Prof. Bernardo Copstein

Prof. Júlio Machado

Roteiro 3: Gateway

Introdução

No roteiro 2 vimos como acrescentar um "name server" para simplificar a localização dos serviços e o balanceamento de carga. Para assumir esse papel usamos o Eureka Server da Netflix. A figura 1 apresenta a arquitetura do nosso sistema até o momento.

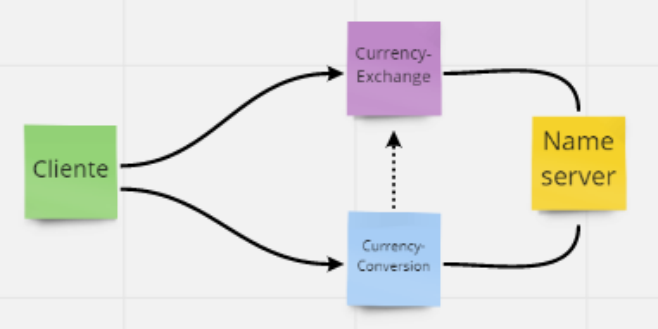

Figura 1 – Sistema desenvolvido no roteiro 2

Um "name server" é um servidor capaz de traduzir "nomes" em endereços "ip". No nosso caso será capaz de traduzir os nomes dos serviços registrados nos respectivos endereços e, quando houver mais de uma cópia do mesmo serviço, prover o balanceamento de carga entre elas.

Ocorre que os clientes que acessam nossa aplicação provavelmente não estarão localizados no mesmo "host". Eventualmente nem todos os micros serviços estarão localizados no mesmo "host". Então para lidar com o fato de que os "componentes" da nossa arquitetura podem estar espalhados em diferentes "hosts", iremos necessitar de um "gateway". Desta forma as aplicações cliente irão concentrar todas as suas requisições no "gateway" que, juntamente com o "name server", saberá localizar os micros serviços. Além disso, termos um ponto de entrada único o que facilita a implementação de serviços de autenticação, entre outros. A figura 2 mostra como ficará nossa arquitetura a partir dessa etapa.

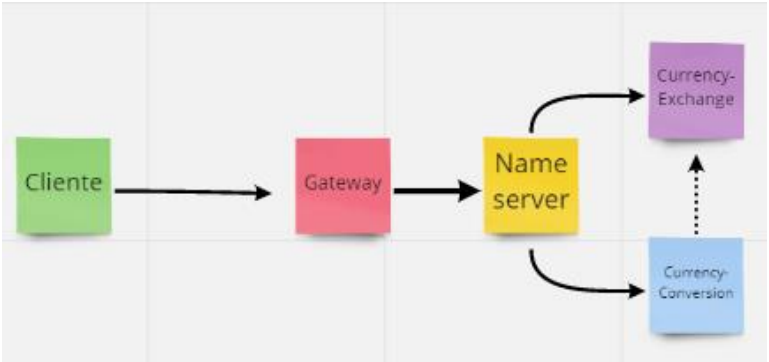

Figura 2 – arquitetura do sistema incluindo o "gateway"

A construção dessa arquitetura irá ocorrer em 3 passos. Antes de seguir estes passos certifique de ter subido, na sequência, o “name server”, o serviço de câmbio e o serviço de conversão configurados no roteiro 2.

Passo 1: criando o “gateway”

A tecnologia de implementação será o Java através do *framework* Spring. Adicionalmente o sistema de *build* será realizado através do Maven e, portanto, o primeiro passo será a configuração das dependências.

Para facilitar o processo, será utilizado o sistema “Spring Initializr” via web no endereço https://start.spring.io/. A figura 3 traz as configurações necessárias, já a figura 4 apresenta o arquivo de dependências correspondente. Clique no botão “GENERATE” e faça o download do projeto em um arquivo “zip” para um diretório local.

