Design and Implement a Simple Microservices-based Application

Software Engineering – Department of Advanced Computing Sciences
tom.pepels@maastrichtuniversity.nl

Objective:

In this assignment, you will design and implement a simple microservices-based application to understand the fundamentals of microservices architecture. This will help you gain hands-on experience in designing, implementing, and integrating different microservices.

Scenario:

you are working on a project for a company that wants to create a simple book management system. The company's goal is to have an application that can perform basic operations on a catalog of books, manage the inventory of these books, and handle book orders. They would like this application to be built using a microservices architecture.

The application should include the following requirements:

- **Book Catalog:** Users should be able to perform CRUD (Create, Read, Update, and Delete) operations on books. Each book should have a unique identifier, title, author, and publication year.
- **Book Inventory:** Users should be able to manage the inventory of books, including tracking the available quantity of each book. When new books are added to the catalog, their initial quantity should be set. Users should also be able to update the quantity of a book.
- **Book Orders:** Users should be able to place orders for books. Each order should include the book's unique identifier, the quantity ordered, and the date of the order. When an order is placed, the inventory should be updated accordingly to reflect the new available quantity of the ordered book.

After completing the design and implementation of the microservices-based application as described in the previous steps, you should have a functional book management system that allows users to manage a catalog of books, their inventory, and orders. The application's architecture should exhibit the key characteristics of microservices, such as modularity, scalability, and ease of maintenance.

Instructions:

1. Identify the microservices: Break down the problem into smaller, independent services that can be developed and deployed individually. For this assignment, identify at least three microservices.
2. Define APIs: For each microservice, define a simple RESTful API with the required endpoints. Use JSON as the data interchange format. Write an API specification using a tool like swagger.io or slate.
3. Implement microservices: Implement each microservice using a programming language and framework of your choice. For this assignment, you can use mock data in your models instead of integrating with a database.

4. Integration: Use the API Gateway published on GitHub (GitHub - book_gateway.py) to route requests to the appropriate microservices. This gateway should also act as a single entry point for the application. Read and understand the Gateway and make sure your microservices are compatible with the Gateway.
5. Testing: Test each microservice individually and the overall application to ensure proper functionality.

Guidance:

1. Consider the following microservices for this assignment: a. Book Catalog Service: Responsible for managing book information. b. Book Inventory Service: Responsible for managing the quantity of each book. c. Book Order Service: Responsible for handling book orders and updating the inventory.
2. For each microservice, design the required endpoints. For example, the Book Catalog Service might have the following endpoints: a. POST /books: To add a book. b. GET /books/{id}: To retrieve book information by ID. c. DELETE /books/{id}: To delete a book by ID.
3. Use a lightweight framework like Flask (Python), Express (Node.js), or Spring Boot (Java) to implement the microservices. (**hint**: there are tools that can generate code given endpoint definitions written in openapi format)
4. Use tools like Postman or curl to test your APIs. Write unit tests for each microservice using appropriate testing libraries for your chosen programming language.

Submission:

Submit your source code, along with files containing:

- The API documentation for each microservice.
- The unittests that test your project

Grading Criteria:

Your assignment will be graded based on the following criteria:

- Proper identification and separation of microservices.
- Clear and concise API design.
- Correct implementation of microservices and API Gateway.
- Comprehensive testing of individual microservices and the overall application.

Setting up Unit Testing for Java and Python:

Unit testing is a crucial aspect of software development that allows you to verify the correctness and functionality of individual units or components of your application. In this section, we will provide a brief guide on setting up unit testing for Java and Python projects. Please note that, depending on the editor you are using, some steps related to running the unittests may be easier within your IDE. For instance, in Visual Studio Code, you can click on

the depicted icon to access built-in unittesting functionality with which you can skip many of the steps outlined below.

Java- JUnit:

JUnit is a widely-used testing framework for Java applications. To set up JUnit for your project, follow these steps:

1. Add JUnit as a dependency to your project. If you are using Maven, add the following dependency to your **pom.xml** file:

   ```
   <dependencies>
     ...
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.13.2</version>
       <scope>test</scope>
     </dependency>
     ...
   </dependencies>
   ```

   For Gradle, add the following dependency to your **build.gradle** file:

   ```
   dependencies { testImplementation 'junit:junit:4.13.2' }
   ```

2. Create a test class for the class you want to test. The test class should be located in the **src/test/java** directory, following the same package structure as the main source code. Test classes are usually named after the class they test, with a "Test" suffix (e.g., **BookCatalogServiceTest** for **BookCatalogService**).
3. Write test methods using JUnit annotations such as **@Test** for test cases, **@Before** for setup methods, and **@After** for teardown methods. Test methods should be designed to cover different scenarios and edge cases for the class being tested.
4. Run the tests using your preferred IDE or build tool. Both Maven and Gradle have built-in support for running JUnit tests (**mvn test** or **./gradlew test**).

Python- unittest:

The **unittest** module is a built-in testing framework for Python applications. To set up **unittest** for your project, follow these steps:

1. Create a test directory at the root level of your project. Within this directory, create a test file for each class or module you want to test. Test files are usually named after

the module they test, with a "_test" suffix (e.g., **book_catalog_service_test.py** for **book_catalog_service.py**).

2. Import the **unittest** module and the class or module you want to test in the test file. Create a test class that inherits from **unittest.TestCase**.
3. Write test methods for your test class. Test methods should start with the word "test" and be designed to cover different scenarios and edge cases for the class or module being tested. Use **unittest**'s assertion methods (e.g., **assertEqual**, **assertTrue**, **assertFalse**) to verify the expected behavior.
4. To run the tests, either use your preferred IDE's built-in test runner, or execute the tests from the command line using the following command:

```
python -m unittest discover
```

This command will discover and run all test files within your project directory that follow the "test*.py" naming convention.