Assignment (Backend Intern at Bynry Services)

Name: A. Dinesh

Gmail: dineshaitha29@gmail.com

Part 1: Code Review & Debugging (30 minutes)

1. No SKU Uniqueness Check

Issue:
The code does not verify whether the provided SKU already exists in the system before creating a new product.

Impact:
This violates a core business rule that SKUs must be unique across the entire platform. If two products share the same SKU, it can lead to major confusion in inventory management, incorrect stock tracking, and errors in downstream processes like sales, billing, and reporting.

2. No Database Transaction Handling

Issue:
The function commits the product and inventory data separately using two db.session.commit() calls.

Impact:
If the first commit (creating the product) succeeds and the second commit (updating inventory) fails, it results in a partially saved state. This creates data inconsistency, which is difficult to detect and fix later. In production systems, such partial transactions can corrupt reporting, analytics, and restocking workflows.

3. Price Treated as Float

Issue:
The price is stored using the float type directly from the request.

Impact:
Floats are not reliable for storing currency values due to precision issues. This can lead to subtle rounding errors, especially in cumulative

calculations, and may result in inaccurate billing or mismatched accounting reports.

4. No Error Handling / Try-Except Block

Issue:
The function lacks structured error handling. Any missing field or database error will raise an uncaught exception.

Impact:
This can crash the server or result in a generic 500 Internal Server Error with no helpful feedback to the client. Also, if an exception occurs after the first commit, it may leave the database in an inconsistent state without rolling back the earlier changes.

5. No Field Validation

Issue:
There is no validation to check whether required fields like name, sku, price, etc., are present and correctly formatted.

Impact:
Missing or malformed data could cause runtime exceptions (like KeyError or TypeError). Furthermore, invalid data could be saved in the system, impacting stock operations and analytics.

6. Incorrect Data Modeling: warehouse_id inside Product

Issue:
The product creation logic includes a warehouse_id as part of the product model.

Impact:
This implies a one-to-one relationship between products and warehouses. However, the business logic states that products can exist in multiple warehouses. This structure would prevent accurate tracking of inventory per warehouse and hinder expansion to multi-location support.

7. Warehouse Existence Not Verified

Issue:
The code assumes the warehouse exists and does not validate the warehouse_id.

Impact:
If the warehouse_id refers to a non-existent warehouse, the inventory record could reference an invalid foreign key, causing integrity violations or silent logical errors in reporting and analytics.

8. Initial Quantity Not Validated

Issue:
There is no check to ensure that the initial_quantity is a non-negative integer.

Impact:
Invalid values like negative numbers, strings, or null may be inserted into the inventory table. This can break the stock logic, generate incorrect low-stock alerts, or lead to application crashes during stock calculations.

9. Missing Handling for Optional Fields

Issue:
All fields are accessed directly (e.g., data['price']) without accounting for which are required and which may be optional.

Impact:
This can cause unnecessary crashes for missing optional fields and reduces the flexibility of the API to evolve over time. APIs should handle optional fields gracefully using default values or by allowing them to be omitted.

10. Response Format Missing Status Code and JSON Formatting

Issue:
The API returns a plain dictionary with no explicit HTTP status code or content-type headers.

Impact:
Clients may not interpret the result correctly. For instance, a successful

creation should return HTTP status 201 (“Created”), not the default 200. Also, using jsonify() ensures the Content-Type is set to application/json.

Fixes Provided:

