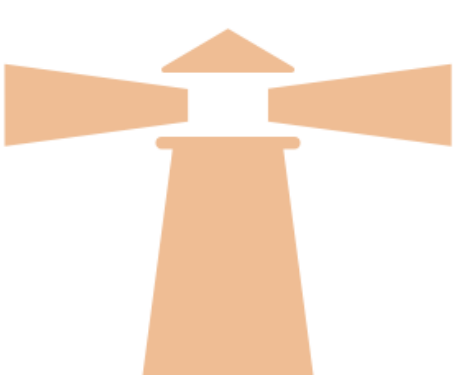
Microservices with Spring Boot — Creating our Microserivces & Gateway (Part 2)

Eureka Server, Client, and the Gateway

OMAR ELGABRY Follow

Jun 11, 2018 · 5 min read



Service Registry, Image/Gallery Services, and the Gateway

> ***The Github repository for the application: https://github.com/OmarElGabry/microservices-spring-boot***

. . .

Recalling our application architecture, we have a service registry, image service, gallery service, and a gateway.

The gallery service uses the image service underneath, and retrieves a list of all images to be displayed.

> *The version of spring boot applications we're going to create is: 2.0.0.RELEASE.*

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
  <relativePath/>
</parent>
```

Eureka Server

It's the naming server, or called service registry. It's duty to give names to each microservice. *Why?*

1. No need to hardcode the IP addresses of microservices.
2. What if services use dynamic IP addresses; when autoscaling.

So, every service registers itself with Eureka, and pings Eureka server to notify that it's alive.

If Eureka server didn't receive any notification from a service. This service is unregistered from the Eureka server automatically.

The steps are fairly simple. 1, 2, 3, … and we're done!.

Ok. So, as usual, create a maven project, or use spring initializr. In the `pom.xml` file, make sure to include these dependencies: Web, Eureka Server, and DevTools (optional).

```
....
 <dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <optional>true</optional>
        </dependency>
</dependencies>
....
```

pom.xml hosted with ♡ by GitHub view raw

Next, in the `application.properties` file, we need to set some configurations.

```
# Give a name to the eureka server
spring.application.name=eureka-server
```

```
spring.application.name=eureka-server

# default port for eureka server
server.port=8761

# eureka by default will register itself as a client. So, we need to set it to false.
# What's a client server? See other microservices (image, gallery, auth, etc).
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

application.properties hosted with ♡ by GitHub view raw

Finally, in your spring boot main application class, enable Eureka server using `@EnableEurekaServer` annotation.

```
package com.eureka.server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer     // Enable eureka server

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

SpringEurekaServerApplication.java hosted with ♡ by GitHub view raw

So far so good?. Next, we create our services; image and gallery.

Image Service

The Eureka client service is an independent service in a microservice architecture. It could be for payment, account, notification, auth, config, etc.

The image service acts as a data source for images, each image has an id, title, and url. Simple enough?.

Ok. So, for the `pom.xml` file, instead of Eureka Server, use Eureka Client.

```
....
 <dependencies>
        <dependency>
               <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</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-starter-data-rest</artifactId>
```

```
        </dependency>
         <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <optional>true</optional>
         </dependency>
</dependencies>
....
```

pom.xml hosted with ♡ by GitHub view raw

In the `application.properties` file, we define configurations (as before)

```
# serivce name
spring.application.name=image-service
# port
server.port=8200
# eureka server url
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

application.properties hosted with ♡ by GitHub view raw

Then, enable eureka client using `@EnableEurekaClient` annotation.

```
package com.eureka.image;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

```

```java
@SpringBootApplication
@EnableEurekaClient     // Enable eureka client. It inherits from @EnableDiscoveryClient.
public class SpringEurekaImageApp {
        public static void main(String[] args) {
                SpringApplication.run(SpringEurekaImageApp.class, args);
        }
}
```

SpringEurekaImageApp.java hosted with ♡ by GitHub view raw

Now, our image service is going to expose some data through endpoints, right?. So, we need to create a controller, and define the action methods.

```java
package com.eureka.image.controllers;

import java.util.Arrays;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.eureka.image.entities.Image;

@RestController
@RequestMapping("/")
public class HomeController {
        @Autowired
        private Environment env;

```

```
19          @RequestMapping("/images")
20          public List<Image> getImages() {
21                  List<Image> images = Arrays.asList(
22                          new Image(1, "Treehouse of Horror V", "https://www.imdb.com/title/tt0096
23                          new Image(2, "The Town", "https://www.imdb.com/title/tt0096697/mediaview
24                          new Image(3, "The Last Traction Hero", "https://www.imdb.com/title/tt009
25                  return images;
26          }
27  }
```

HomeController.java hosted with ♡ by GitHub view raw

> *Don't forget to create the Image entity class with three fields; id, title, and url.*

Gallery Service

The Eureka client service can be also a REST client that calls (consumes) other services (REST API services) in our microservice application.