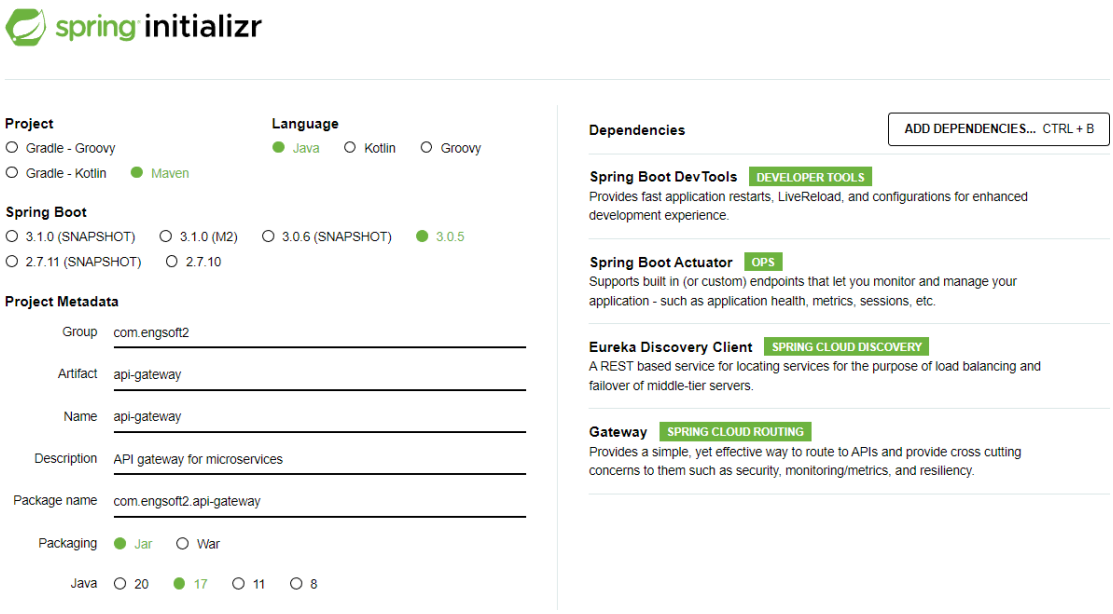

Figura 3 – configuração do Spring Initializr

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>3.0.5</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.engsoft2</groupId>
	<artifactId>api-gateway</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>api-gateway</name>
	<description>API gateway for microservices</description>
	<properties>
		<java.version>17</java.version>
		<spring-cloud.version>2022.0.2</spring-cloud.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-actuator</artifactId>
```

```
		</dependency>
		<dependency>
			<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-starter-gateway</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-devtools</artifactId>
			<scope>runtime</scope>
			<optional>true</optional>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>
	<dependencyManagement>
		<dependencies>
			<dependency>
				<groupId>org.springframework.cloud</groupId>
				<artifactId>spring-cloud-dependencies</artifactId>
				<version>${spring-cloud.version}</version>
				<type>pom</type>
				<scope>import</scope>
			</dependency>
		</dependencies>
	</dependencyManagement>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
	<repositories>
		<repository>
			<id>netflix-candidates</id>
			<name>Netflix Candidates</name>
			<url>https://artifactory-oss.prod.netflix.net/artifactory/maven-
oss-candidates</url>
			<snapshots>
				<enabled>false</enabled>
			</snapshots>
		</repository>
	</repositories>

</project>
```

Figura 4 - arquivo POM do micro serviço de gateway.

```
@EnableEurekaServer
@SpringBootApplication
public class NamingServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(NamingServerApplication.class, args);
    }
}
```

Figura - Código do serviço de gateway

Além do código propriamente dito, necessitamos configurar algumas propriedades no arquivo “application.properties” no diretório “resources” como pode ser visto na figura 5. As duas primeiras propriedades indicam o nome do serviço e a porta onde ele está “escutando”. A propriedade seguinte indica onde está o “name server”. Finalmente as duas últimas

propriedades ativam o serviço de descoberta de nomes integrado com o “Eureka” admitindo que os nomes dos serviços registrados sejam informados com minúsculas.

```
spring.application.name=api-gateway
server.port=8765

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

Figura 5 - arquivo de propriedades

Uma vez criado este código podemos subir o “gateway” com o comando “mvn spring-boot:run”. Note que ele deverá se registrar junto ao “name server” como os demais serviços (figura 6).

