Part 1: Code Review & Debugging

Issue1:
Check the required fields.
The code does check if the fields like name, price, SKU and `warehouse_id` are entered by the user.

Impact:
On missing values which are not checked,there will be issues in db.Like for example if we do not add certain fields like name, SKU for product, it still gets registered.The database might crash.

Solution:
Use validation and check that all the required field are entered by the user while creating the product . We should add checks for the length of fields as well.

```
required_fields = ['name', 'sku', 'price', 'warehouse_id',
'initial_quantity']
missing = [f for f in required_fields if not data.get(f)]
if missing:
    return {"error": f"Missing required fields: {',
'.join(missing)}"}, 400

Add additional checks for length or type:

if len(data['name']) > 255:
    return {"error": "Product name too long"}, 400
```

Issue2:
SKU refers to *stock-keeping unit* that tracks each product . No checks for uniqueness of this values are there.

Impact:
This might lead to having duplicate sku values.This will lead to inconsistencies in database. For example,two product gets same sku then how will we track down the product we are looking for.The paths may clash with each other.

Solution:
Check if entered SKU already exits or not . We can add Unique constraint here.Adding this validation can solve this issue.

```
existing_sku = Product.query.filter_by(sku=data['sku']).first()
if existing_sku:
    return {"error": "SKU already exists"}, 409

Also enforce this at the model level:

sku = db.Column(db.String(100), unique=True, nullable=False)
```

Issue3:
Price constraints are not defined.

Impact:
The user can add negative value to a certain price.That will be absurd. So,we need to prepare our code for this.

Solution:
Use Decimal to validate the format.

```
from decimal import Decimal, InvalidOperation

try:
    price = Decimal(str(data['price']))
    if price < 0:
        return {"error": "Price cannot be negative"}, 400
except InvalidOperation:
    return {"error": "Invalid price format"}, 400
```

Issue4:
Commit is done halfway .Two separate commits are made .

Impact:
What if the errors gets while making inventory . So ,the product will be made without the inventory as the second commit would not be executed but the product is made . It will violate the integrity of transaction.

Solution:
Add commit at the end of both the transactions.

```
try:
    with db.session.begin_nested():  # Safe transaction block
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=price
        )
        db.session.add(product)
        db.session.flush()  # Assigns product.id

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

    db.session.commit()
    return {"message": "Product created", "product_id": product.id},
201
except Exception as e:
    db.session.rollback()
    return {"error": "Transaction failed", "details": str(e)}, 500
```

Issue5:
Products can exist in multiple warehouses, but still here a products gets associated with single warehouse only.

Impact:
Products can exist in multiple warehouses.But this code does not give it that liberty. It says that a product can be in a single warehouse only.But this condition does not apply in distribution system.They tend to have same product at multiple locations for supply management.

Solution:

Remove warehouse_id from product.

```
# models.py
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255), nullable=False)
    sku = db.Column(db.String(100), unique=True, nullable=False)
    price = db.Column(db.Numeric(10, 2), nullable=False)
    # ✅ Remove warehouse_id from here

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

Issue6:
No proper error handling is there.

Impact:
Not all the checked so it may lead to failures in the syatem . It could return HTML errors instead of JSON .

Solution:
Add try and except statements for proper error handling.

```
try:
    # all processing logic
except IntegrityError as e:
    db.session.rollback()
    return {"error": "Database integrity error", "details": str(e)},
500
except Exception as e:
    db.session.rollback()
    return {"error": "Unexpected error", "details": str(e)}, 500
```

Issue7:
Proper HTTP status codes are not used.

Impact:
The developer can estimate the state of any API by referring to the codes associated. So as a good and efficient developer ,it is important to add codes for various conditions so that debugging can be done easily.

Solution:
Return responses with HTTP status code.

```
# Success
return {"message": "Product created", "product_id": product.id}, 201

# Bad Request
return {"error": "Invalid input"}, 400

# Conflict
return {"error": "SKU already exists"}, 409
```

Issue8:
Using product_id before inserting it into the db and ensuring it works fine.

Impact:
Look,if we use product_id before committing the changes.The product_id field wil be none for future use. It inserts Inventory will null ,leads to foreign key error.

Solution:
We need to use flush() in order to insert it into the database and to make it available for future references.

```
db.session.add(product)
db.session.flush()  # Safely assigns product.id before using it

# Now we can reference product.id
```

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

Part 2: Database Design

Companies

```
CREATE TABLE companies (
    id PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

Warehouses

```
CREATE TABLE warehouses (
    id PRIMARY KEY,
    company_id INTEGER NOT NULL REFERENCES companies(id) ON DELETE
CASCADE,
    location VARCHAR(255)
);
```

Products

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

Bundled Products

```
CREATE TABLE product_bundles (
    bundle_id INTEGER NOT NULL REFERENCES products(id) ON DELETE
CASCADE,
    component_id INTEGER NOT NULL REFERENCES products(id) ON DELETE
CASCADE,
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (bundle_id, component_id)
);
```

Suppliers

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

CREATE TABLE supplier_products (
    supplier_id INTEGER NOT NULL REFERENCES suppliers(id) ON DELETE
CASCADE,
    product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE
CASCADE,
    PRIMARY KEY (supplier_id, product_id)
);
```

Inventory (quantity of product in a warehouse)

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

