Eureka: Service Discovery

In this lesson, we'll discuss service discovery with Eureka.

WE'LL COVER THE FOLLOWING

Introduction

Eureka implements service discovery as we learned in the introduction.

For synchronous communication, microservices have to find out at **which port and IP address other microservices can be accessed**.

Features

Let's discuss some essential characteristics of Eureka.

REST interface

Eureka has a **REST interface**.

Microservices can use this interface to **register or request information** about other microservices.

Replication

Eureka supports **replication**.

The information from the Eureka servers is distributed to other servers enabling the system to **compensate for the failure** of Eureka servers.

In a distributed system, service discovery is essential for communication between microservices. Therefore, service discovery must be implemented in a way that a **failure of one server does not cause the entire service discovery to fail**.

Cache improved performance

Due to **caches on the client**:

- The performance of Eureka **very good**.
- **Availability is improved** because the information is stored on the client compensating for server failure.
- The server only sends information to the client about new or deleted microservices and not information about all registered services, **communication very efficient**.

Amazon Web Services support

Netflix supports **AWS (Amazon Web Services)**, i.e., the Amazon Cloud.

In AWS, servers run in **availability zones**. These are basically separate data centers. The failure of an availability zone does not affect other availability zones.

Several availability zones form a **region** which is located in a geographical zone. For example, the data centers for the region called EU-West-1 are located in Ireland.

Eureka can take regions and availability zones into account and offer a microservice instance from the same availability zone to a client as a result of

the service discovery in order to increase speed.

Crash detection

Eureka expects the microservices to regularly send **heartbeats**.

In this way, Eureka detects crashed instances and excludes them from the system. This increases the probability that Eureka will return service instances that are available.

However, it may happen that a microservice instance is returned even though it is no longer running.

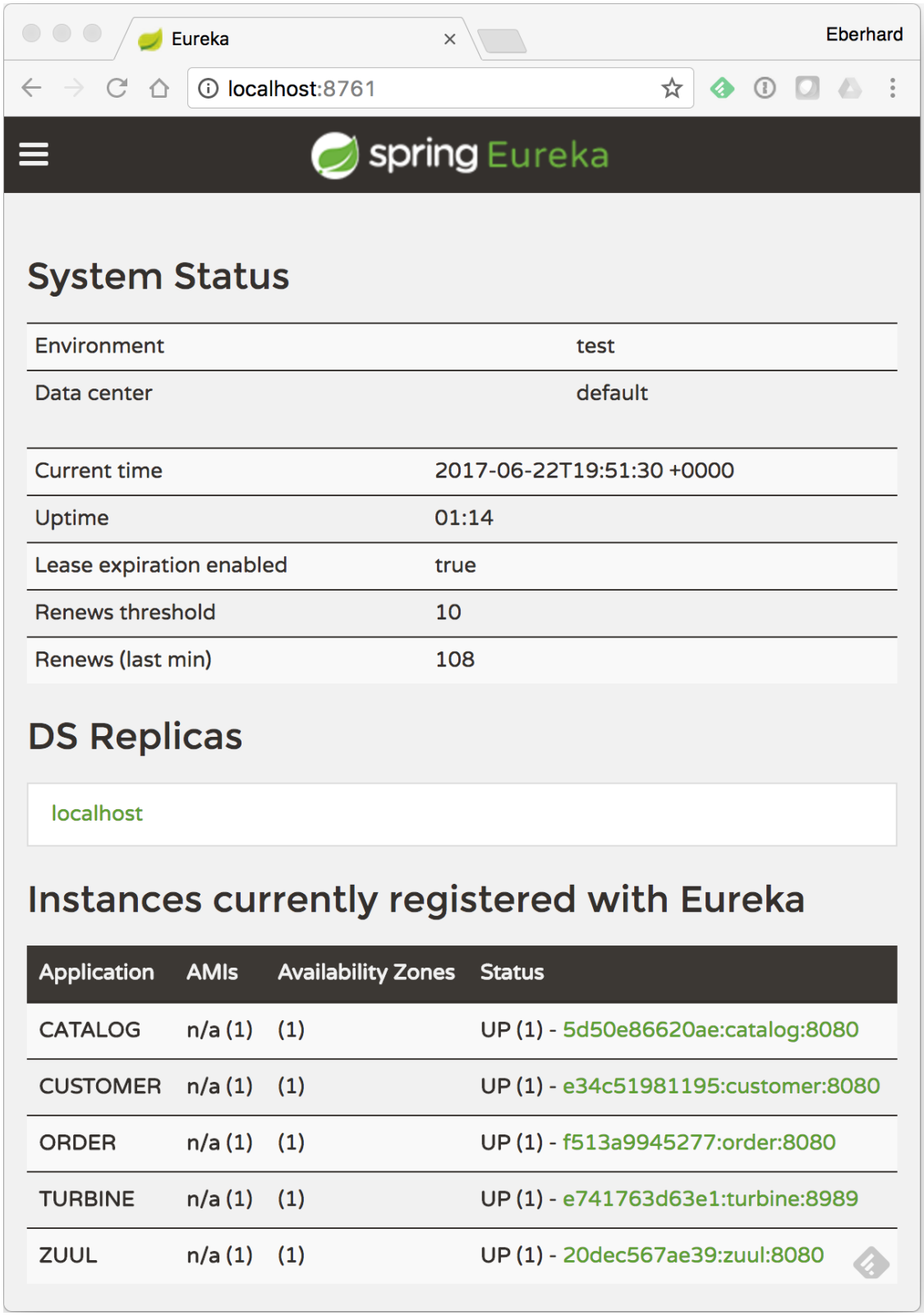
But whether a microservice instance has crashed or not is obvious once it is used so that a different instance can be used if it has crashed.

