Inventory Management System for B2B SaaS

Created By: Siddhi Chhapre

Part 1: Code Review & Debugging

Review:

Identified Issues:

1. **request.json can be None (Technical)**
   Description : No check for malformed/empty JSON body.
   Impact : Results in TypeError: 'NoneType' object is not subscriptable, returns 500 error.

2. **Insufficient Input Validation** (Technical)
   Description : The code directly accesses data from the request with data['key'] without checking if the data exists or is valid.
   Impact : If the request body isn't JSON, or if required fields like 'name' or 'sku' are missing, the app will crash and return a generic 500 Internal Server Error. Might result in poor user experience and fails to handle cases where some fields might be optional.

3. **No Atomic Transactions** (Technical)
   Description : The code commits the Product and Inventory in two separate transactions.
   Impact : If the system fails after creating the product but before creating the inventory, you'll have an orphaned product in the database with no inventory record. This leaves your data in an inconsistent state.

4. No SKU uniqueness check (business and tech)
   Description : Duplicate SKUs may be created, violating uniqueness across platform.
   Impact : Duplicates confuse inventory tracking and reporting

5. Poor HTTP Response Semantics (Tech)
   Description : A successful creation returns a default 200 OK status code.
   Impact : This isn't a bug, but it's not a best practice. The standard for successfully creating a resource is a **201 Created** status code, which gives clients more precise feedback.

Fixed code:

```
from flask import jsonify, request
from decimal import Decimal, InvalidOperation

@app.route('/api/products', methods=['POST'])
def create_product():
    # 1. Validate request body and required fields
    data = request.json
    if not data:
        return jsonify({"error": "Invalid or missing JSON body"}), 400

    required = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
    if any(field not in data for field in required):
        return jsonify({"error": "Missing one or more required fields"}), 400

    # 2. Check for SKU uniqueness
    if Product.query.filter_by(sku=data['sku']).first():
        return jsonify({"error": f"Product with SKU '{data['sku']}' already exists"}), 409

    try:
        # 3. Validate data types and create objects in a single transaction
        price = Decimal(data['price'])
        quantity = int(data['initial_quantity'])

        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=price,
            # Use .get() for any optional fields
            description=data.get('description')
        )
        db.session.add(product)
        db.session.flush() # Flush to get the product ID for the inventory record

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

        # 4. Commit the single, atomic transaction
        db.session.commit()

        # 5. Return a proper success response
        return jsonify({
            "message": "Product created successfully",
            "product_id": product.id
        }), 201 # 201 Created

    except (ValueError, InvalidOperation):
        return jsonify({"error": "Invalid data type for price or quantity"}), 400
    except Exception as e:
        # If anything goes wrong, roll back all changes
        db.session.rollback()
        # In production, you should log the error `e`
        return jsonify({"error": "An internal error occurred"}), 500
```

Explanation:

- **Atomic Transactions**: Operations are now in a **try...except** block with a single commit and rollback. This guarantees that the product and its inventory are created together or not at all, preventing data corruption.
- **Robust Validation**: **Input is validated** for required fields and correct data types (e.g., decimal for price). This prevents server crashes from bad data and returns helpful 400 Bad Request errors instead.
- **SKU Uniqueness**: A pre-check for **SKU uniqueness** prevents duplicates by returning a 409 Conflict error, enforcing a key business rule.
- **Proper Semantics**: The API now returns the standard **201 Created** status code on success and handles **optional fields** gracefully to prevent errors.

Part 2: Database Design

Review:

Table Schema:

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

-- Warehouses
CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    company_id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
    location TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (company_id) REFERENCES companies(id) ON DELETE CASCADE
);

-- Products
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) NOT NULL UNIQUE,
    price DECIMAL(10, 2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Bundled Products (Many-to-Many: bundle -> products)
CREATE TABLE product_bundles (
    bundle_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (bundle_id, product_id),
    FOREIGN KEY (bundle_id) REFERENCES products(id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE
);

-- Suppliers
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    contact_info TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Supplier-Product Mapping (Many-to-Many)
CREATE TABLE supplier_products (
    supplier_id INT NOT NULL,
    product_id INT NOT NULL,
    PRIMARY KEY (supplier_id, product_id),
    FOREIGN KEY (supplier_id) REFERENCES suppliers(id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE
);
```

```
-- Inventory (per product per warehouse)
CREATE TABLE inventory (
   id SERIAL PRIMARY KEY,
   product_id INT NOT NULL,
   warehouse_id INT NOT NULL,
   quantity INT NOT NULL DEFAULT 0,
   last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   UNIQUE (product_id, warehouse_id),
   FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE,
   FOREIGN KEY (warehouse_id) REFERENCES warehouses(id) ON DELETE CASCADE
);

