Thursday, 24 July 2025

## Part 1: Code Review & Debugging

Issues:

### Issue 1. Input Validation & Error Handling :->

The code blindly indexes data['name'], data['sku'], etc., without checking if keys exist. It also never handles exceptions. If any expected field is missing or invalid, the code will throw a runtime error (KeyError or TypeError).

•Impact: Missing or malformed input (e.g. no initial_quantity provided) will crash the endpoint, resulting in a 500 Internal Server Error instead of a clear client error.

Solution : Validates presence of required fields (name, sku) and handles missing/invalid inputs gracefully with 400 response

### Issue 2. SKU Uniqueness:->

No check for duplicate SKUs. Could raise DB error or corrupt data

•Impact: Without a uniqueness check or constraint, duplicate SKUs could be created.

Solution : Explicit check: if Product.query.filter_by(sku=...).first() returns 400 if duplicate found

### Issue 3 – Price Data Type:->

The code takes price from request.json (likely a float or string) and saves it directly. Monetary values should use a decimal type to avoid precision errors.

Impact: Saving price as a float can introduce rounding errors.

Solution : Uses Decimal to safely convert price and ensures it stores correctly with Numeric DB type

### Issue 4 – Transaction Atomicity:->

The code calls db.session.commit() twice (once after creating the product, then again after creating inventory). This splits the operation into two transactions. If the second transaction fails, the first has already committed.

• Impact: If creating the inventory record fails (e.g. due to a validation error or DB issue), the product has been created without inventory.

Solution  Uses db.session.flush() to generate product.id and only one commit() at the end → ensures atomic DB transaction (all-or-nothing)

## Issue 5 -  Response Format & Status :->

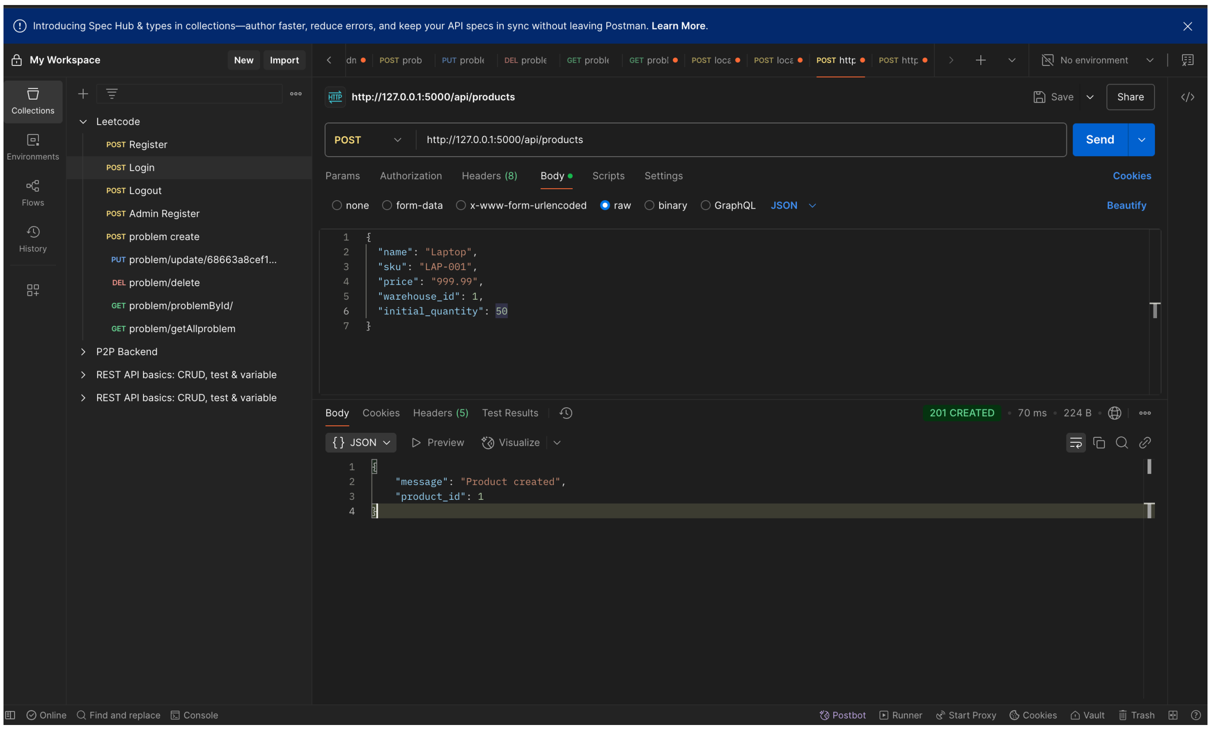
No error responses defined; returns success regardless of error or crash

