Slide Deck: Lecture 5 - Implementing an Independent Microservice

1. Title Slide: Building the Product Service

- **Title:** Software Architecture: Lecture 5
- **Subtitle:** Implementing an Independent Microservice
- **Focus:** Translating the **Product Service** contract (Lecture 4) into a runnable, standalone component.

---

2. Recap: The Microservice Contract

- **ASR Driver: Scalability** demands that the Product Service must be able to deploy and scale independently of the rest of ShopSphere.
- **Key Principle: Data Ownership** – This service must own its data (Product details). No sharing a database with the Order Service.
- **Goal:** Implement the REST API (GET /api/products) and the persistence layer within a single, isolated application.

---

3. Knowledge Base: Microservice Isolation

- **Isolation in Practice:** A Microservice is a process that can be developed and run without needing any other service online (except for external dependencies like a message broker or its own database).
- **Technology Choice:** We use **Flask** for the API and **Flask-SQLAlchemy** with **SQLite** for the dedicated, internal database.
- **Simulating Production:** SQLite acts as a dedicated database instance, emphasizing that no other service will connect to this file.

---

4. Activity 1: Project Setup and Installation

- **Goal:** Create the necessary directory and install the ORM (Object-Relational Mapper).
- **Installation Steps:**
    1. mkdir product_service

2. cd product_service
3. python -m venv venv
4. source venv/bin/activate
5. pip install Flask Flask-SQLAlchemy
6. touch app.py

---

5. Knowledge Base: ORM and Persistence

- **ORM (Object-Relational Mapper):** A tool (like SQLAlchemy) that allows developers to interact with a database using object-oriented code (e.g., Python classes) instead of raw SQL.
- **Benefit:** Increases development speed and maintains the **separation of concerns** within the service.

---

6. Code Step 1: Initialize Flask and Database

- **Goal:** Set up the basic application structure and configure the SQLite file.
- **File: app.py (Initial Setup)**

Python

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

app = Flask(__name__)
# 1. Dedicated Database Configuration
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

---

7. Code Step 2: Define the Product Model (Schema)

- **Goal:** Map the Python class structure to a permanent database table.
- **Implementation:** Define the attributes (id, name, price, stock) as database columns.
- **File: app.py (Model Definition)**

Python

```
class Product(db.Model):
    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):
        # Helper for API response formatting
        return {
            'id': self.id,
            'name': self.name,
            'price': self.price,
            'stock': self.stock,
            'is_active': self.is_active
        }
```

---

8. Activity 2: Database Initialization and Seeding

- **Goal:** Create the physical database file (products.db) and inject initial test data.
- **Action Steps (Interactive Python Shell):**
    1. python

2. >>> from app import app, db, Product
3. >>> with app.app_context():
4. ... db.create_all() # Creates the table
5. ... db.session.add(Product(name='Laptop Z1', price=1500.00, stock=10))
6. ... db.session.add(Product(name='Mouse Pro', price=50.00, stock=50))
7. ... db.session.commit()

---

9. Knowledge Base: Service Contract Fulfillment (Read)

- **Contract:** The Product Service must fulfill two read-only endpoints:
    1. List/Search all active products.
    2. Retrieve a single product by ID.
- **Implementation:** Use SQLAlchemy's methods (query.filter_by, query.get) to translate API requests into database queries.

---

10. Code Step 3: List and Search Endpoint (GET /api/products)

- **Goal:** Implement the primary catalog view, including an optional search parameter.
- **File: app.py (List/Search Route)**

Python

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

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

    if query:
```

```
        # Filtering logic for search
        products = products.filter(Product.name.like(f'%{query}%'))

    return jsonify([p.to_dict() for p in products.all()]), 200
```

---

11. Code Step 4: Detail Endpoint (GET /api/products/int:product_id)

- **Goal:** Retrieve a specific product, handling the **404 Not Found** error if the ID is invalid.
- **File: app.py (Detail Route)**

Python

```
@app.route('/api/products/<int:product_id>', methods=['GET'])
def get_product_details(product_id):
    # Retrieve by Primary Key
    product = Product.query.get(product_id)

    if product and product.is_active:
        return jsonify(product.to_dict()), 200
    else:
        # Handle Not Found (Standard REST API Error)
        return jsonify({'message': 'Product not found or is inactive'}), 404
```

---

12. Code Step 5: Service Execution

- **Goal:** Define the entry point and ensure the service runs on a **dedicated port**.
- **Reasoning:** To prevent port collisions with the API Gateway (Lecture 6) or other services.
- **File: app.py (End of file)**

Python

```
if __name__ == '__main__':
```

```
# Running on port 5001, separate from the Gateway (5000)
app.run(port=5001, debug=True)
```

---

13. Activity 3: Proof of Concept - Isolation Testing

- **Goal:** Verify the service runs and responds correctly in complete isolation.
- **Action Steps:**
    1. Ensure the database has been seeded.
    2. Run the application: python app.py

---

14. Verification (Hands-On): Test Listing (R)

- **Tool:** cURL or Postman.
- **Test 1: List All Products:**
    - **Method:** GET
    - **URL:** http://127.0.0.1:5001/api/products
    - **Expected:** HTTP **200 OK** containing the two seeded products (Laptop Z1 and Mouse Pro).

---

15. Verification (Hands-On): Test Details (R)

- **Test 2: Retrieve Product 1:**
    - **Method:** GET
    - **URL:** http://127.0.0.1:5001/api/products/1
    - **Expected:** HTTP **200 OK** with the JSON details of "Laptop Z1".

---

16. Verification (Hands-On): Test Error Handling

- **Test 3: Non-existent ID:**
    - **Method:** GET

- **URL:** http://127.0.0.1:5001/api/products/999
- **Expected:** HTTP **404 Not Found** with the defined error message.

---

17. Architectural Significance: Data Ownership

- **Achieved:** The service successfully created and managed its own products.db file.
- **Key Insight:** If the Order Service needed product information, it **must not** access this file directly. It must communicate through the stable REST API (GET /api/products/{id}). This prevents coupling.

---

18. Architectural Review: Scaling Readiness

- **Scaling:** Because the service is isolated and stateless (the database access is the only state, which would be external in production), we can now:
    1. Deploy a second, third, or fourth instance of this service (on different ports/servers).
    2. Place all instances behind a Load Balancer.
    3. Meet the **Scalability ASR** without affecting any other service.

---

19. Summary of Deliverables for Submission

- **Document 1:** Code listing of the **Product Model** showing the SQLAlchemy column definitions.
- **Document 2:** Code listing of the GET /api/products route showing the query filtering (query.filter_by).
- **Document 3:** Output logs/screenshots showing the service running on **port 5001** and the successful **404 Not Found** test result.

---

20. Q&A and Next Steps

- **Questions?** (Review ORM mapping vs. raw SQL, the importance of the dedicated database).

- **Pre-work:** Research the **API Gateway Pattern**—its role in routing and security.
- **Next Lecture: Lecture 6: API Gateway Pattern & Security**. We will implement the crucial front-door component.