So, for example, the gallery service calls image service to get a list of all images, or maybe only images created during a specific year.

The calls from this REST client to the other services can be done using:

1. RestTemplate. An object that's capable of sending requests to REST API services.
2. FeignClient (acts like a proxy), and provides another approach to RestTemplate.

Both, load balance requests across the services.

> ***— What's load balancing?***
>
> *What if more than one instance of a service running on different ports. So, we need to balance the requests among all the instances of a service.*
>
> *When using 'Ribbon' approach (default), requests will be distributed equally among them.*

So, as usual, we start by `pom.xml` . It's the same as the one for image service. Next is the `application.properties` file.

```
spring.application.name=gallery-service
server.port=8100
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

application.properties hosted with ♡ by GitHub view raw

In the spring boot main application class, besides enabling the eureka client, we need to create a bean for `RestTemplate` to call the image service.

```
package com.eureka.gallery;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableEurekaClient             // Enable eureka client.
public class SpringEurekaGalleryApp {

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

@Configuration
class RestTemplateConfig {

        // Create a bean for restTemplate to call services
        @Bean
        @LoadBalanced           // Load balance between service instances running at different p
        public RestTemplate restTemplate() {
            return new RestTemplate();
        }
}
```

```
}
```

SpringEurekaGalleryApp.java hosted with ♡ by GitHub view raw

In the controller, call image service using `RestTemplate` and return the result.

```java
package com.eureka.gallery.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import com.eureka.gallery.entities.Gallery;

@RestController
@RequestMapping("/")
public class HomeController {
        @Autowired
        private RestTemplate restTemplate;

        @Autowired
        private Environment env;

        @RequestMapping("/")
        public String home() {
```

```
25                  // This is useful for debugging
26                  // When having multiple instance of gallery service running at different ports.
27                  // We load balance among them, and display which instance received the request.
28                  return "Hello from Gallery Service running at port: " + env.getProperty("local.s
29          }
30
31          @RequestMapping("/{id}")
32          public Gallery getGallery(@PathVariable final int id) {
33                  // create gallery object
34                  Gallery gallery = new Gallery();
35                  gallery.setId(id);
36
37                  // get list of available images
38                  List<Object> images = restTemplate.getForObject("http://image-service/images/",
39                  gallery.setImages(images);
40
41                  return gallery;
42          }
43
44          // -------- Admin Area --------
45          // This method should only be accessed by users with role of 'admin'
46          // We'll add the logic of role based auth later
47          @RequestMapping("/admin")
48          public String homeAdmin() {
49                  return "This is the admin area of Gallery service running at port: " + env.getPr
50          }
51  }
```

Ok, here's one thing to note. Since we are using `restTemplate` — *which in turn uses Eureka Server for naming of services, and Ribbon for load balancing*.

So, we can use the service name (like `image-service`) instead of `localhost:port`

Gateway — Zuul

When calling any service from the browser, we can't call it by it's name as we did from Gallery service — *This is used internally between services.*

And as we spin more instances of services, each with a different port numbers, So, now the question is: *How can we call the services from the browser and distribute the requests among their instances running at different ports?*

Well, a common solution is to use a Gateway.

A gateway is a single entry point into the system, used to handle requests by routing them to the corresponding service. It can also be used for authentication, monitoring, and more.

What's Zuul?

It's a proxy, gateway, an intermediate layer between the users and your services.

Eureka server solved the problem of giving names to services instead of hardcoding their IP addresses.

But, still, we may have more than one service (instances) running on different ports. So, Zuul …

1. Maps between a prefix path, say `/gallery/**` and a service `gallery-service`. It uses Eureka server to route the requested service.
2. It load balances (using Ribbon) between instances of a service running on different ports.
3. *What else?* We can filter requests, add authentication, etc.

In the `pom.xml` add dependencies: Web, Eureka Client, and Zuul

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

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
....
```