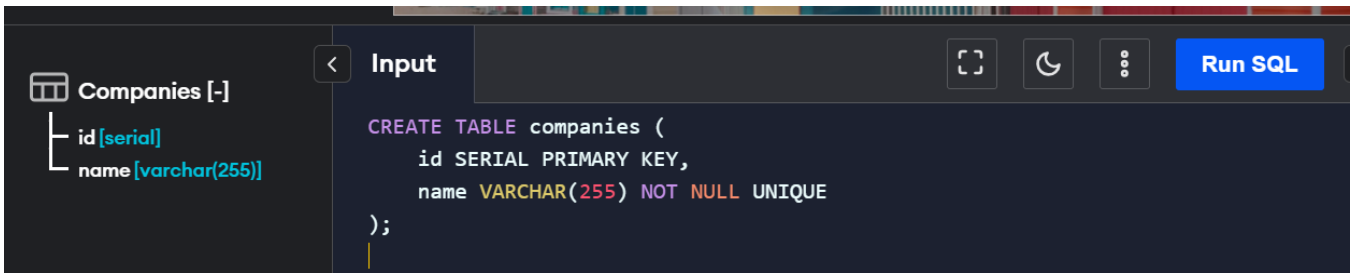
```
from flask import Flask, request, jsonify
from decimal import Decimal
from sqlalchemy.exc import IntegrityError
from models import Product, Inventory, Warehouse
from app import db, app
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json()
    required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
    missing = [f for f in required_fields if f not in data]
    if missing:
        return jsonify({"error": f"Missing required fields: {', '.join(missing)}"}), 400
    existing_product = Product.query.filter_by(sku=data['sku']).first()
    if existing_product:
        return jsonify({"error": "SKU already exists"}), 400
    warehouse = Warehouse.query.get(data['warehouse_id'])
    if not warehouse:
        return jsonify({"error": "Warehouse not found"}), 404
    try:
        quantity = int(data['initial_quantity'])
        if quantity < 0:
            raise ValueError()
    except:
        return jsonify({"error": "Invalid initial quantity"}), 400
    try:
        price = Decimal(str(data['price']))
    except:
        return jsonify({"error": "Invalid price format"}), 400
    try:
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=price
        )
        db.session.add(product)
```

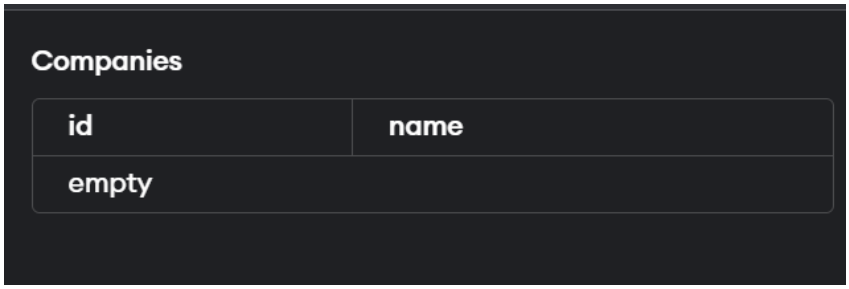
```
        db.session.flush()
        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=quantity
        )
        db.session.add(inventory)
        db.session.commit()
        return jsonify({"message": "Product created", "product_id": 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
```

Part 2: Database Design (25 minutes)

1. Design Schema: Create tables with columns, data types, and relationships
2. Identify Gaps: List questions you'd ask the product team about missing requirements
3. Explain Decisions: Justify your design choices (indexes, constraints, etc.)

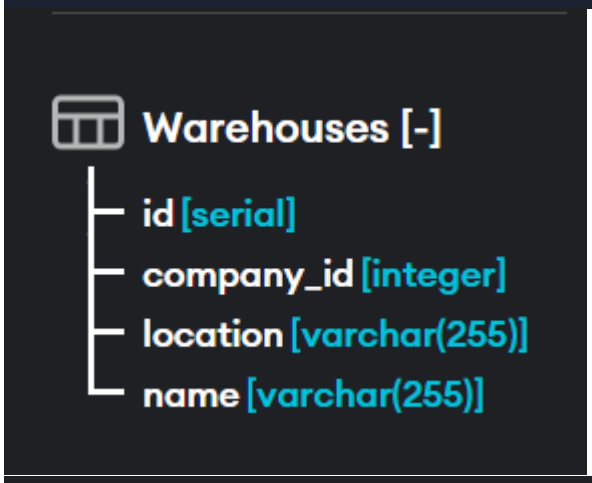
Format: Use any notation (SQL DDL, ERD, text description, etc.)





