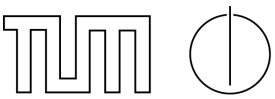
Exercises for Advanced Topics of Software Engineering
Winter semester 2021/22
Prof. Dr. Florian Matthes, Tri Huynh, Burak Öz



Technische Universität München
Fakultät für Informatik

Exercise Sheet 11: Service-oriented Architectures and Microservices

Welcome to the eleventh week of Advanced Topics of Software Engineering. In this week's theoretical exercise, we will practice service-oriented architectures with a focus on microservices. We are going to discuss patterns in microservice architectures, namely API Gateway, service discovery, container-per-service, database-per-service, and shared database.

Theoretical Exercise

Exercise 1: Patterns in Microservices

In the lecture you were introduced to common problems in service-oriented and microservices architectures such as communication between services or their deployment. Over the past two decades, patterns have evolved which tackle these problems in a language-agnostic way. The following exercises cover four of the most important patterns, which you will get to know in the following by *self-studying* their respective technical description.

1. **API Gateway**

   The API Gateway is one of the communication patterns around microservices. Read the following article and answer the questions below: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/

   - Which problem is solved by the API Gateway pattern?
   - What are the benefits of using an API Gateway?
   - What are potential drawbacks of the API Gateway pattern?

2. **Service Discovery**

   The Service Discovery pattern has been discussed in the lecture already, but to get a clearer understanding of the concept, read this article: https://microservices.io/patterns/service-registry.html

   - Which problem is solved by the Service Discovery pattern?
   - What are the entities involved in the Service Discovery pattern and which responsibilities do they have?
   - What are the differences between client- and server-side service discovery?
   - How can one integrate **load balancing** into the Service Discovery pattern?[1]

3. **Service Instance per Container**

   The Service Instance per Container pattern is a deployment pattern for microservices, which is described in more detail here: https://microservices.io/patterns/deployment/service-per-container.html

   Read through the documentation and answer the following questions.

   - How does the service instance per container pattern facilitate the deployment of microservices?

[1] Load Balancing 101: Nuts and Bolts https://www.f5.com/pdf/white-papers/load-balancing101-wp.pdf

- Which implications does this have on the supplied hardware resources for a service?

4. **Database per Service & Shared Database**

   The Database per Service pattern is a data management pattern to be used with the microservice architecture. Shared Database is another pattern for data management that can be utilized when using microservices. Read the following articles about these two patterns and answer the questions:

   - Database per Service: https://microservices.io/patterns/data/database-per-service.html
   - Shared Database: https://microservices.io/patterns/data/shared-database.html

   (a) Which problem is solved with the Database per Service pattern?

   (b) What are the possible ways to apply the Database per Service pattern when using a relational database?

   (c) What are the drawbacks of the Database per Service pattern? Can they be resolved with the Shared Database pattern?

   (d) Is Shared Database pattern an **anti-pattern**? Why?

Practical Exercise

Even though the microservices patterns are conceptually language-agnostic, you still need to implement them in some programming language for your specific domain. This week's practical exercises will teach you how to concretely implement two patterns of microservices architectures, namely Service Discovery and API Gateway.

As discussed in the theoretical exercise, these patterns serve the following purposes:

- **Service Discovery:** Discovering locations of microservice instances in a network of services
- **API Gateway:** Single entry point (edge service) for clients requesting microservices

The following practical exercises give examples of how to implement these patterns in the Spring environment using Spring Cloud Gateway[2] and The Spring Cloud Netflix project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and other Spring programming model idioms.[3]. You will then transfer the gained insights and knowledge about those patterns to your ASE Delivery and finally hook up your separate microservices to one connected structure.

Note: Microservices patterns solve technical problems (as described in the theoretical exercise) for specific software architectures regardless of a specific (business) domain. As a consequence, the following practical exercises do not focus on business logic or context, but rather on understanding the technical concepts of the patterns.

Exercise 2: Service Discovery Pattern

The Service Discovery pattern is one of the key tenets of a microservice architecture and is used to discover locations of dynamically changing numbers of microservice instances. Knowledge of the locations of microservice instances is **crucial for communication between services** and should be obtained with **minimum networking overhead**. In order to keep track of the available service instances, a service registry contains a list of all registered instances and their locations.

When implementing service discovery, we are concerned with (i) configuring the service registry server, (ii) setting up a standardized (de-)registering process for service instances, and (iii) returning registered instances from the registry to service clients who request specific services.

