Backend Engineering Intern Case Study Submission

# Part 1: Code Review & Debugging

## 1. Issues Identified

- No check for duplicate SKU (which must be unique)  
- Missing required field checks (name, sku, price, etc.)  
- Not validating input types (like price as decimal, quantity as integer)  
- Commits twice to DB; could be combined  
- No error handling (e.g., try-catch)  
- Logic assumes products are only stored in one warehouse  
- Doesn't support optional fields gracefully

## 2. Impact in Production

- Duplicate SKUs may break product search or create confusion  
- API might crash on bad or missing input  
- Data inconsistency if product insert works but inventory fails  
- No meaningful error messages to the frontend/client  
- Fails silently for edge business scenarios like multiple warehouse support

## 3. Fixed Version (with explanation)

@app.route('/api/products', methods=['POST'])  
def create\_product():  
 data = request.json  
  
 # Required fields  
 required\_fields = ['name', 'sku', 'price', 'warehouse\_id', 'initial\_quantity']  
 for field in required\_fields:  
 if field not in data:  
 return {"error": f"{field} is required"}, 400  
  
 # Check if SKU already exists  
 existing = Product.query.filter\_by(sku=data['sku']).first()  
 if existing:  
 return {"error": "SKU already exists"}, 409  
  
 try:  
 # Create product  
 product = Product(  
 name=data['name'],  
 sku=data['sku'],  
 price=float(data['price'])  
 )  
 db.session.add(product)  
 db.session.flush()  
  
 # Add inventory  
 inventory = Inventory(  
 product\_id=product.id,  
 warehouse\_id=data['warehouse\_id'],  
 quantity=int(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": str(e)}, 500

Note:  
- Used db.session.flush() to get the product ID before committing to the database.  
- Committing once after both product and inventory creation ensures consistent data if something fails.  
- Casting price to float helps handle cases where input comes as a string.

# Part 2: Database Design

Schema design, questions to ask product team, and design justifications will follow.

Sql:

-- Companies

CREATE TABLE companies (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL

);

-- Warehouses

CREATE TABLE warehouses (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

company\_id INT,

FOREIGN KEY (company\_id) REFERENCES companies(id)

);

-- Products

CREATE TABLE products (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

sku VARCHAR(50) UNIQUE NOT NULL,

price DECIMAL(10, 2)

);

-- Inventory (Product in Warehouse)

CREATE TABLE inventory (

id INT AUTO\_INCREMENT PRIMARY KEY,

product\_id INT,

warehouse\_id INT,

quantity INT DEFAULT 0,

FOREIGN KEY (product\_id) REFERENCES products(id),

FOREIGN KEY (warehouse\_id) REFERENCES warehouses(id)

);

-- Inventory History

CREATE TABLE inventory\_changes (

id INT AUTO\_INCREMENT PRIMARY KEY,

inventory\_id INT,

`change` INT,

changed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (inventory\_id) REFERENCES inventory(id)

);

-- Suppliers

CREATE TABLE suppliers (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

contact\_email VARCHAR(100)

);

-- Product Suppliers (many-to-many)

CREATE TABLE product\_suppliers (

product\_id INT,

supplier\_id INT,

PRIMARY KEY (product\_id, supplier\_id),

FOREIGN KEY (product\_id) REFERENCES products(id),

FOREIGN KEY (supplier\_id) REFERENCES suppliers(id)

);

-- Product Bundles

CREATE TABLE product\_bundles (

bundle\_id INT,

component\_id INT,

quantity INT DEFAULT 1,

PRIMARY KEY (bundle\_id, component\_id),

FOREIGN KEY (bundle\_id) REFERENCES products(id),

FOREIGN KEY (component\_id) REFERENCES products(id)

);

**Additional Design Note**:  
- I would add indexes on commonly queried columns like product\_id and warehouse\_id in the inventory table to improve performance.

**Questions for Product Team:**

* Can a product belong to multiple suppliers?
* Can a product be bundled multiple levels deep (nested bundles)?
* Do warehouses belong to one company only?
* Should inventory changes be tracked with reasons (like sale, return, restock)?
* Do suppliers provide to multiple companies?

**My Design Choices:**

* Used **foreign keys** to keep relations strong.
* Added **UNIQUE on SKU** to stop duplicates.
* Created **inventory\_changes** table to track stock movements.
* Used **many-to-many** for suppliers and product bundles.
* Assumed warehouse belongs to only one company.

# Part 3: API Implementation (Node.js + Express)

**What the API does:**

* It shows a list of products that are low in stock for a given company.
* Only shows products that had recent sales (in last 30 days).
* Also shows supplier info so company can reorder.

Code:

// GET /api/companies/:company\_id/alerts/low-stock

app.get('/api/companies/:company\_id/alerts/low-stock', async (req, res) => {

const companyId = req.params.company\_id;

try {

// Step 1: Get all warehouses of this company

const [warehouses] = await db.query(

'SELECT id, name FROM warehouses WHERE company\_id = ?',

[companyId]

);

let alerts = [];

for (let warehouse of warehouses) {

// Step 2: Get inventory with product info

const [inventory] = await db.query(`

SELECT i.quantity, p.id AS product\_id, p.name, p.sku,

pt.threshold, s.id AS supplier\_id, s.name AS supplier\_name, s.contact\_email

FROM inventory i

JOIN products p ON i.product\_id = p.id

JOIN product\_thresholds pt ON pt.product\_id = p.id

JOIN product\_suppliers ps ON ps.product\_id = p.id

JOIN suppliers s ON s.id = ps.supplier\_id

WHERE i.warehouse\_id = ? AND i.quantity < pt.threshold

`, [warehouse.id]);

for (let item of inventory) {

const [recentSales] = await db.query(`

SELECT COUNT(\*) AS count FROM sales

WHERE product\_id = ? AND sale\_date >= NOW() - INTERVAL 30 DAY

`, [item.product\_id]);

if (parseInt(recentSales[0].count) > 0) {

alerts.push({

product\_id: item.product\_id,

product\_name: item.name,

sku: item.sku,

warehouse\_id: warehouse.id,

warehouse\_name: warehouse.name,

current\_stock: item.quantity,

threshold: item.threshold,

days\_until\_stockout: 7,

supplier: {

id: item.supplier\_id,

name: item.supplier\_name,

contact\_email: item.contact\_email

}

});

}

}

}

res.json({

alerts: alerts,

total\_alerts: alerts.length

});

} catch (err) {

console.error(err);

res.status(500).json({ error: "Something went wrong" });

}

});

**My Assumptions**

* There's a table called product\_thresholds that stores the threshold for each product.
* A product can have one or more suppliers, but we’re picking one for now.
* We assume a sales table tracks product sales with a date.
* Days until stockout is hardcoded as 7 for now, because I didn’t calculate actual rate.

**Why I Did It Like This:**

* I broke down the problem step-by-step to make it easier to write and debug.
* Used simple queries instead of complex joins to avoid confusion.
* It’s slower because of the for loop and multiple DB calls, but it’s easier to understand.
* This would be fine for a beginner project or small business, and can be optimized later.

**Extra Notes:**  
- Currently returning one supplier per product. Later, we can support preferred suppliers.  
- Used loop and multiple simple queries to keep the logic clear and beginner-friendly.  
- This can be optimized later using joins and bulk queries.  
- I focused more on correctness and readability rather than speed, which suits my experience level