Software Architecture: Lecture 5

Implementing an Independent Microservice

This lecture focuses on translating theoretical microservice contracts into runnable, standalone components. We will dive deep into implementing a **Product Service**, showcasing how to build an isolated, scalable service that adheres to core microservice principles.

Recap: The Microservice Contract

Before diving into implementation, let's briefly recap the critical aspects of our microservice contract, specifically for the Product Service within our ShopSphere ecosystem. These principles guide our architectural decisions and ensure a robust, scalable system.

ASR Driver: Scalability

The Product Service must be capable of independent deployment and scaling. This is a non-negotiable architectural significant requirement (ASR) to handle fluctuating loads without impacting other parts of ShopSphere. Imagine a flash sale: only the Product Service needs to scale up, not the entire application.

Key Principle: Data Ownership

A fundamental microservice tenet. The Product Service **owns** its data (product details). This means no shared databases with other services like the Order Service. Each service manages its persistence layer to prevent tight coupling and ensure autonomy.

Implementation Goal

Our objective is to implement the specified REST API (specifically `GET /api/products` and `GET /api/products/{id}`) and its persistence layer within a single, isolated application. This ensures the service can function as a truly independent component.

Knowledge Base: Microservice Isolation

Microservice isolation is not just a concept; it's a practical approach to building resilient and maintainable systems. Here's what it means in our context.

Isolation in Practice

A microservice should operate as an independent process. This means it can be developed, deployed, and run without requiring other services to be online, save for essential external dependencies like its dedicated database or a message broker for asynchronous communication. This autonomy is crucial for continuous delivery and operational flexibility.

Technology Choice

For our Product Service, we've selected:

- **Flask**: A lightweight Python web framework perfect for building small, focused APIs.
- **Flask-SQLAlchemy**: An extension that simplifies database interactions in Flask applications.
- **SQLite**: A file-based database used here to simulate a dedicated database instance.

Activity 1: Project Setup and Installation

Let's get our hands dirty and set up the development environment for our Product Service. This involves creating the project directory, setting up a virtual environment, and installing the necessary libraries.

Create Project Directory

```
mkdir product_service
```

This command creates the root directory for our new microservice. Keep your microservices in separate directories to maintain clear boundaries.

Navigate into Directory

```
cd product_service
```

Change your current working directory to the newly created project folder. All subsequent commands will be executed from here.

Create Virtual Environment

```
python -m venv venv
```

A virtual environment isolates our project dependencies, preventing conflicts with other Python projects on your system. It's a best practice for Python development.

Activate Virtual Environment

```
source venv/bin/activate
```

This command activates the virtual environment. Your terminal prompt should change to indicate that the virtual environment is active.

Install Dependencies

```
pip install Flask Flask-SQLAlchemy
```

Install Flask (our web framework) and Flask-SQLAlchemy (our ORM extension) into the activated virtual environment.

Create Application File

```
touch app.py
```

Create an empty Python file named `app.py`. This file will contain all the code for our Product Service.