```
CREATE TABLE inventory_log (
    id SERIAL PRIMARY KEY,
    warehouse_id INTEGER NOT NULL REFERENCES warehouses(id) ON DELETE
CASCADE,
    product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE
CASCADE,
    quantity_change INTEGER NOT NULL,
    changed_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    reason TEXT
);
```

Questions to be asked

1. What about the SKU of the bundles.SKU will be separate or will be from its parts.
2. How can supplier supply bundles and manage it?
3. Are warehouses company specific?

Explanation

Constraints:
Proper Unique values and constraints are used here.

Normalisation:
All the entities are normal

Bundle Supprt:
Here we have used a combination of bundle_id, component_id as the primary key which makes it efficient.

Part 3: API Implementation

```
class Company(models.Model):
    name = models.CharField(max_length=255)

class Warehouse(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
```

```python
    name = models.CharField(max_length=255)

class Product(models.Model):
    name = models.CharField(max_length=255)
    sku = models.CharField(max_length=100, unique=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    low_stock_threshold = models.IntegerField(default=10)

class Supplier(models.Model):
    name = models.CharField(max_length=255)
    contact_email = models.EmailField()

class SupplierProduct(models.Model):
    supplier = models.ForeignKey(Supplier, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)

class Inventory(models.Model):
    warehouse = models.ForeignKey(Warehouse, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.IntegerField()

class Sale(models.Model):
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    warehouse = models.ForeignKey(Warehouse, on_delete=models.CASCADE)
    sold_at = models.DateTimeField(auto_now_add=True)




# serializers.py

from rest_framework import serializers

class SupplierSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField()
    contact_email = serializers.EmailField()
```

```
class LowStockAlertSerializer(serializers.Serializer):
    product_id = serializers.IntegerField()
    product_name = serializers.CharField()
    sku = serializers.CharField()
    warehouse_id = serializers.IntegerField()
    warehouse_name = serializers.CharField()
    current_stock = serializers.IntegerField()
    threshold = serializers.IntegerField()
    days_until_stockout = serializers.IntegerField()
    supplier = SupplierSerializer()



# views.py

from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from django.utils.timezone import now
from datetime import timedelta
from .models import (
    Product, Inventory, Warehouse, Company,
    SupplierProduct, Supplier, Sale
)
from .serializers import LowStockAlertSerializer
from django.db.models import F, Q

class LowStockAlertView(APIView):
    def get(self, request, company_id):
        try:
            # Step 1: Get all warehouses for the company
            warehouses =
Warehouse.objects.filter(company_id=company_id)
            warehouse_ids = warehouses.values_list('id', flat=True)

            # Step 2: Get products with recent sales in these
warehouses (last 30 days)
            cutoff_date = now() - timedelta(days=30)
            recent_product_ids = Sale.objects.filter(
                warehouse_id__in=warehouse_ids,
                sold_at__gte=cutoff_date
            ).values_list('product_id', flat=True).distinct()

            # Step 3: Get inventories below threshold and having
recent sales
            inventory_qs = Inventory.objects.select_related(
                'product', 'warehouse'
            ).filter(
                warehouse_id__in=warehouse_ids,
                product_id__in=recent_product_ids,
                quantity__lt=F('product__low_stock_threshold')
            )

            alerts = []

            for inventory in inventory_qs:
                product = inventory.product
                warehouse = inventory.warehouse

                # Estimate days until stockout (e.g., 5/day usage)
                daily_usage = 5
                days_left = inventory.quantity // daily_usage if
daily_usage else None

                # Get a supplier for this product
                supplier_product =
SupplierProduct.objects.filter(product=product).select_related('suppli
er').first()
                supplier = supplier_product.supplier if
supplier_product else None

                if supplier:
                    alerts.append({
                        "product_id": product.id,
                        "product_name": product.name,
                        "sku": product.sku,
                        "warehouse_id": warehouse.id,
                        "warehouse_name": warehouse.name,
                        "current_stock": inventory.quantity,
                        "threshold": product.low_stock_threshold,
                        "days_until_stockout": days_left,
                        "supplier": {
                            "id": supplier.id,
                            "name": supplier.name,
                            "contact_email": supplier.contact_email
                        }
                    })

            return Response({
                "alerts": alerts,
                "total_alerts": len(alerts)
            }, status=status.HTTP_200_OK)

        except Exception as e:
            return Response({
                "error": "Server error",
                "details": str(e)
            }, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

Edge cases :

1. **No warehouses present for a given company**
   a. Querying inventory or sales to a company with no warehouses it could result in empty or invalid results.
   b. To manage this, the API retrives all warehouse IDs and uses them in filters, if none exist, next queries return empty results with the final response containing an empty alerts list with “total_alerts = 0”

2. **Product with no recent sales**
   a. Low stock alerts should only trigger for actively selling products

b. The API filters products based on recent entries in the Sale model using a time cutoff now() - timedelta(days = 30)

3. **Products has no supplier assigned**
    a. A product with no yet linked supplied in the database
    b. Trying to access supplier_product.supplier withour its exsitence will raise an error
    c. To handle this , API uses .first() on the supplier_product relationship and checks for supplier before adding the alert.

4. **Product stock is not low**
    a. To avoid false errors, API checks if the current inventory for a product is below its low_stock_threshold.
    b. This is done using **quantity__lt=F('product__low_stock_threshold')** filter in the query.

5. **Division by ZERO in days until stockout**
    a. While estimating day_until_stockout, dividing by zero can cause a Division Error
    b. If the API's hardcoded usage rate is zero it safely returns None for days left

6. **Unexpected server or Database errors**
    a. Any unexpected error can cause the API and expose sensitive errors.
    b. To manage this the entire logic is wrapped within a error handling block i.e try-except