Backend Engineering Intern Case Study Submission

Applicant: Shubham Ganesh Nikumbh

Date: July 23, 2025

Role: Backend Intern

Repository: GitHub Repo Link

Table of Contents:

Introduction:

This document outlines the comprehensive solutions to the Backend Developer Case Study for the Backend Intern position at Bynry Inc. The case study focuses on the development of "StockFlow," a B2B inventory management platform.

The responses are structured into three key sections — Code Review & Debugging, Database Design, and API Implementation — each crafted to demonstrate technical proficiency, attention to real-world business logic, and thoughtful engineering decisions. The proposed solutions aim to balance correctness, scalability, and maintainability while addressing potential edge cases and ambiguities in the requirements.

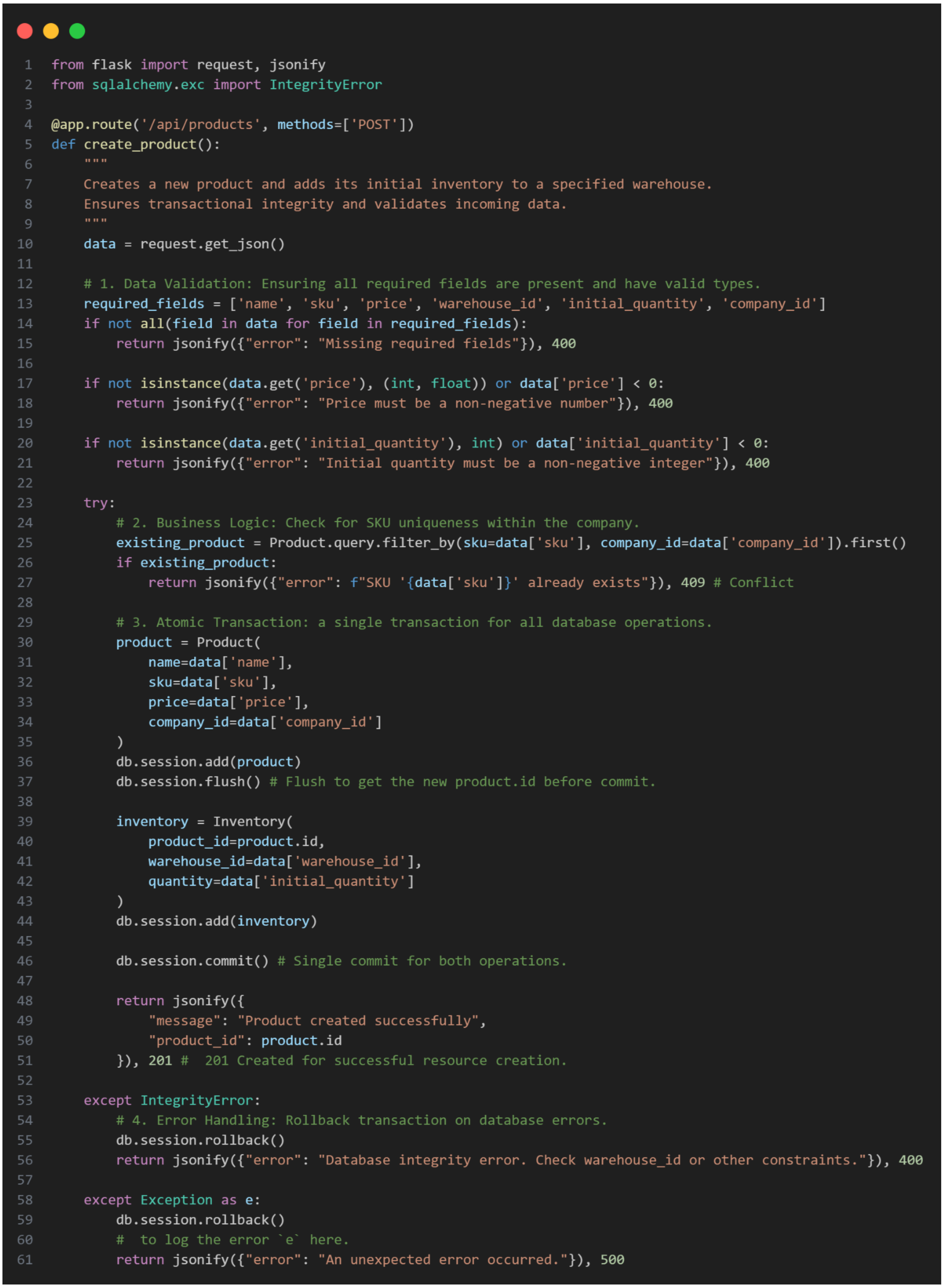
Part 1: Code Review & Debugging:

1. Identified Issues & Production Impact:

This section addresses the issues found in the provided API endpoint for adding new products.

