Lab 5: Implementing the Product Microservice

This lab focuses on the practical implementation of one of the core services identified in Lab 4: the **Product Service**. This service will be completely independent, owning its data and exposing a RESTful API, adhering to the principles of Microservices Architecture.

Objectives

1. Set up a **standalone Flask application** dedicated solely to product management.
2. Implement the **Product Service** logic and persistence (simulated by a database).
3. Expose the defined **Service Contract (REST API)** for reading and searching products.
4. Test the service in isolation.

Technology & Tool Installation

We will continue using **Python/Flask**, but this time we'll introduce **SQLAlchemy** (a Python SQL Toolkit and ORM) and **SQLite** to simulate a dedicated database for this single microservice.

Tool	Purpose	Installation/Setup Guide
Python 3.x	Core programming language.	Ensure Python 3 is installed.
Flask	Lightweight web framework for the API/Presentation Layer.	Run: pip install Flask
SQLAlchemy & **Flask-SQLAlchemy**	Object Relational Mapper (ORM) and Flask integration for database access.	Run: pip install Flask-SQLAlchemy
Postman / cURL	API Testing tool.	Install Postman or use the built-in curl command line tool.

Activity Practice 1: Project Setup and Data Modeling

Goal: Create the project structure and define the Product database schema using SQLAlchemy.

Step-by-Step Instructions & Coding Guide

1. **Create Service Directory:**

Bash

```
# Ensure you are outside the shopsphere_layered directory
mkdir product_service
cd product_service
python -m venv venv
source venv/bin/activate
pip install Flask Flask-SQLAlchemy
touch app.py
```

2. **Initialize Flask and SQLAlchemy:** Set up the basic application and configure the SQLite database (which will live in a file named products.db).

File: app.py (Initial Setup)

Python

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
# Configure SQLite database dedicated only to this service
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

3. **Define the Product Model (Schema):** Map the Product entity to a database table.

File: app.py (Add inside)

Python

```
class Product(db.Model):
    # The primary key for the Product Service's database
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)

    description = db.Column(db.String(500), nullable=True)

    price = db.Column(db.Float, nullable=False)

    stock = db.Column(db.Integer, nullable=False)

    is_active = db.Column(db.Boolean, default=True)


    def to_dict(self):

        # Converts the database object to a dictionary for API response

        return {

            'id': self.id,

            'name': self.name,

            'description': self.description,

            'price': self.price,

            'stock': self.stock,

            'is_active': self.is_active

        }
```

4. **Create Database Tables and Initial Data:**
    - **Action:** Open a Python shell inside the project folder:

Bash

```
python

>>> from app import app, db, Product

>>> with app.app_context():

...     db.create_all() # Creates the products.db file and table

...     # Insert some initial data

...     db.session.add(Product(name='Laptop X1', description='High-performance notebook.', price=1500.00, stock=10))
...     db.session.add(Product(name='Mouse Pro', description='Ergonomic wireless mouse.', price=50.00, stock=50))
...     db.session.commit()
...     print("Database initialized with sample data.")
```

Activity Practice 2: Implementing the Service API

Goal: Implement the REST API endpoints to read product data, fulfilling the service contract defined in Lab 4.

Step-by-Step Instructions & Coding Guide

1. **Implement GET endpoint (List/Search):** Allows retrieving all active products or searching by name.

File: app.py (Add route)

Python

```
@app.route('/api/products', methods=['GET'])
def list_products():
    # Get optional search query from request arguments
    query = request.args.get('q')

    # Start with all active products
    products = Product.query.filter_by(is_active=True)

    if query:
        # Add search filtering (case-insensitive name search)
        products = products.filter(Product.name.like(f'%{query}%'))

    # Execute query and convert results to a list of dictionaries
    return jsonify([p.to_dict() for p in products.all()]), 200
```

2. **Implement GET endpoint (Details):** Allows retrieving a single product by ID.

File: app.py (Add route)

Python

```
@app.route('/api/products/<int:product_id>', methods=['GET'])
def get_product_details(product_id):
    # Query the database for the specific product ID
    product = Product.query.get(product_id)

    if product and product.is_active:
        return jsonify(product.to_dict()), 200
    else:
        # Handle the case where the resource is not found
        return jsonify({'message': 'Product not found or is inactive'}), 404
```

3. **Add Run Block:** Ensure the application runs on a specific port (e.g., 5001) to avoid conflicts with the Layered app (Lab 3) or the Gateway (Lab 6).

File: app.py (End of file)

Python

```
if __name__ == '__main__':
    # Run the microservice on a dedicated port
    app.run(port=5001, debug=True)
```

Activity Practice 3: Isolation Testing

Goal: Verify that the service operates correctly and independently.

Step-by-Step Instructions

1. **Start the Service:**

Bash

```
# Ensure you are in the product_service directory and the virtual environment is active

python app.py
```

2. **Test Product Listing (cURL or Postman):**
   - **Action:** Send an HTTP GET request to list all products.
   - **Command:** curl -X GET http://127.0.0.1:5001/api/products
   - **Expected Result:** A JSON array containing the initial Laptop X1 and Mouse Pro entries (HTTP 200 OK).
3. **Test Product Details Lookup:**
   - **Action:** Send an HTTP GET request to retrieve a specific product (assuming ID 1).
   - **Command:** curl -X GET http://127.0.0.1:5001/api/products/1
   - **Expected Result:** A JSON object detailing 'Laptop X1' (HTTP 200 OK).
4. **Test Error Handling:**
   - **Action:** Send an HTTP GET request for a non-existent ID (e.g., ID 99).
   - **Command:** curl -X GET http://127.0.0.1:5001/api/products/99
   - **Expected Result:** A JSON response with an error message and **HTTP 404 Not Found**.

This lab successfully establishes an independent, data-owning microservice, ready to be integrated into the larger ShopSphere architecture via an API Gateway in the next lab.