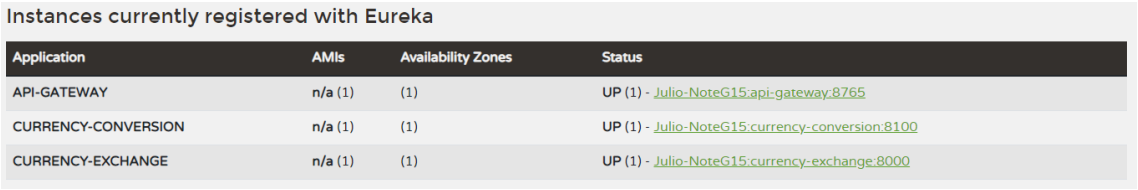

Figura 6 – “Gateway” e demais micros serviços registrados no “name server”

Para testar o uso do “gateway” use as URLs que seguem:

http://localhost:8765/currency-exchange/currency-exchange/from/USD/to/INR

http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/INR/quantity/10

Passo 2: configurando rotas no “gateway”

No passo 1 criamos um “gateway” que usa diretamente o “name server” para localizar os serviços locais. Isso fez com que necessitássemos detalhar o nome do serviço nas requisições HTTP, o que pode atrapalhar o cliente desses endpoints. O “gateway”, entretanto, nos permite mapear rotas de maneira que podemos adequar a maneira como os clientes irão acessar os serviços.

Para fazermos isso podemos “desligar” a propriedade “discovery.locator” que habilitamos no passo 1 e criamos um objeto de configuração do “gateway”. O código do nosso “APIGatewayConfiguration” está na figura 7. Adicione um novo arquivo “ApiGatewayConfiguration.java” no projeto da aplicação.

```
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApiGatewayConfiguration {
    @Bean
       public RouteLocator gatewayRouter(RouteLocatorBuilder builder) {
              return builder.routes()
                           .route(p -> p.path("/currency-exchange/**")
                                        .uri("lb://currency-exchange"))
                           .route(p -> p.path("/currency-conversion/**")
                                        .uri("lb://currency-conversion"))
                     .route(p -> p.path("/currency-conversion-feign/**")
```

```
                                        .uri("lb://currency-conversion"))
                            .build();
        }
}
```

Figura 7 – configurando o “Gateway”

O módulo de configuração nos permite construir novas rotas a partir das demandas que chegam. Note que recebemos um “RouteLocatorBuilder”. Usando este “builder” podemos reconfigurar a rota usando uma série de métodos auxiliares até chamar o método “build” que é quem realmente cria a rota (este é uma aplicação direta do padrão de projeto Builder).

O método “route” nos permite indicar que as solicitações iniciadas por “/currency-exchange” devem ser redirecionadas para o serviço “currency-exchange”. O “lb:” que antecede o nome do serviço já solicita que seja feito o balanceamento de carga. Cada chamada para o método “route” permite criar uma “rota” dessas. Desta forma criamos 3 novas rotas que permitem que acionemos os nossos micros serviços sem a necessidade de indicar o nome do micro serviço na solicitação. A API suporta diferentes formatos de configuração de rotas de modo avançado, incluindo filtragens de diferentes tipos e modificação das requisições/respostas do HTPP.

Teste com as URLs que seguem:

http://localhost:8765/currency-exchange/from/USD/to/INR

http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10

http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10

Passo 3: configurando ações globais

O gateway permite que configuremos também ações globais (para monitoramento, por exemplo). O código da figura 8 pode ser acrescentado em nosso “gateway” para criar um “log” das requisições que são feitas. Acrescente o código em um novo arquivo “LoggingFilter.java” e veja os resultados na janela de console do “gateway”.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;

import reactor.core.publisher.Mono;

@Component
public class LoggingFilter implements GlobalFilter{

    private Logger logger = LoggerFactory.getLogger(LoggingFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        logger.info(
            "Path of the request received -> {}",
                    exchange.getRequest().getPath()
        );
             return chain.filter(exchange);
    }
}
```

Figura 8 – código para monitorar as requisições que chegam no “gateway”

Teste com as URLs que seguem:

http://localhost:8765/currency-exchange/from/USD/to/INR

http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10

http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10

Exercício

Verifique se o micro serviço “information-collector” está perfeitamente integrado ao “gateway”. Faça os ajustes necessários para tanto.