-- Inventory History (auditing inventory level changes)
CREATE TABLE inventory_logs (
   id SERIAL PRIMARY KEY,
   inventory_id INT NOT NULL,
   change_type VARCHAR(50) NOT NULL CHECK (change_type IN ('INCREASE', 'DECREASE',
'TRANSFER')),
   quantity_change INT NOT NULL,
   updated_by VARCHAR(100), -- could be user_id if auth system exists
   reason TEXT,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (inventory_id) REFERENCES inventory(id) ON DELETE CASCADE
);
```

Identify Gaps in Requirements – Questions

About Inventory:

- **Do we need to track expiry dates, batch numbers, or serial numbers for products?**
- **Should we prevent stock duplication (same SKU added twice to same warehouse)?**
- **Should transfers between warehouses be logged as a specific event?**

On Suppliers:

- **Do we need to track the cost of a product from different suppliers? For example, does "Supplier A" sell "Widget A" for a different price than "Supplier B"?**
- **What is the business logic for choosing a "primary" supplier if a product has multiple suppliers? Is it the cheapest, fastest, or manually assigned?**

About Products and Bundles:

- **Should bundles inherit pricing from individual products or have their own price?**
- **Can bundles contain other bundles (nested)?**
- **Can the same product exist in multiple bundles?**

Design Decisions

Decision	Justification
sku UNIQUE	Ensures global SKU uniqueness (as per your business rule)
Decimal(10,2) for price	Prevents floating-point precision issues in currency
product_bundles table	Enables flexible many-to-many mapping of bundled items
supplier_products table	Supports multiple suppliers per product
inventory table has (product_id, warehouse_id) UNIQUE	Ensures one record per product per warehouse
inventory_logs table	Supports full audit history of stock changes
Foreign keys with ON DELETE CASCADE	Keeps data integrity and avoids orphaned rows

Decision	Justification
created_at, last_updated	Allows for sorting, filtering, and tracking updates
Indexes on foreign keys	Improves JOIN performance

Part 3: API Implementation

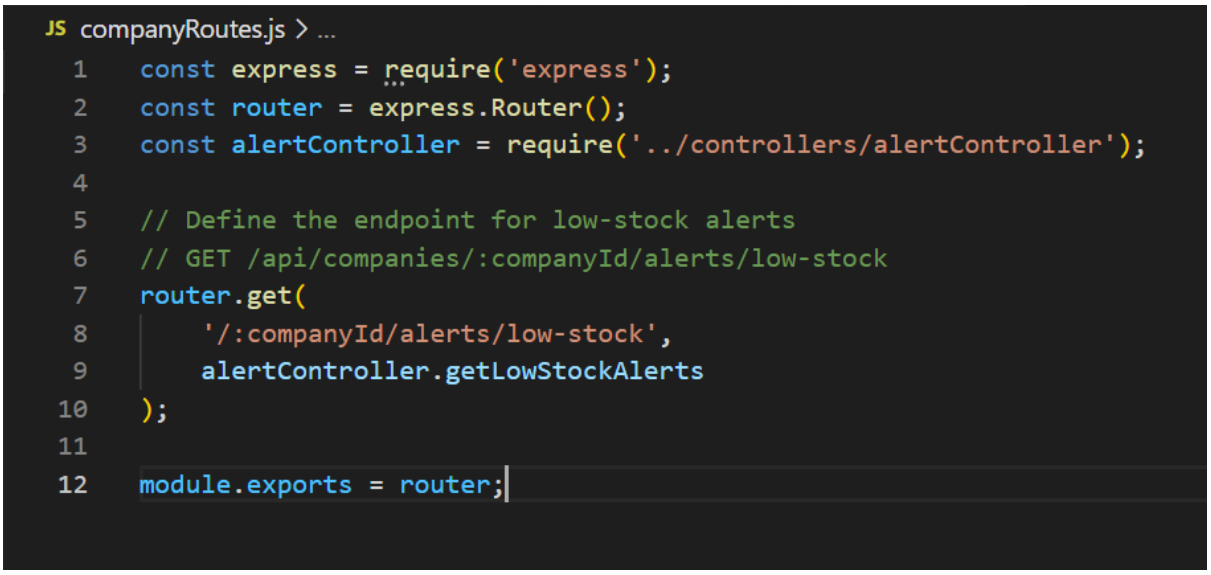
Assumptions :

- **Database Setup: I'm assuming all the necessary database models (Product, Warehouse, etc.) and their relationships are already configured.**
- **Recent Sales: "Recent sales activity" means any sales within the last 30 days.**
- **Stockout Calculation: The "days until stockout" is based on average daily sales; it'll be**
- **null if an item hasn't sold recently.**
- **Primary Supplier: The code looks for a "primary" supplier to include for reordering. If a product doesn't have one, that field will be**
- **null.**
- **Security: I've skipped user authentication for this exercise, but a real app would definitely need it.**

Implementation:

1. Route Definition (routes/companyRoutes.js)

This file defines the API endpoint and links it to the controller logic.



companyRoutes.js > ...

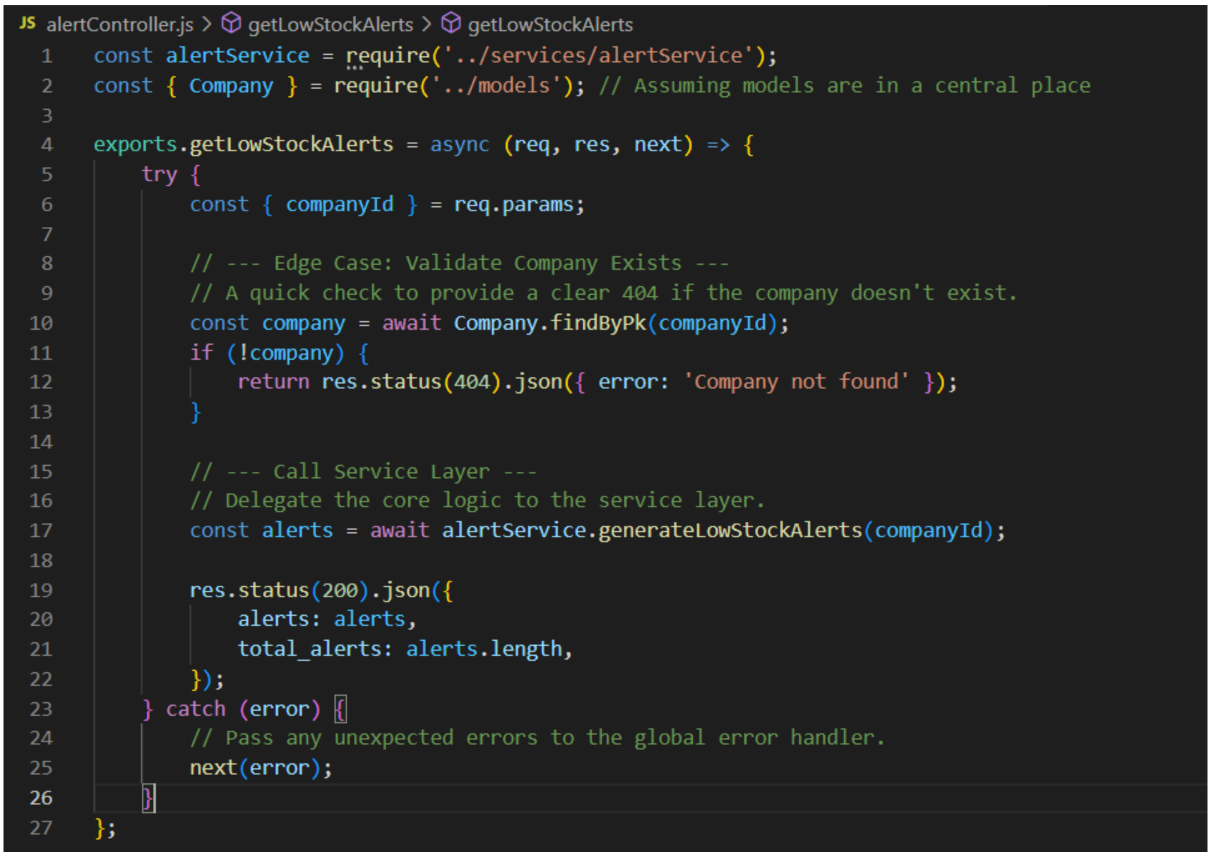
```javascript
const express = require('express');
const router = express.Router();
const alertController = require('../controllers/alertController');