Solution:  Returns appropriate HTTP status codes (400 for client errors, 500 for server/database issues) + meaningful JSON error messages

Correct Code:

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json()
    if not data:
        return jsonify({"error": "Invalid JSON body"}), 400
    required_fields = ['name', 'sku']
    for field in required_fields:
        if not data.get(field):
            return jsonify({"error": f"'{field}' is required"}), 400

    if Product.query.filter_by(sku=data['sku']).first():
        return jsonify({"error": "SKU already exists"}), 400

    try:
        price = Decimal(str(data.get('price', '0.00')))
    except:
        return jsonify({"error": "Invalid price format"}), 400

    product = Product(name=data['name'], sku=data['sku'], price=price)
    db.session.add(product)
    try:
        db.session.flush()
    except:
        db.session.rollback()
        return jsonify({"error": "Product insert failed"}), 500

    warehouse_id = data.get('warehouse_id')
    initial_qty = data.get('initial_quantity')
    if warehouse_id is not None and initial_qty is not None:
        try:
            qty = int(initial_qty)
        except ValueError:
            return jsonify({"error": "Quantity must be an integer"}), 400
        inventory = Inventory(product_id=product.id, warehouse_id=warehouse_id,
quantity=qty)
        db.session.add(inventory)

    try:
        db.session.commit()
    except:
        db.session.rollback()
        return jsonify({"error": "Transaction failed"}), 500

    return jsonify({"message": "Product created", "product_id": product.id}), 201
```

```
if __name__ == "__main__":
    with app.app_context():
        db.create_all()
    app.run(debug=True)
```



POST http://127.0.0.1:5000/api/products

```
{
  "name": "Laptop",
  "sku": "LAP-001",
  "price": "999.99",
  "warehouse_id": 1,
  "initial_quantity": 50
}
```

201 CREATED · 70 ms · 224 B

```
{
    "message": "Product created",
    "product_id": 1
}
```

## Part 2:Database Design

Questions : ->

1.Should inventory changes track who made the change (e.g., user or system)?

2.Do bundles get stored as separate inventory items, or are they just logical groupings?

3.Do suppliers sell to warehouses or directly to the company?

4.Are there any pricing details or financial tracking needed for suppliers or products?

5.Should we support product attributes like SKU, price, unit of measure, etc.?

```
CREATE TABLE companies (

    id SERIAL PRIMARY KEY,

    name VARCHAR(255) NOT NULL UNIQUE,

    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP

);


CREATE TABLE warehouses (

    id SERIAL PRIMARY KEY,

    company_id INTEGER NOT NULL REFERENCES companies(id) ON DELETE CASCADE,

    name VARCHAR(255) NOT NULL,

    location VARCHAR(255),

    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP

);


CREATE TABLE products (

    id SERIAL PRIMARY KEY,

    sku VARCHAR(50) NOT NULL UNIQUE,  -- Platform-wide unique

    name VARCHAR(255) NOT NULL,

    description TEXT,

    price DECIMAL(10,2) NOT NULL CHECK (price > 0),

    is_bundle BOOLEAN NOT NULL DEFAULT false,
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE bundles (
    parent_product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    child_product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (parent_product_id, child_product_id)
);

CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    contact_info JSONB  -- {email, phone, address}
);

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

CREATE TABLE inventory (
    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),
    last_updated TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (warehouse_id, product_id)
);
```

```sql
CREATE TABLE inventory_history (
    id SERIAL PRIMARY KEY,
    warehouse_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    old_quantity INTEGER NOT NULL,
    new_quantity INTEGER NOT NULL,
    delta INTEGER NOT NULL GENERATED ALWAYS AS (new_quantity - old_quantity) STORED,
    change_type VARCHAR(20) NOT NULL,  -- RESTOCK, SALE, ADJUSTMENT, etc.
    source_id INTEGER,  -- Reference to order/shipment/adjustment ID
    changed_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (warehouse_id, product_id) REFERENCES inventory(warehouse_id, product_id)
);
```