[2] Spring Cloud Gateway: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/
[3] Spring Cloud Netflix Documentation: https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/

Netflix Eureka[4] is a REST-based service discovery library developed and used by Netflix at least since 2008. If we were to implement classical (XML-based) *web services* and describe them with the Web Services Description Language (WSDL), we could instead use UDDI[5] for service discovery, which was introduced in the lecture.

The two core components of Eureka are the **Eureka Server** and the **Eureka Client**. They simplify interactions with registered services. The client also has a built-in load balancer that does basic round-robin load balancing, i.e. assigning requests in turn for more equitably request load.

As it can be difficult to hand-configure Eureka, the aforementioned Spring Cloud Netflix project provides autoconfiguration and capabilities for using Eureka in Spring Boot applications.[6]

In the following, we will first create a Eureka server and then register two services, with multiple instances as Eureka clients. These two services are the *Project* service from exercise sheet 9 and a new *Regulation* service which contains the regulations that a project has to conform. The *Project* service will make a call to the *Regulation* one to fetch the required regulations. Each *Project* contains a list of *Regulation* IDs, and will invoke an RPC call to the *Regulation* service via `RestTemplate` with hard-coded URLs. This is a typical example for communication between microservices.

To simply demonstrate this communication, we will return a hard-coded *Regulation* in the *Regulation* service instead of creating a full-fledged service.

The described communication behavior can be improved by integrating the Service Discovery pattern.

1. **Creating the Eureka server (service registry)**

   (a) Use the Spring Initializr (https://start.spring.io/) to generate and download a new Spring Boot Project with the following options:

   - Project: Maven
   - Language: Java
   - Spring Boot: 2.2.2 (or the latest)
   - Group: edu.tum.ase
   - Artifact: discovery-server
   - Packaging: jar
   - Java: 8 (or higher)
   - Dependencies: Eureka Server

   and import the project into your IDE as a Maven project

   (b) The Eureka server configuration is straightforward, you simply need to add the annotation `@EnableEurekaServer` to your Spring Boot application class.

   Then, you can provide a standard configuration for the Eureka server in the `application.yml` which sets the port the server listens for service registrations to 8761 and prevents the server from registering with itself:

```yaml
spring:
  application:
    name: discovery-service
server:
  port: ${PORT:8761}
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

2. **Creating the Regulation as a Eureka client (service instance)**

   (a) Use the Spring Initializr (https://start.spring.io/) to generate and download two new Spring Boot Projects with the following options:

   - Project: Maven
   - Language: Java

[4] Eureka at a Glance: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance
[5] Web Services Discovery (UDDI): https://en.wikipedia.org/wiki/Web_Services_Discovery
[6] Spring Cloud Eureka: https://cloud.spring.io/spring-cloud-netflix/reference/html/#service-discovery-eureka-clients

- Spring Boot: 2.2.2 (or the latest)
- Group: edu.tum.ase
- Artifact: regulation-service
- Packaging: jar
- Java: 8 (or higher)
- Dependencies: Spring Web, Eureka Discovery Client

and import the projects into your IDE as Maven projects

(b) To configure a Spring Boot application as Eureka, add the annotation `@EnableEurekaClient` to your Spring Boot application class.
Then, you can provide a standard configuration for the Eureka client in the `application.yml`:

```yaml
spring:
  application:
    name: regulation-service
server:
  port: ${PORT:9091}
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_SERVER:http://localhost:8761/eureka}
```

Define a similar Eureka Client configuration in the `application.yml` for the *Project* service.

```yaml
spring:
  application:
    name: project-service
server:
  port: ${PORT:8080}
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_SERVER:http://localhost:8761/eureka}
```

3. To simplify the process of creating a full-fledged `Regulation` service, we will offer an endpoint that returns two regulations directly in the main class of the Spring Boot Application, and a `Regulation` model:

```java
@SpringBootApplication
@EnableEurekaClient
@RestController
public class RegulationServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(RegulationServiceApplication.class, args);
    }

    @GetMapping("/regulation")
    public List<Regulation> getRegulation() {
        Regulation gdpr = new Regulation("GDPR", "https://gdpr-info.eu/" );
        Regulation jsg = new Regulation("Java Style Guide", "https://google.github.io/
            styleguide/javaguide.html" );

        return Arrays.asList(gdpr, jsg);
    }
}

// --------------------

public class Regulation {

    String name;
    String url;

    public Regulation(){}

    public Regulation(String name, String url) {
        this.name = name;
        this.url = url;
    }