Creating table warehouses:

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

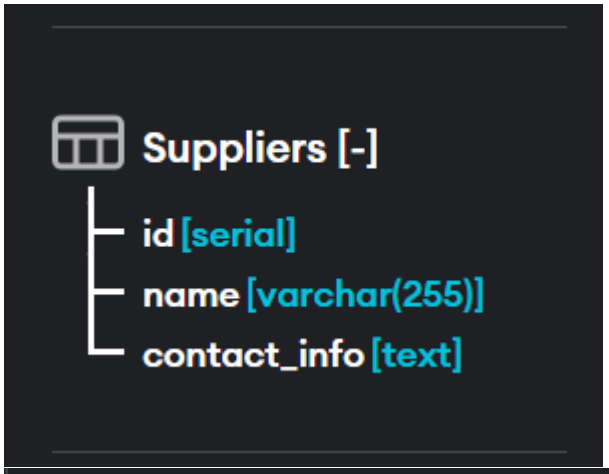


Warehouses

id	company_id	location	name
empty			

Creating table suppliers

```sql
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    contact_info TEXT
);
```

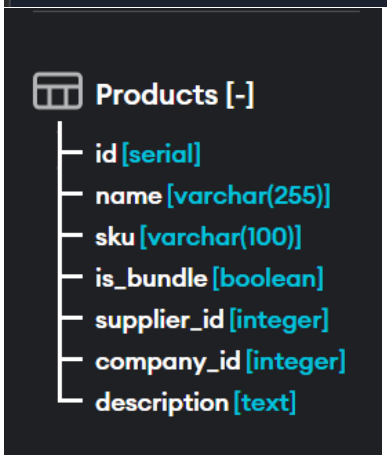


Suppliers

id	name	contact_info
empty		

Creating table products

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) NOT NULL UNIQUE,
    is_bundle BOOLEAN DEFAULT FALSE,
    supplier_id INTEGER REFERENCES suppliers(id),
    company_id INTEGER REFERENCES companies(id),
    description TEXT
);
```



Products

id	name	sku	is_bundle	supplier_id
empty				

Products

ndle	supplier_id	company_id	description

product_bundles (for bundles containing other products)

```
CREATE TABLE product_bundles (
    bundle_id INTEGER REFERENCES products(id),
    product_id INTEGER REFERENCES products(id),
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (bundle_id, product_id),
    CHECK (bundle_id <> product_id)
);
```

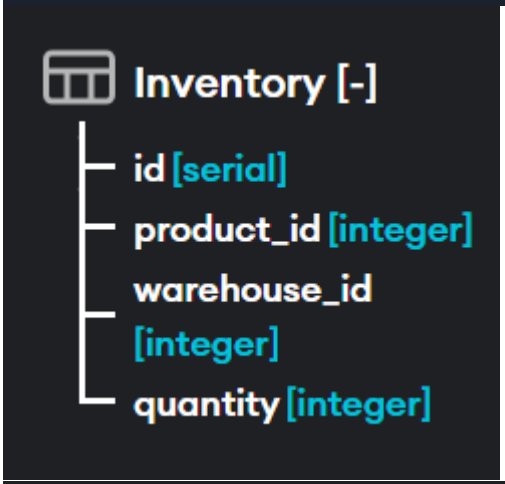
Product_bundles
[-]
- bundle_id [integer]
- product_id [integer]
- quantity [integer]

Product_bundles

bundle_id	product_id	quantity
empty		

Create table inventory