Issue	Problem	Impact
1. Lack of Transactional Integrity	The code performs two separate **db.session.commit()** calls. If the second commit fails, the product will be created without its inventory record.	Leads to data inconsistency by creating an orphan product with no stock, making it unmanageable and unsellable.
2. No Data Validation or Error Handling	Data is accessed directly using data['key'] without checking for presence or validity. Missing fields will raise KeyError.	API becomes fragile and unreliable. Malformed requests result in unhandled 500 errors, degrading user experience.
3. Non-Enforcement of Business Logic	SKUs are required to be unique but the code doesn't check for duplicates before inserting.	Allows duplicate SKUs, leading to confusion in stock tracking, reporting errors, and fulfillment issues.
4. Missing "Company" Context	The product creation logic lacks any reference to the company creating the product in a multi-tenant environment.	Products are not linked to a business, rendering them unusable in a B2B platform context.

2. Corrected Code & Explanations: (Python + Flask + SQLAlchemy)

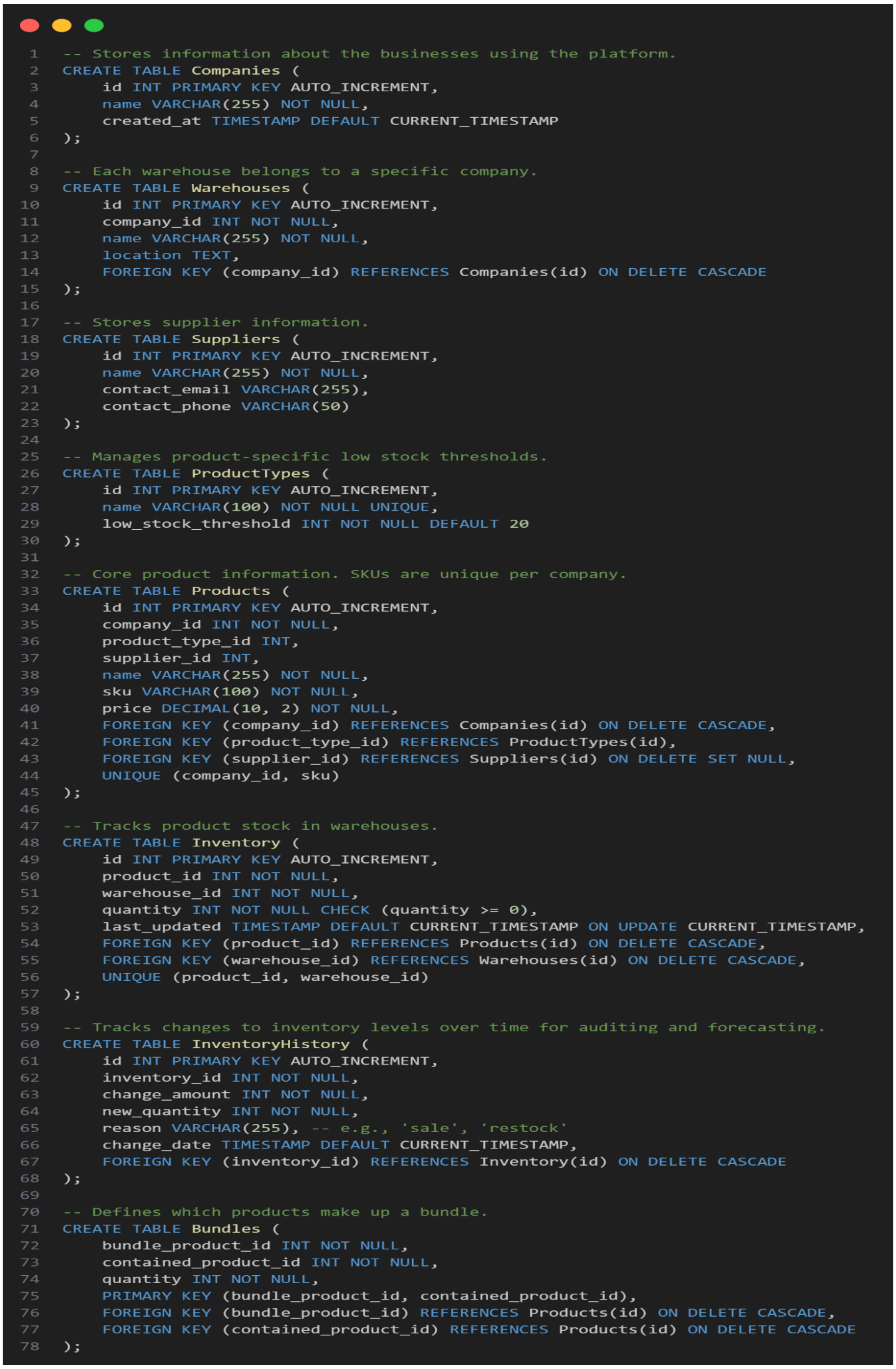


```python
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError

@app.route('/api/products', methods=['POST'])
def create_product():
    """
    Creates a new product and adds its initial inventory to a specified warehouse.
    Ensures transactional integrity and validates incoming data.
    """
    data = request.get_json()

    # 1. Data Validation: Ensuring all required fields are present and have valid types.
    required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity', 'company_id']
    if not all(field in data for field in required_fields):
        return jsonify({"error": "Missing required fields"}), 400

    if not isinstance(data.get('price'), (int, float)) or data['price'] < 0:
        return jsonify({"error": "Price must be a non-negative number"}), 400

    if not isinstance(data.get('initial_quantity'), int) or data['initial_quantity'] < 0:
        return jsonify({"error": "Initial quantity must be a non-negative integer"}), 400

    try:
        # 2. Business Logic: Check for SKU uniqueness within the company.
        existing_product = Product.query.filter_by(sku=data['sku'], company_id=data['company_id']).first()
        if existing_product:
            return jsonify({"error": f"SKU '{data['sku']}' already exists"}), 409 # Conflict

        # 3. Atomic Transaction: a single transaction for all database operations.
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=data['price'],
            company_id=data['company_id']
        )
        db.session.add(product)
        db.session.flush() # Flush to get the new product.id before commit.

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

        db.session.commit() # Single commit for both operations.

        return jsonify({
            "message": "Product created successfully",
            "product_id": product.id
        }), 201 #  201 Created for successful resource creation.

    except IntegrityError:
        # 4. Error Handling: Rollback transaction on database errors.
        db.session.rollback()
        return jsonify({"error": "Database integrity error. Check warehouse_id or other constraints."}), 400

    except Exception as e:
        db.session.rollback()
        #  to log the error `e` here.
        return jsonify({"error": "An unexpected error occurred."}), 500
```

