            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": round(days_until_stockout) if
days_until_stockout else None,
                "supplier": {
                    "id": row.supplier_id,
                    "name": row.supplier_name,
                    "contact_email": row.contact_email
                }
            })

        response = {
            "alerts": alerts,
            "total_alerts": len(alerts)
        }
        return jsonify(response), 200

    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Edge Cases:

- No recent sales = no alert generated.
- Division by zero avoided by checking avg daily sales.
- Products with multiple suppliers: currently picks the first due to join, this can be enhanced.
- Company with no warehouses or no matching inventory handled gracefully (empty alerts).
- DB errors are caught and return 500 error.

Approach:

1. Use aggregation to compute recent sales per product/warehouse.
2. Join inventory with recent sales to filter low stock only on products with recent sales.
3. Calculate days until stockout dynamically.
4. Return supplier contact info to aid reorder.
5. Filter on company_id to scope alerts.
6. Modular queries for clarity.