// Define the endpoint for low-stock alerts
// GET /api/companies/:companyId/alerts/low-stock
router.get(
    '/:companyId/alerts/low-stock',
    alertController.getLowStockAlerts
);

module.exports = router;
```

2. Controller (controllers/alertController.js)

The controller handles the HTTP request and response. It validates input and calls the service layer to execute the business logic.



JS alertController.js > getLowStockAlerts > getLowStockAlerts

```javascript
const alertService = require('../services/alertService');
const { Company } = require('../models'); // Assuming models are in a central place

exports.getLowStockAlerts = async (req, res, next) => {
    try {
        const { companyId } = req.params;

        // --- Edge Case: Validate Company Exists ---
        // A quick check to provide a clear 404 if the company doesn't exist.
        const company = await Company.findByPk(companyId);
        if (!company) {
            return res.status(404).json({ error: 'Company not found' });
        }

        // --- Call Service Layer ---
        // Delegate the core logic to the service layer.
        const alerts = await alertService.generateLowStockAlerts(companyId);

        res.status(200).json({
            alerts: alerts,
            total_alerts: alerts.length,
        });
    } catch (error) {
        // Pass any unexpected errors to the global error handler.
        next(error);
    }
};
```

3. Service Layer (services/alertService.js)

This is where the core business logic resides. It queries the database and transforms the data into the required response format.

alertService.js