```
CREATE TABLE inventory (
    id SERIAL PRIMARY KEY,
    product_id INTEGER REFERENCES products(id),
    warehouse_id INTEGER REFERENCES warehouses(id),
    quantity INTEGER NOT NULL CHECK (quantity >= 0),
    UNIQUE(product_id, warehouse_id)
);
```

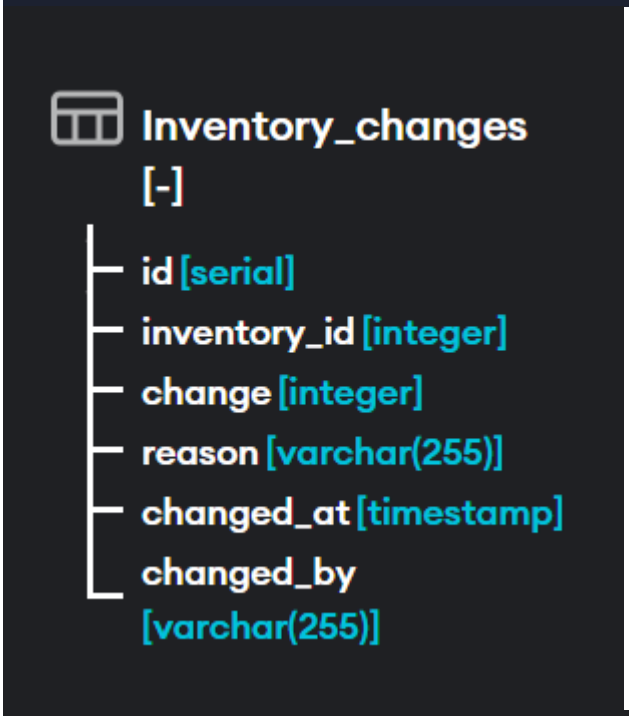


Inventory

id	product_id	warehouse_id	quantity
empty			

Create table inventory_changes (tracking inventory movements)

```sql
CREATE TABLE inventory_changes (
    id SERIAL PRIMARY KEY,
    inventory_id INTEGER NOT NULL,
    change INTEGER NOT NULL,
    reason VARCHAR(255),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    changed_by VARCHAR(255),
    FOREIGN KEY (inventory_id) REFERENCES inventory(id)
);
```



Inventory_changes

id	inventory_id	change	reason	chan
empty				

Inventory_changes

change	reason	changed_at	changed_by

2. Identify Gaps – Questions for the Product Team

Inventory Change Tracking

- Should we track **who** made each inventory change (e.g., user ID, system name)?
- Do inventory changes need a **category** or type (e.g., sale, restock, manual adjustment)?

Bundles

- Can a **bundle contain other bundles** (i.e., nested structures)?
- Are bundles meant to be **fixed sets of products**, or should they support dynamic configuration?

Suppliers

- Can a **product be associated with multiple suppliers**?
- Is it possible for **different companies to use the same supplier**?

Warehouse-Company Relationship

- Are **warehouses exclusive** to a company, or can multiple companies share a warehouse?

Stock Thresholds or Alerts

- Do we need to support **minimum stock levels** for alerts or automatic reordering?
- Should these thresholds be set **per product, per warehouse, or globally**?

Product Customization

- Can the **same SKU** have different details (like description or price) **depending on the company**?

Pricing

- Should pricing be **tracked over time** (price history)?
- Does the system require different prices **per warehouse or per supplier**?

Deletion Behavior

- When a product is deleted:
    - Should its related **inventory**, **bundle data**, and **change logs** also be removed?
    - Or should it be a **soft delete** (mark as inactive)?

3. Design Decisions

Normalization

- Database is structured in **Third Normal Form (3NF)**:
    - No duplicated columns or repeating groups.
    - Each table has a **single purpose** (e.g., inventory, products, bundles).
    - Eliminates redundancy and improves maintainability.

Relationships & Constraints

- Used **foreign keys** with ON DELETE CASCADE where appropriate, to ensure cleanup of dependent data.
- Ensured uniqueness:
  - sku in the products table.
  - Combination of product_id + warehouse_id in inventory.
- Applied **CHECK constraints**:
  - Prevent negative quantities.
  - Prevent self-referencing in bundle relationships (bundle_id <> product_id).

Indexing

- **Implicit indexes** are created on:
  - Primary keys
  - Foreign keys
  - Unique fields (e.g., sku, company_id + name)
- Considered **additional indexes** for:
  - inventory_changes.changed_at → for audit reports and date-based queries.
  - inventory.quantity → for low-stock checks and alert systems.

Scalability

- Designed to scale with business growth:
  - Supports **many-to-many** between products and warehouses.
  - Allows tracking of **inventory changes over time**.
  - Flexible enough to accommodate **product bundles**.
  - Company-specific data structures ensure **multi-tenant** support.

Part 3: API Implementation (35 minutes)

API Implementation: Low Stock Alert Endpoint

Tech Stack Used

- Language: Python
- Framework: Flask
- ORM: SQLAlchemy
- Database: Assumes PostgreSQL or SQLite

1. Assumptions Made

Before jumping into code, here are some practical assumptions based on the business rules:

1. Each product has a **low stock threshold** stored in a product_thresholds table.
2. **Recent sales activity** is pulled from a sales table.
3. Inventory is tracked per product per warehouse in the inventory table.

4. Supplier details are linked directly to the products table.
5. We only alert for products that had sales in the **last 30 days**.
6. We calculate days_until_stockout based on recent average sales per day.

2. Flask Route Implementation

python
CopyEdit