    // Getter & Setters
}
```

4. Run all three projects (Eureka server and the two Eureka clients, project-service and regulation-service). Open a new browser window at `localhost:8761`, where you should see an overview of all registered service instances (two at the moment).

   Spring Cloud's wrapper around Eureka does all the magic necessary to register the services, get their health status and show this information in a simple dashboard. With only a few lines of configuration, we were able to register Eureka clients with the Eureka service registry. For more details on the exact implementation of Eureka within Spring, read through the documentation at https://cloud.spring.io/spring-cloud-netflix/reference/html/#service-discovery-eureka-clients.

5. **Now, you have three tasks to complete:**

   - Open another browser window at `localhost:9091`, where you should see a regulation list with two regulations, which the project-service retrieved from the regulation-service. At the moment the project-service uses a **hard-coded URL** to access the regulation-service for fetching the list of regulations. Replace it by making use of the service name registered at the Eureka server.
   - Simulate the horizontal scaling of a service by running and registering more instances of it. You can check the amount of registered service instances at the Eureka dashboard (`localhost:8761`).
   - Find a way to log the addresses of all available service instances before sending a request from the frontend-service to the backend-service.

     Note that having this list of available service instances could be a starting point for implementing custom load balancing methods.

Exercise 3: API Gateway

Routing is an integral part of a microservice architecture. The API Gateway pattern facilitates routing by defining a single entry point to a network of microservices, the API gateway, that is responsible to route incoming requests to appropriate service instances. A service client (i.e. requester of a service) can be either a client outside the network of microservices or another service within the network.

Spring Cloud Gateway[7] is an API Gateway built on top of Spring Boot, Spring WebFlux and Project Reactor. Spring Cloud gateway aims to provide a simple yet effective mechanism for APIs routing and provide security, monitoring/metrics and resiliency as its cross-cutting concerns.

To demonstrate how to use Spring Cloud Gateway to create an API Gateway, you will extend the service discovery example from the previous exercise. Ultimately, this exercise will give you a complete example for server-side service discovery integrated with a Spring Cloud Gateway, the single entry point and router for requests.

Let us start by creating the API Gateway service as a new Spring Boot Project. Use the Spring Initializr (https://start.spring.io/) to generate and download two new Spring Boot Projects with the following options:

- Project: Maven
- Language: Java
- Spring Boot: 2.2.2 (or the latest)
- Group: edu.tum.ase
- Artifact: api-gateway
- Packaging: jar
- Java: 8 (or higher)
- Dependencies: Gateway, Eureka Discovery Client

First, add `@EnableEurekaClient` into the main `SpringBootApplication` class to let the API Gateway search for *Project* and *Regulation* Services. Next, we will configure the `application.yml` such that it routes project and regulation requests to the corresponding `Project` and `Regulation` services.

[7] Spring Cloud Gateway: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/

```yaml
spring:
  application:
    name: api-gateway

  cloud:
    gateway:
      routes:
      - id: regulation-service
        uri: lb://REGULATION-SERVICE
        predicates:
        - Path=/regulation

      # TODO: Configure routes to access project services

eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://${eureka.instance.hostname}:8761/eureka

server:
  port: ${PORT:10789}
```

Note: A wrong tab in `application.yml` file can cause minutes to hours of debugging. So check the tab carefully during your configuration

At this point, if you configure the API Gateway correctly, you can see the list of regulations by sending a GET request to `http://localhost:10789/regulation`. The *uri* `lb://REGULATION-SERVICE` contains the `lb` scheme, which denotes that the Spring Cloud `ReactorLoadBalancer` is used to replace the service name (REGULATION-SERVICE in this case) by its actual host and port. The *Path* predicate[8] indicates the route in the request path that will be mapped to the service defined in the *uri*. In the above configuration of the regulation service, if a client requests a `/regulation` call to the API Gateway, the gateway maps `/regulation` as the endpoint of the REGULATION-SERVICE running in localhost at port 9091, hence the gateway makes a call to `http://localhost:9091/regulation`.

Your task is to configure the project service, such that you can call the authentication and other project services as well.

When you call the API Gateway from React, a number of challenges can arise comparing to request via Postman, like configuring the CORS[9] policy of the gateway. Furthermore, you will also need to remove duplicated headers from the response of the API gateway. The duplication can be caused by an overlapping configuration of the Gateway API and services, hence the `-DedupeResponseHeader`[10] can be helpful for this purpose.

The exercise of Week 10 finishes here. In the last exercise, we will see how to orchestrate and deploy our system in Docker.

[8] Spring Cloud Gateway Predicate: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#gateway-request-predicates-factories

[9] Configure CORS in Spring Cloud Gateway: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#cors-configuration

[10] DedupeResponseHeader: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-deduperesponseheader-gatewayfilter-factory