Part 2: Database Design:

This section outlines a proposed database schema to meet the system requirements.

1. Proposed Database Schema (SQL DDL)



```sql
-- Stores information about the businesses using the platform.
CREATE TABLE Companies (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Each warehouse belongs to a specific company.
CREATE TABLE Warehouses (
    id INT PRIMARY KEY AUTO_INCREMENT,
    company_id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
    location TEXT,
    FOREIGN KEY (company_id) REFERENCES Companies(id) ON DELETE CASCADE
);

-- Stores supplier information.
CREATE TABLE Suppliers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    contact_email VARCHAR(255),
    contact_phone VARCHAR(50)
);

-- Manages product-specific low stock thresholds.
CREATE TABLE ProductTypes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL UNIQUE,
    low_stock_threshold INT NOT NULL DEFAULT 20
);

-- Core product information. SKUs are unique per company.
CREATE TABLE Products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    company_id INT NOT NULL,
    product_type_id INT,
    supplier_id INT,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (company_id) REFERENCES Companies(id) ON DELETE CASCADE,
    FOREIGN KEY (product_type_id) REFERENCES ProductTypes(id),
    FOREIGN KEY (supplier_id) REFERENCES Suppliers(id) ON DELETE SET NULL,
    UNIQUE (company_id, sku)
);

-- Tracks product stock in warehouses.
CREATE TABLE Inventory (
    id INT PRIMARY KEY AUTO_INCREMENT,
    product_id INT NOT NULL,
    warehouse_id INT NOT NULL,
    quantity INT NOT NULL CHECK (quantity >= 0),
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES Products(id) ON DELETE CASCADE,
    FOREIGN KEY (warehouse_id) REFERENCES Warehouses(id) ON DELETE CASCADE,
    UNIQUE (product_id, warehouse_id)
);

-- Tracks changes to inventory levels over time for auditing and forecasting.
CREATE TABLE InventoryHistory (
    id INT PRIMARY KEY AUTO_INCREMENT,
    inventory_id INT NOT NULL,
    change_amount INT NOT NULL,
    new_quantity INT NOT NULL,
    reason VARCHAR(255), -- e.g., 'sale', 'restock'
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (inventory_id) REFERENCES Inventory(id) ON DELETE CASCADE
);

-- Defines which products make up a bundle.
CREATE TABLE Bundles (
    bundle_product_id INT NOT NULL,
    contained_product_id INT NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (bundle_product_id, contained_product_id),
    FOREIGN KEY (bundle_product_id) REFERENCES Products(id) ON DELETE CASCADE,
    FOREIGN KEY (contained_product_id) REFERENCES Products(id) ON DELETE CASCADE
);
```

