Spring Cloud - Gateway

Introduction

In a distributed environment, services need to communicate with each other. However, this is interservice communication. We also have use-cases where a client outside our domain wants to hit our services for the API. So, either we can expose the address of all our microservices which can be called by clients OR we can create a Service Gateway which routes the request to various microservices and responds to the clients.

Creating a Gateway is much better approach here. There are two major advantages −

- The security for each individual services does not need to maintained.
- And, cross-cutting concerns, for example, addition of meta-information can be handled at a single place.

Netflix Zuul and **Spring Cloud Gateway** are two well-known Cloud Gateways which are used to handle such situations. In this tutorial, we will use Spring Cloud Gateway.

Spring Cloud Gateway – Dependency Setting

Let us use the case of Restaurant which we have been using. Let us add a new service (gateway) in front of our two services, i.e., Restaurant services and Customer Service. First, let us update the **pom.xml** of the service with the following dependency −

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

And then, annotate our Spring application class with the correct annotation, i.e., @EnableDiscoveryClient.

```
package com.tutorialspoint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
@SpringBootApplication
@EnableDiscoveryClient
public class RestaurantGatewayService{
   public static void main(String[] args) {
      SpringApplication.run(RestaurantGatewayService.class, args);
   }
}
```

We are annotating with @EnableDiscoveryClient because we want to use Eureka service discovery to get the list of hosts which are serving a particular use-case

Dynamic Routing with Gateway

The Spring Cloud Gateway has three important parts to it. Those are −

- **Route** − These are the building blocks of the gateway which contain URL to which request is to be forwarded to and the predicates and filters that are applied on the incoming requests.
- **Predicate** − These are the set of criteria which should match for the incoming requests to be forwarded to internal microservices. For example, a path predicate will forward the request only if the incoming URL contains that path.
- **Filters** − These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client.

Let us write a simple configuration for the Gateway for our Restaurant and Customer service.

```
spring:
   application:
      name: restaurant-gateway-service
   cloud:
      gateway:
      discovery:
         locator:
            enabled: true
      routes:
         - id: customers
            uri: lb://customer-service
            predicates:
            - Path=/customer/**
```

```
            - id: restaurants
               uri: lb://restaurant-service
               predicates:
               - Path=/restaurant/**
server:
   port: ${app_port}
eureka:
   client:
      serviceURL:
         defaultZone: http://localhost:8900/eureka
```

Points to note about the above configuration −

- We have enabled the **discovery.locator** to ensure that the gateway can read from the Eureka server.
- We have used Path predicated here to route the request. What this means is that any request which begins with /**customer** would be routed to Customer Service and for /**restaurant**, we will forward that request to Restaurant Service.

Now let us setup other services prior to the Gateway service −

- Start the Eureka Server
- Start the Customer Service
- Start the Restaurant Service

Now, let us compile and execute the Gateway project. We will use the following command for the same −

```
java -Dapp_port=8084 -jar .\target\spring-cloud-gateway-1.0.jar
```

Once this is done, we have our Gateway ready to be tested on port 8084. Let's first hit http://localhost:8084/customer/1 and we see the request is correctly routed to Customer Service and we get the following output −

```
{
   "id": 1,
   "name": "Jane",
   "city": "DC"
}
```

And now, hit our restaurant API, i.e., http://localhost:8084/restaurant/customer/1 and we get the following output −

```
[
   {
      "id": 1,
      "name": "Pandas",
      "city": "DC"
   },
   {
      "id": 3,
      "name": "Little Italy",
      "city": "DC"
   }
]
```

This means that both the calls were correctly routed to the respective services.

Predicates & Filters Request

We had used Path predicate in our above example. Here are a few other important predicates −

Predicate	**Description**
Cookie predicate (input: name and regex)	Compares the cookie with the ‘name’ to the ‘regexp’
Header predicate (input: name and regex)	Compares the header with the ‘name’ to the ‘regexp’
Host predicate (input: name and regex)	Compares the ‘name’ of the incoming to the ‘regexp’
Weight Predicate (input: Group name and the weight)	Weight Predicate (input: Group name and the weight)

Filters are used to add/remove data from the request before sending the data to the downstream service or before sending the response back to the client.

Following are a few important filters for adding metadata.

Filter	Description
Add request header filter (input: header and the value)	Add a ‘header’ and the ‘value’ before forwarding the request downstream.
Add response header filter (input: header and the value)	Add a ‘header’ and the ‘value’ before forwarding the request upstream that is to the client.
Redirect filter (input: status and URL)	Adds a redirect header along with the URL before passing over o the downstream host.
ReWritePath (input: regexp and replacement)	This is responsible for rewriting the path by replacing the ‘regexp’ matched string with the input replacement.

The exhaustive list for filters and predicates is present at https://cloud.spring.io/spring-cloudgateway/reference/html/#the-rewritepath-gatewayfilter-factory

Monitoring

For monitoring of the Gateway or for accessing various routes, predicates, etc., we can enable the actuator in the project. For doing that, let us first update the pom.xml to contain the actuator as a dependency.

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

For monitoring, we will use a separate application property file which would contain flags to enable the actuator. So, here is how it would look like −

```
spring:
   application:
      name: restaurant-gateway-service
   cloud:
      gateway:
         discovery:
            locator:
               enabled: true
         routes:
            - id: customers
              uri: lb://customer-service
              predicates:
              - Path=/customer/**
            - id: restaurants
              uri: lb://restaurant-service
```

```
            predicates:
            - Path=/restaurant/**
server:
   port: ${app_port}
eureka:
   client:
      serviceURL:
         defaultZone: http://localhost:8900/eureka
management:
   endpoint:
      gateway:
         enabled: true
   endpoints:
      web:
         exposure:
            include: gateway
```

Now, to list all the routes, we can hit: http://localhost:8084/actuator/gateway/routes

```
[
   {
      "predicate": "Paths: [/customer/**], match trailing slash: true",
      "route_id": "customers",
      "filters": [],
      "uri": "lb://customer-service",
      "order": 0
   },
   {
      "predicate": "Paths: [/restaurant/**], match trailing slash: true",
      "route_id": "restaurants",
      "filters": [],
      "uri": "lb://restaurant-service",
      "order": 0
   }
]
```

Other important APIs for monitoring −

API	Description
GET /actuator/gateway/routes/{id}	Get information about a particular route
POST /gateway/routes/{id_to_be assigned}	Add a new route to the Gateway
DELETE /gateway/routes/{id}	Remove the route from Gateway
POST /gateway/refresh	Remove all the cache entries