Introduction to Spring Cloud Netflix – Eureka

1. Overview

In this tutorial, we'll introduce client-side service discovery via "Spring Cloud Netflix Eureka".

Client-side service discovery allows services to find and communicate with each other without hard coding hostname and port. The only 'fixed point' in such an architecture consists of a service registry with which each service has to register.

A drawback is that all clients must implement a certain logic to interact with this fixed point. This assumes an additional network round trip before the actual request.

With Netflix Eureka each client can simultaneously act as a server, to replicate its status to a connected peer. **In other words, a client retrieves a list of all connected peers of a service registry and makes all further requests to any other services through a load-balancing algorithm**.

To be informed about the presence of a client, they have to send a heartbeat signal to the registry.

To achieve the goal of this write-up, we will implement three microservices:

1. a service registry (Eureka Server),
2. a REST service which registers itself at the registry (Eureka Client) and
3. a web-application, which is consuming the REST service as a registry-aware client (Spring Cloud Netflix Feign Client).

2. Eureka Server

To implement a Eureka Server for using as service registry is as easy as: adding spring-cloud-starter-netflix-eureka-server to the dependencies, enable the Eureka Server in a @SpringBootApplication per annotate it with @EnableEurekaServer and configure some properties. But we'll do it step by step.

Firstly we'll create a new Maven project and put the dependencies into it. You have to notice that we're importing the spring-cloud-starter-parent to all projects described in this tutorial:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
   <version>2.0.2.RELEASE</version>
</dependency>

<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-parent</artifactId>
         <version>Finchley.SR2</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

Next, we're creating the main application class:

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

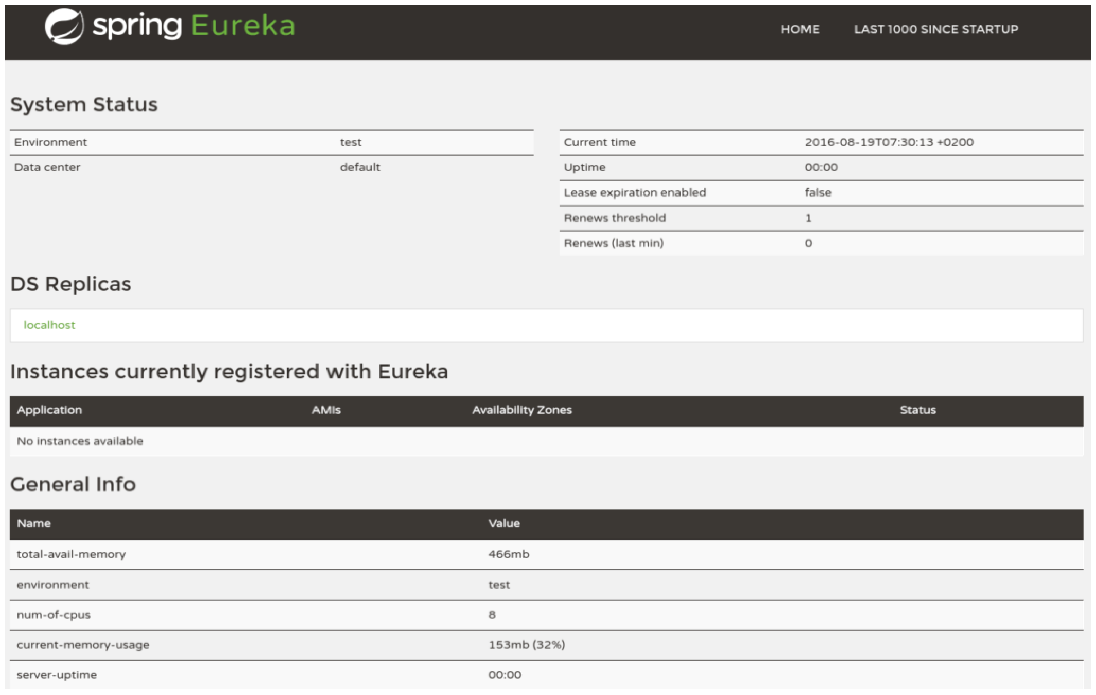
Finally, we're configuring the properties in YAML format; so an application.yml will be our configuration file:

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Here we're configuring an application port – 8761 is the default one for Eureka servers. We are telling the built-in Eureka Client not to register with 'itself', because our application should be acting as a server.

Now we will point our browser to http://localhost:8761 to view the Eureka dashboard, where we will later inspecting the registered instances.

At the moment, we see basic indicators such as status and health indicators.