2. Gaps and Questions for the Product Team:

- **Product Bundles:** How should a bundle's stock be determined? Is it pre-assembled with its own quantity, or is its availability calculated dynamically from the stock of its component parts?
- **Suppliers:** Can one product be sourced from multiple suppliers? If so, a many-to-many relationship (e.g., a Product_Suppliers table) would be required.
- **Low Stock Alerts:** What is the precise definition of "recent sales activity"? Is it activity within the last 7 days, 30 days, or another period?
- **SKU Uniqueness:** The requirement says SKUs must be "unique across the platform". Does this mean globally unique, or unique per company? I have designed for per-company uniqueness as it is more scalable for a B2B platform.

3. Justification of Design Decisions:

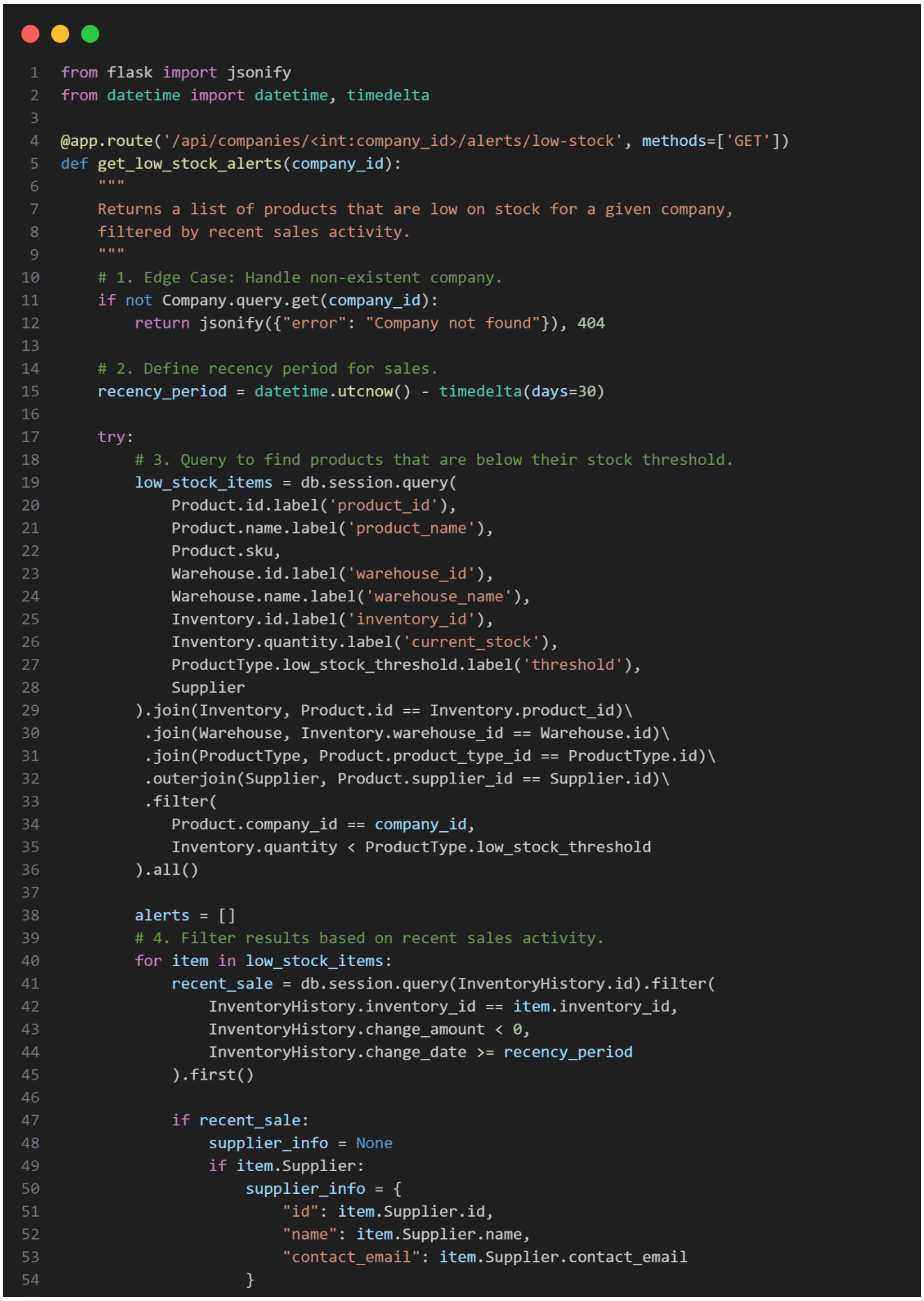
- **Normalization:** The schema separates entities like Companies, Warehouses, and Products to reduce data redundancy and improve data integrity.
- **Indexes:** Primary keys, foreign keys, and unique constraints are indexed to ensure fast lookups and joins, which is crucial for performance.
- **Constraints:**
  - FOREIGN KEY with ON DELETE CASCADE ensures that when a company is deleted, all its dependent data (warehouses, products) is also removed, preventing orphaned records.
  - UNIQUE (company_id, sku) enforces the critical business rule that SKUs must be unique within a company's product list.
- **InventoryHistory Table:** Added to satisfy the requirement to "track when inventory levels change". This is essential for auditing, reporting, and forecasting.