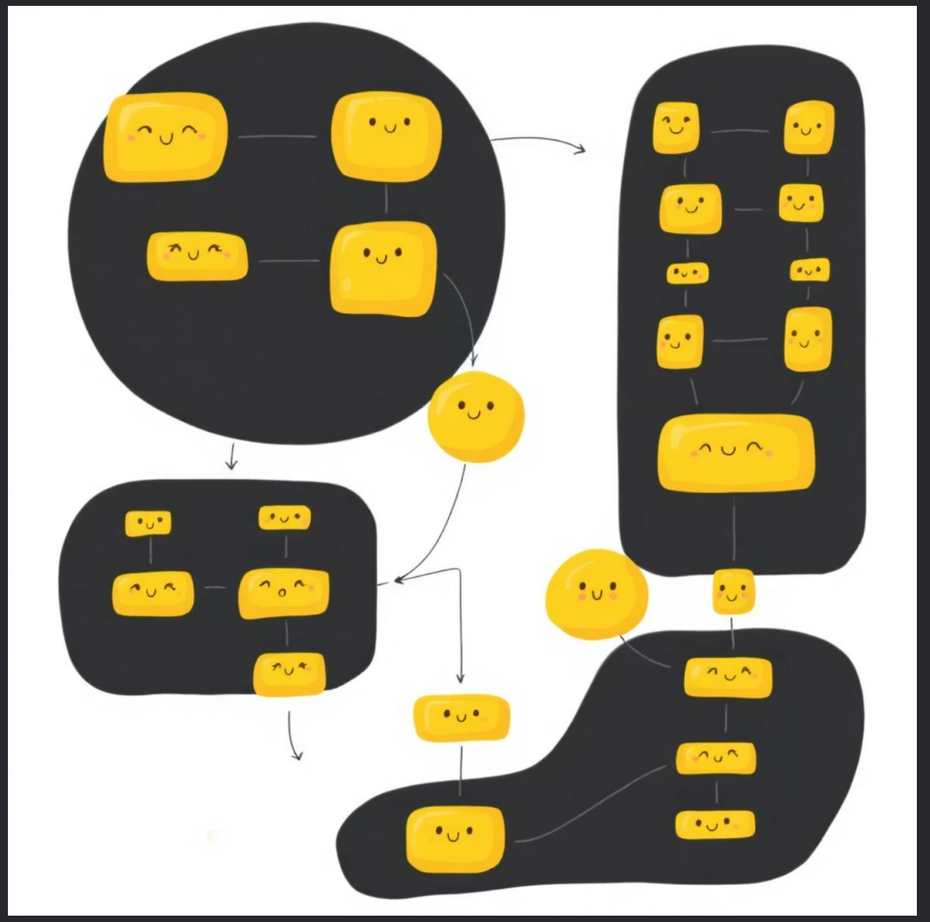
Knowledge Base: ORM and Persistence

Understanding how Object-Relational Mappers (ORMs) work is crucial for efficient and maintainable microservice development, especially when dealing with data persistence.

What is an ORM?

An ORM, or Object-Relational Mapper, is a programming technique that allows you to interact with a relational database using an object-oriented paradigm. Instead of writing raw SQL queries, you define Python classes that map directly to database tables. Instances of these classes represent rows in the table, and their attributes correspond to table columns.

Flask-SQLAlchemy is a popular ORM for Flask applications that integrates SQLAlchemy (a powerful Python SQL toolkit and ORM) with Flask, making database operations seamless.



The Benefits of Using an ORM

- **Increased Development Speed:** ORMs reduce the amount of boilerplate code needed for database interactions, allowing developers to focus on business logic rather than SQL syntax.
- **Abstraction and Maintainability:** They provide a layer of abstraction over the database, making the code more database-agnostic and easier to maintain. You can switch databases with minimal code changes.
- **Separation of Concerns:** ORMs help maintain a clear separation between your application's business logic and its data access layer.
- **Reduced SQL Injection Risks:** By parameterizing queries, ORMs inherently offer better protection against SQL injection attacks compared to manually concatenating strings for SQL queries.

While ORMs offer many advantages, it's important to understand the underlying SQL. For complex queries or performance-critical sections, sometimes raw SQL can be more efficient, and most ORMs provide ways to execute it directly.

Code Step 1: Initialize Flask and Database

We start by setting up the foundational components of our Product Service: the Flask application and its connection to our dedicated database. Open your `app.py` file and add the following code.

File: app.py (Initial Setup)

```
from flask import Flask, jsonify, requestfrom flask_sqlalchemy import SQLAlchemyapp = Flask(__name__)# 1. Dedicated Database Configurationapp.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = Falsedb = SQLAlchemy(app)
```

Imports

We import `Flask` to create our web application, `jsonify` for sending JSON responses, `request` to handle incoming requests, and `SQLAlchemy` from `flask_sqlalchemy` to manage our database.