```
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime, timedelta
from sqlalchemy import func

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///inventory.db'
db = SQLAlchemy(app)

# Simplified models just to illustrate structure
class Company(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)

class Warehouse(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    company_id = db.Column(db.Integer, db.ForeignKey('company.id'))

class Supplier(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    contact_email = db.Column(db.String)

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    sku = db.Column(db.String)
    company_id = db.Column(db.Integer, db.ForeignKey('company.id'))
    supplier_id = db.Column(db.Integer, db.ForeignKey('supplier.id'))

class ProductThreshold(db.Model):
```

```
    product_id = db.Column(db.Integer, db.ForeignKey('product.id'),
primary_key=True)
    threshold = db.Column(db.Integer)

class Inventory(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    product_id = db.Column(db.Integer, db.ForeignKey('product.id'))
    warehouse_id = db.Column(db.Integer, db.ForeignKey('warehouse.id'))
    quantity = db.Column(db.Integer)

class Sale(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    product_id = db.Column(db.Integer)
    warehouse_id = db.Column(db.Integer)
    quantity = db.Column(db.Integer)
    sale_date = db.Column(db.DateTime)

@app.route('/api/companies/<int:company_id>/alerts/low-stock',
methods=['GET'])
def low_stock_alerts(company_id):
    # Look back over the last 30 days
    cutoff = datetime.utcnow() - timedelta(days=30)

    # Aggregate sales in that window
    sales_subquery = db.session.query(
        Sale.product_id,
        Sale.warehouse_id,
        func.sum(Sale.quantity).label('recent_sales')
    ).join(Product, Product.id == Sale.product_id
    ).filter(
        Product.company_id == company_id,
        Sale.sale_date >= cutoff
    ).group_by(
        Sale.product_id, Sale.warehouse_id
    ).subquery()

    # Main query
    results = 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'),
        ProductThreshold.threshold,
        Supplier.id.label('supplier_id'),
        Supplier.name.label('supplier_name'),
        Supplier.contact_email,
        sales_subquery.c.recent_sales
    ).join(Product, Product.id == Inventory.product_id
    ).join(Warehouse, Warehouse.id == Inventory.warehouse_id
    ).join(ProductThreshold, ProductThreshold.product_id == Product.id
    ).join(Supplier, Supplier.id == Product.supplier_id
    ).join(sales_subquery, (sales_subquery.c.product_id ==
Inventory.product_id) &
                           (sales_subquery.c.warehouse_id == Inventory.warehouse_id)
    ).filter(
        Product.company_id == company_id,
        Inventory.quantity < ProductThreshold.threshold
    ).all()

    # Build response
    alerts = []
    for row in results:
        avg_daily_sales = row.recent_sales / 30
        if avg_daily_sales > 0:
            days_left = int(row.current_stock / avg_daily_sales)
        else:
            days_left = 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": days_left,
            "supplier": {
```

```
            "id": row.supplier_id,
            "name": row.supplier_name,
            "contact_email": row.contact_email
        }
    })

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

3. Edge Cases Handled

- **No sales activity** in last 30 days? Product is skipped.
- **Threshold not defined?** The product won't show up (due to join).
- **Division by zero** for stockout days? Safely handled using a check.
- **Multiple warehouses**? Fully supported via warehouse_id joins.