System Status

Environment	test
Data center	default

Current time	2016-08-19T07:30:13 +0200
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	466mb
environment	test
num-of-cpus	8
current-memory-usage	153mb (32%)
server-uptime	00:00

3. Eureka Client
For a @SpringBootApplication to be discovery-aware, we have to include some Spring Discovery Client (for example spring-cloud-starter-netflix-eureka-client) into our classpath.

Then we need to annotate a @Configuration with either @EnableDiscoveryClient or @EnableEurekaClient – note that this annotation is optional if we have the spring-cloud-starter-netflix-eureka-client dependency on the classpath.

The latter tells Spring Boot to use Spring Netflix Eureka for service discovery explicitly. To fill our client application with some sample-life, we'll also include the spring-boot-starter-web package in the pom.xml and implement a REST controller.

But first, we will add the dependencies:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-starter</artifactId>
    <version>2.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.0.1.RELEASE</version>
</dependency>
```

Here we will implement the main application class:

```
@SpringBootApplication
@RestController
public class EurekaClientApplication implements GreetingController {

    @Autowired
    @Lazy
    private EurekaClient eurekaClient;

    @Value("${spring.application.name}")
    private String appName;

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

    @Override
    public String greeting() {
        return String.format("Hello from '%s'!", eurekaClient.getApplication(appName).getName());
    }
}
```

And the GreetingController interface:

```
public interface GreetingController {
    @RequestMapping("/greeting")
    String greeting();
}
```

Here, instead of the interface, we could also simply declare the mapping inside the EurekaClientApplication class. The interface can be useful if we want to share it between server and client.

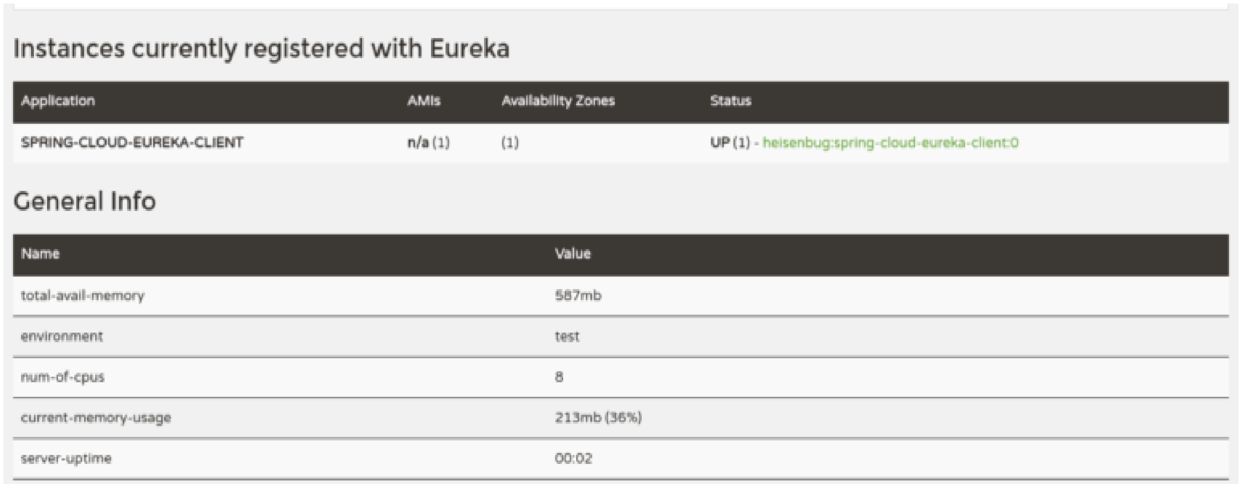
Next, we have to set-up an application.yml with a configured Spring application name to uniquely identify our client in the list of registered applications.

We can let Spring Boot choose a random port for us because later we are accessing this service with its name, and finally, we have to tell our client, where it has to locate the registry:

```
spring:
  application:
    name: spring-cloud-eureka-client
server:
  port: 0
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
  instance:
    preferIpAddress: true
```

When we decided to set up our Eureka Client this way, we had in mind that this kind of service should later be easily scalable.

Now we will run the client and point our browser to http://localhost:8761 again, to see its registration status on the Eureka Dashboard. By using the Dashboard, we can do further configuration e.g. link the homepage of a registered client with the Dashboard for administrative purposes. The configuration options, however, are beyond the scope of this article.



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SPRING-CLOUD-EUREKA-CLIENT	n/a (1)	(1)	UP (1) - heisenbug:spring-cloud-eureka-client:0

General Info

Name	Value
total-avail-memory	587mb
environment	test
num-of-cpus	8
current-memory-usage	213mb (36%)
server-uptime	00:02

4. Feign Client

To finalize our project with three dependent microservices, we will now implement a REST-consuming web application using Spring Netflix Feign Client.

Think of Feign as discovery-aware Spring RestTemplate using interfaces to communicate with end-points. This interfaces will be automatically implemented at runtime and instead of service-urls, it is using service-names.

Without Feign we would have to autowire an instance of EurekaClient into our controller with which we could receive a service-information by service-name as an Application object.

We would use this Application to get a list of all instances of this service, pick a suitable one and use this InstanceInfo to get hostname and port. With this, we could do a standard request with any http client.

For example:

```
@Autowired
private EurekaClient eurekaClient;

public void doRequest() {
    Application application
      = eurekaClient.getApplication("spring-cloud-eureka-client");
    InstanceInfo instanceInfo = application.getInstances().get(0);
    String hostname = instanceInfo.getHostName();
    int port = instanceInfo.getPort();
    // ...
}
```

A RestTemplate can also be used to access Eureka client-services by name, but this topic is beyond this write-up.

To set up our Feign Client project, we’re going to add the following four dependencies to its pom.xml:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.0.1.RELEASE</version>
</dependency>
```

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
   <version>2.0.1.RELEASE</version>
</dependency>
```

The Feign Client is located in the spring-cloud-starter-feign package. To enable it, we have to annotate a @Configuration with @EnableFeignClients. To use it, we simply annotate an interface with @FeignClient("service-name") and auto-wire it into a controller.

A good method to create such Feign Clients is to create interfaces with @RequestMapping annotated methods and put them into a separate module. This way they can be shared between server and client. On server-side, you can implement them as @Controller, and on client-side, they can be extended and annotated as @FeignClient.

Furthermore, the spring-cloud-starter-eureka package needs to be included in the project and enabled by annotating the main application class with @EnableEurekaClient.

The spring-boot-starter-web and spring-boot-starter-thymeleaf dependencies are used to present a view, containing fetched data from our REST service.

This will be our Feign Client interface:

```
@FeignClient("spring-cloud-eureka-client")
public interface GreetingClient {
   @RequestMapping("/greeting")
   String greeting();
}
```

Here we will implement the main application class which simultaneously acts as a controller:

```
@SpringBootApplication
@EnableFeignClients
@Controller
public class FeignClientApplication {
   @Autowired
   private GreetingClient greetingClient;

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

   @RequestMapping("/get-greeting")
   public String greeting(Model model) {
      model.addAttribute("greeting", greetingClient.greeting());
      return "greeting-view";
   }
}
```

This will be the HTML template for our view:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Greeting Page</title>
    </head>
    <body>
        <h2 th:text="${greeting}"/>
    </body>
</html>
```

At least the application.yml configuration file is almost the same as in the previous step:

```
spring:
  application:
    name: spring-cloud-eureka-feign-client
server:
  port: 8080
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
```

Now we can build and run this service. Finally, we'll point our browser to http://localhost:8080/get-greeting and it should display something like the following:
1

```
Hello from SPRING-CLOUD-EUREKA-CLIENT!
```

5. TransportException: Cannot execute request on any known server

While running Eureka server we often run into exceptions like:

```
com.netflix.discovery.shared.transport.TransportException: Cannot execute request on any known server
```

Basically, this happens due to the wrong configuration in application.properties or application.yml. Eureka provides two properties for the client that can be configurable.

- **registerWithEureka**: If we make this property as true then while the server starts the inbuilt client will try to register itself with the Eureka server.
- **fetchRegistry**: The inbuilt client will try to fetch the Eureka registry if we configure this property as true.

Now when we start up the Eureka server, we don't want to register the inbuilt client to configure itself with the server.

If we mark the above properties as true (or don't configure them as they're true by default) while starting the server, the inbuilt client tries to register itself with the Eureka server and also tries to fetch registry which is not yet available. As a result, we get TransportException.

So we should never configure these properties as true in the Eureka server applications. The correct settings that should be put in application.yml are given below:

```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

6. Conclusion

As we've seen, we're now able to implement a service registry using Spring Netflix Eureka Server and register some Eureka Clients with it.

Because our Eureka Client from step 3 listens on a randomly chosen port, it doesn't know its location without the information from the registry. With a Feign Client and our registry, we can locate and consume the REST service, even when the location changes.

Finally, we've got a big picture of using service discovery in a microservice architecture.