Flask App Initialization

`app = Flask(__name__)` creates our Flask application instance. The `__name__` argument tells Flask where to look for static files and templates.

Database Configuration

`app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'` configures Flask-SQLAlchemy to use an SQLite database named `products.db` in the same directory as our `app.py` file. The `SQLALCHEMY_TRACK_MODIFICATIONS` setting is disabled to save memory, as it's not needed for our current use case.

DB Instance

`db = SQLAlchemy(app)` initializes Flask-SQLAlchemy with our Flask application, making the `db` object available for defining models and interacting with the database.

Code Step 2: Define the Product Model (Schema)

Now we define how our product data will be structured in the database. This Python class, `Product`, will serve as our model, mapping directly to a database table.

File: app.py (Model Definition)

```
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        }
```

The `Product` class inherits from `db.Model`, making it a SQLAlchemy model. Each attribute (`id`, `name`, `price`, etc.) is defined using `db.Column`, specifying its data type and constraints (e.g., `primary_key`, `nullable`, `default`). The `to_dict` method is a convenience helper to easily serialize product objects into a dictionary format suitable for JSON API responses.

Activity 2: Database Initialization and Seeding

With our database configured and product model defined, it's time to create the physical database file and populate it with some initial data. This step is crucial for testing our API endpoints later.

1

1. Enter Python Shell

```
python
```

Open your terminal and type `python` to enter the interactive Python interpreter.

2

2. Import Essentials

```
>>> from app import app, db, Product
```

Import the Flask app instance, the database object, and our Product model from `app.py`.

3

3. Create Tables

```
>>> with app.app_context():...     db.create_all()
```

Within the application context, `db.create_all()` will inspect all defined models and create corresponding tables in the `products.db` file. If the file doesn't exist, it will be created.

4

4. Add Initial Data

```
...     db.session.add(Product(name='Laptop Z1', price=1500.00, stock=10))...     db.session.add(Product(name='Mouse Pro', price=50.00, stock=50))
```

We create two new `Product` objects and add them to the database session. These are temporary in memory until committed.

5

5. Commit Changes

```
...     db.session.commit()
```

This command persists the new product records from the session into the `products.db` file. You can then exit the Python shell using `exit()`.

After completing these steps, you should see a new file named `products.db` in your `product_service` directory. This database now contains our initial product catalog.

Knowledge Base: Service Contract Fulfillment (Read Operations)

The core responsibility of our Product Service, as defined by its contract, is to provide product information. This primarily involves read-only operations.

The Contract: Read-Only Endpoints

The Product Service must expose two critical read-only endpoints to fulfill its service contract:

1. **List/Search Active Products:** This endpoint allows consumers to retrieve a collection of all active products, with the option to filter or search the list based on specific criteria.
2. **Retrieve Single Product by ID:** This endpoint provides detailed information for a specific product, identified by its unique ID.

These endpoints adhere to RESTful principles, providing a clear and standardized way for other services (or front-end applications) to interact with our product data.

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

This endpoint allows clients to retrieve a list of active products, with an optional search capability by product name. Add this code to your `app.py` file.

File: app.py (List/Search Route)

```
@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
```

Code Step 4: Detail Endpoint (GET /api/products/<int:product_id>)

This endpoint is responsible for retrieving the details of a single product by its ID, and gracefully handling cases where the product is not found or is inactive. Add this to `app.py`.

File: app.py (Detail Route)

```
@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 Handle Not Found (Standard REST API Error)        return jsonify({'message': 'Product not found or is inactive'}), 404
```

Code Step 5: Service Execution

Finally, we define the entry point for our Flask application and specify the port it will run on. It's crucial to use a dedicated port to avoid conflicts with other services. Add this at the very end of `app.py`.

File: app.py (End of file)