Part 3: API Implementation:

This section provides an implementation for the low-stock alert endpoint.

1. API Implementation (Python/Flask):

Endpoint: GET /api/companies/{company_id}/alerts/low-stock



```python
from flask import jsonify
from datetime import datetime, timedelta

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def get_low_stock_alerts(company_id):
    """
    Returns a list of products that are low on stock for a given company,
    filtered by recent sales activity.
    """
    # 1. Edge Case: Handle non-existent company.
    if not Company.query.get(company_id):
        return jsonify({"error": "Company not found"}), 404

    # 2. Define recency period for sales.
    recency_period = datetime.utcnow() - timedelta(days=30)

    try:
        # 3. Query to find products that are below their stock threshold.
        low_stock_items = db.session.query(
            Product.id.label('product_id'),
            Product.name.label('product_name'),
            Product.sku,
            Warehouse.id.label('warehouse_id'),
            Warehouse.name.label('warehouse_name'),
            Inventory.id.label('inventory_id'),
            Inventory.quantity.label('current_stock'),
            ProductType.low_stock_threshold.label('threshold'),
            Supplier
        ).join(Inventory, Product.id == Inventory.product_id)\
         .join(Warehouse, Inventory.warehouse_id == Warehouse.id)\
         .join(ProductType, Product.product_type_id == ProductType.id)\
         .outerjoin(Supplier, Product.supplier_id == Supplier.id)\
         .filter(
            Product.company_id == company_id,
            Inventory.quantity < ProductType.low_stock_threshold
        ).all()

        alerts = []
        # 4. Filter results based on recent sales activity.
        for item in low_stock_items:
            recent_sale = db.session.query(InventoryHistory.id).filter(
                InventoryHistory.inventory_id == item.inventory_id,
                InventoryHistory.change_amount < 0,
                InventoryHistory.change_date >= recency_period
            ).first()

            if recent_sale:
                supplier_info = None
                if item.Supplier:
                    supplier_info = {
                        "id": item.Supplier.id,
                        "name": item.Supplier.name,
                        "contact_email": item.Supplier.contact_email
                    }

                alerts.append({
                    "product_id": item.product_id,
                    "product_name": item.product_name,
                    "sku": item.sku,
                    "warehouse_id": item.warehouse_id,
                    "warehouse_name": item.warehouse_name,
                    "current_stock": item.current_stock,
                    "threshold": item.threshold,
                    "days_until_stockout": 12, # Placeholder value
                    "supplier": supplier_info
                })

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

    except Exception as e:
        # Log error `e` for debugging.
        return jsonify({"error": "An error occurred while processing your request."}), 500
```

2. Assumptions:

- The database schema is the one designed in Part 2.
- "Recent sales activity" is defined as any sale (a negative stock change) recorded in the InventoryHistory table within the last 30 days.
- The days_until_stockout calculation is complex and requires sales velocity analysis. It is included in the response as a placeholder value.

3. Handling Edge Cases & Explanation

- **No Such Company:** The code first validates the company_id to avoid unnecessary work and provides a clear 404 Not Found error.
- **Product without a Supplier:** The query uses an outerjoin to the Suppliers table. This ensures products are included even if they lack a supplier. The code then conditionally builds the supplier object in the final response.
- **Performance:** For very large datasets, checking InventoryHistory inside a loop could be inefficient (N+1 problem). A more optimized solution would use a subquery or a single, more complex JOIN to filter for items with recent sales in one database query.
- **Readability:** The approach separates the initial data fetching from the business logic filtering (recent sales check). This improves code readability and makes it easier to modify the logic for "recent activity" in the future without changing the main database query.