Servers

The Netflix Eureka project is available for download at GitHub.You can build the project and get both the server and the client.

The Spring Cloud project also supports Eureka, the Eureka server can even be started as a Spring Boot application. For this, the main class which also has the annotation `@SpringBootApplication` has to be annotated with `@EnableEurekaServer`. In `pom.xml` in the `dependencyManagement` section, the Spring Cloud dependencies have to be imported, and a dependency on the library `spring-cloud-starter-eureka-server` has to be inserted. In addition, a configuration in the `application.properties` file is necessary. The project `microservice-demo-eureka-server` provides all of that and implements an Eureka server.

At first glance, it does not seem to make sense to build the Eureka server by yourself in this way, especially since the implementation essentially consists of an annotation. But the Eureka server can be treated like all the other microservices. The Spring Cloud Eureka server is a JAR file and, like all other microservices, can be stored and started in a Docker image. It is also possible to secure it like a Java web application with Spring security and configure logging and monitoring as with all other microservices.



Eureka Dashboard

Eureka provides a **dashboard**, see the screenshot above.

- It displays an overview of the microservices which are registered with Eureka.
    - This includes the names of the microservices and the URLs at which they can be accessed.
- However, the URLs only work in the Docker internal network meaning

the links in the dashboard do not work.

- The dashboard is accessible on the Docker host at port 8761 i.e. http://localhost:8761/ if the example runs locally.

Client

Each microservice is a Eureka client and must **register** with the Eureka server in order to inform the Eureka server about the name of the microservice, the IP-address, and port at which it can be reached.

Spring Cloud simplifies the configuration for clients.

Registration

The client has to have a dependency on `spring-cloud-starter-eureka` to add the necessary libraries. The main class which has the annotation `@SpringBootApplication` additionally has to be annotated with `@EnableEurekaClient`.

An alternative is `@EnableDiscoveryClient`. In contrast to `@EnableEurekaClient`, the annotation `@EnableDiscoveryClient` is generic. Thus it also works with Consul (see chapter 11).

```
spring.application.name=catalog
eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/
eureka.instance.leaseRenewalIntervalInSeconds=5
eureka.instance.metadataMap.instanceId=${spring.application.name}:${random
.value}
eureka.instance.preferIpAddress=true
```

In the file `application.properties` shown above, appropriate settings must be entered for the application to register.

- `spring.application.name` contains the name under which the application registers at the Eureka server.
- `eureka.client.serviceUrl.defaultZone` defines which Eureka server is to be used.
- The setting `eureka.instance.leaseRenewalIntervalInSeconds` ensures that the registration information is replicated every **five seconds** and is

replicated faster than the default setting.

    - This makes new microservice instances **usable more quickly**. In production, this value should **not** be set so low that there is no network traffic.

- `eureka.instance.metadataMap.instanceId` provides each microservice instance with a random ID, for instance, to be able to discriminate between two instances for load balancing.

- Due to `eureka.instance.preferIpAddress`, the services register with their IP address and not with their host name. This avoids problems that arise because hostnames cannot be resolved in the Docker environment.

During registration, the name of the microservice is automatically converted to uppercase letters. Thus, `order` is turned into `ORDER`.

Other programming languages

- For programming languages other than Java, a library must be used to access Eureka.
- There are some libraries that implement Eureka clients for certain programming languages.
- Of course, Eureka provides a REST interface which can also be used.

Sidecars

To use the Netflix infrastructure with other programming languages, it is also possible to use a **sidecar**.

> A **sidecar** is an application written in Java that uses the Java libraries to talk to the Netflix infrastructure.

The application uses the sidecar to communicate with the Netflix infrastructure, so the sidecar is a **helper** for the real application. That way the application can be written in **any programming language**. Essentially the sidecar is the **interface** to the Netflix infrastructure.

In this way, Eureka can support other programming languages; but this

requires an additional process that **consumes additional resources**.

Netflix itself offers Prana as a sidecar. Spring Cloud also provides an implementation of such a sidecar.

Access to other services

In the example in this chapter, **Ribbon** implements the access to other services in order to implement load balancing across multiple instances.

Thus, the Eureka API is only used via Ribbon to find information about other microservices.

QUIZ

1 In what case could Eureka return an instance of a microservice that has crashed **despite** the implementation of heartbeats?

COMPLETED 0%

1 of 4

In the next lesson, we'll study routing with Zuul in more detail.