pom.xml hosted with ♡ by GitHub view raw

> *It's worth mentioning that Zuul acts as a Eureka client. So, we give it a name, port, and link to Eureka server (same as we did with image service).*

```
server.port=8762
spring.application.name=zuul-server
eureka.client.service-url.default-zone=http://localhost:8761/eureka/

# A prefix that can added to beginning of all requests.
#zuul.prefix=/api

# Disable accessing services using service name (i.e. gallery-service).
# They should be only accessed through the path defined below.
zuul.ignored-services=*

# Map paths to services
zuul.routes.gallery-service.path=/gallery/**
zuul.routes.gallery-service.service-id=gallery-service

```

application.properties hosted with ♡ by GitHub view raw

Finally, enable Zuul and Eureka Client.

```java
package com.eureka.zuul;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
@EnableEurekaClient             // It acts as a eureka client
@EnableZuulProxy                // Enable Zuul

public class SpringZuulApplication {

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

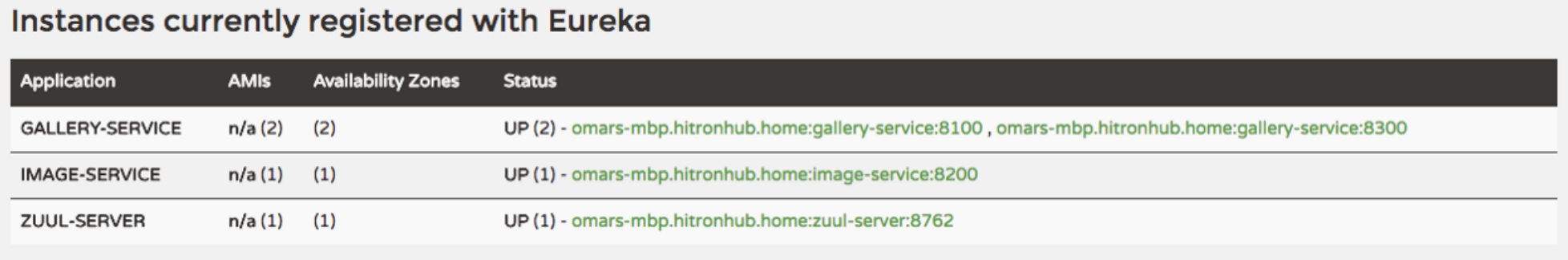
SpringZuulApplication.java hosted with ♡ by GitHub view raw

Testing our Microservices

Ok. So, we have a service discovery; Eureka server. Two services; image and gallery. And a gateway; Zuul.

To test our application, run eureka server, zuul, and then the two services. Then, go to Eureka Server running at `localhost:8761` , you should see the running services.

> *To run multiple instances. In eclipse, go to Run →Configurations/Arguments →VM options and add* `-Dserver.port=8300`



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
GALLERY-SERVICE	n/a (2)	(2)	UP (2) - omars-mbp.hitronhub.home:gallery-service:8100 , omars-mbp.hitronhub.home:gallery-service:8300
IMAGE-SERVICE	n/a (1)	(1)	UP (1) - omars-mbp.hitronhub.home:image-service:8200
ZUUL-SERVER	n/a (1)	(1)	UP (1) - omars-mbp.hitronhub.home:zuul-server:8762

To ping the gallery service, send a request to the gateway, adding the path for the galley service `localhost:8762/gallery` .

You should see the below message, and if you hit the url again, the request will be routed to the second instance of gallery service; thanks to the load balancer.

> *Hello from Gallery Service running at port: 8100*

> *Hello from Gallery Service running at port: 8300*

To get all images, hit `localhost:8762/gallery/1` in the browser

```
{
    "id": 1,
    "images": [
        {
            "id": 1,
            "title": "Treehouse of Horror V",
            "url": "https://.../rm3842005760"
        },
        {
            "id": 2,
            "title": "The Town",
            "url": "https://.../rm3698134272"
        },
        {
            "id": 3,
            "title": "The Last Traction Hero",
            "url": "https://.../rm1445594112"
        }
    ]
}
```

But, we aren't done yet!. We still have to authenticate the users. Next, we'll use JSON Web Tokens (JWT) for authentication.

. . .

Thank you for reading! If you enjoyed it, please clap 👏 for it.

Microservices Java Software Development Programming Other