heise Developer

Architektur/Methoden 20.10.2015 - 09:01

Stefan Janser

Eureka – Microservice-Registry mit Spring Cloud

Die Magie des Service-Bindings

Eine wichtige Komponente einer Microservice-Infrastruktur ist die Service-Registry, die Services, ihre Instanzen und Lokationen in einer Datenstruktur zusammenfasst. Der Artikel zeigt die Funktionsweise des Netflix-Projekts Eureka an einem einfachen Beispiel.

Eine wichtige Komponente einer Microservice-Infrastruktur ist die Service-Registry, die Services, ihre Instanzen und Lokationen in einer Datenstruktur zusammenfasst. Der Artikel zeigt die Funktionsweise des Netflix-Projekts Eureka an einem einfachen Beispiel.

Netflix erzeugt mit über 35 Prozent die höchste Downstream-Bandbreite in den USA, weit mehr als YouTube (15 %) und Facebook (6 %) **zusammen [1]**. Als die monolithische Architektur dem wachsenden Traffic nicht mehr gerecht wurde, baute sie der Streaming-Dienst zwischen 2008 und 2012 zu einem System aus mehr als hundert Microservices um. Seitdem setzt Netflix auf feingranulare, lose gekoppelte Services. Das Unternehmen hat technische Komponenten und Bibliotheken entwickelt, um die Vielzahl der Services zu überwachen und die hohen Anforderungen an Ausfallsicherheit und Skalierbarkeit zu erfüllen. Sie bilden die Infrastruktur für die Netflix-Services und implementieren gebräuchliche Patterns für verteilte Systeme.

Die Komponenten müssen ihre Qualität in der Netflix-Umgebung täglich in einem komplexen, lastintensiven Szenario beweisen. Dank eines ausgefeilten Monitorings entdeckt Netflix Fehler und Schwachstellen in den Komponenten schnell und behebt sie – entsprechend hoch ist die Qualität der Komponenten. Sie werden im Netflix Open Source Software Center (**Netflix OSS [2]**) als freie Software zur Verfügung gestellt. **Spring Cloud Netflix [3]**, ein Unterprojekt von **Spring Cloud [4]**, benutzt sechs Dienste aus dem Netflix OSS:

- Eureka: Service-Registry und Mid-tier Loadbalancer
- Hystrix: Bibliothek, die Fehler beim Zugriff auf externe Dienste isoliert und so die Ausfallsicherheit in einem verteilten System erhöht (**Circuit Breaker [5]**)
- Ribbon: Library für Interprozesskommunikation mit eingebautem Softwarelastverteiler
- Archaius: Bibliothek für Konfigurationsmanagement, basiert auf Apache Commons Configuration und erweitert das Projekt für den Cloud-Einsatz
- Zuul: Edge-Service für dynamisches Routing, Monitoring, Security, Ausfallsicherheit und einiges andere mehr
- Feign: Library, die das Schreiben von HTTP-Clients vereinfacht

Eine wichtige Komponente einer Microservice-Infrastruktur ist die Service-Registry, die Services, ihre Instanzen und Lokationen in einer Datenstruktur zusammenfasst. Der Artikel zeigt die Funktionsweise von Eureka an einem einfachen Beispiel. Grundkenntnisse in **Spring Boot [6]** sind sinnvoll, da Spring Cloud darauf basiert. Eine Einführung gibt es zum Beispiel bei spring.io.

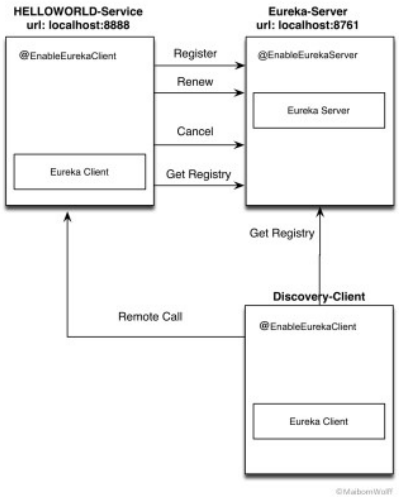
Microservices mit Spring Cloud Netflix registrieren

Mit Eureka werden Microservices unter einem logischen Namen registriert, anschließend lassen sie sich über diesen Namen ansprechen. Damit können Services dynamisch auf unterschiedlichen Instanzen laufen, ohne dass der Nutzer die tatsächliche URL des Services wissen muss.

Eureka verwaltet die Services dynamisch: Das Projekt erkennt, wenn ein Service nicht mehr verfügbar ist und entfernt ihn aus der Registry. Außerdem können sich mehrere Instanzen eines Services unter dem gleichen logischen Namen registrieren; Eureka übernimmt die Lastverteilung zwischen den Instanzen. Services werden durch die Interaktion zwischen Eureka-Client und -Server gefunden. Der Server sammelt die Informationen über die registrierten Services. Für eine höhere Ausfallsicherheit wird er im produktiven Umfeld in der Regel redundant betrieben. Die redundanten Instanzen synchronisieren ihre Service-Registries untereinander. Der Eureka-Client ist eine Bibliothek zur Kommunikation mit dem Eureka-Server.

Das Beispiel in diesem Tutorial besteht aus drei Microservices (s. Abb. 1):

- dem Eureka-Server, der die Service-Registry bereitstellt,
- dem HELLOWORLD-SERVICE, über den der Eureka-Client seinen Dienst beim Eureka-Server registriert,
- und dem Discovery-Client, der den HELLOWORLD-SERVICE über den Eureka-Server bezieht und ihn aufruft.



Das Beispiel besteht aus drei Microservices: Eureka-Server, HELLOWORLD-SERVICE und Discovery-Client (Abb. 1).

Um einen Dienst zu registrieren, schickt der Eureka-Client eine Registrierung mit einem logischen Namen und der dazugehörigen URL an den Eureka-Server (Register). Danach sendet er alle 30 Sekunden einen sogenannten Heartbeat als Lebenszeichen an den Server (Renew). Bleibt der Heartbeat aus, löscht der Eureka-Server die Instanz automatisch im Verzeichnis (Cancel). Der Client aktualisiert die Registry in regelmäßigen Abständen mit allen registrierten Services (Get Registry). Beim Aufruf eines anderen Dienstes kann der Client die URL des aufzurufenden Services anhand des logischen Namens ermitteln und damit einen Remote Call an den Dienst absetzen.

Service-Registry mit dem Eureka-Server einrichten

Der Eureka-Server ist ein um spezielle Annotationen angereicherter Spring-Boot-Service. Er sammelt Informationen über registrierte Services. Zuerst sind die notwendigen Libraries in der Maven *pom.xml* als Dependencies hinzuzufügen. Neben der Referenz auf das Parent-POM (Project Object Model) *spring-cloud-starter-parent* müssen die Artefakte *spring-cloud-starter-eureka-server*, *spring-boot-starter-security* und *spring-cloud-starter-hystrix* eingebunden werden.

```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>1.0.1.RELEASE</version>
    <relativePath/>
</parent>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

Der Eureka-Server wird in der Datei *application.yml* konfiguriert:

```
server:
  port: 8761
security:
  basic:
    enabled: false

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

Der erste Schritt definiert, dass der Eureka-Server unter dem Port 8761 läuft. Der zweite schaltet einige Optionen aus und vereinfacht die Beispielkonfiguration: *registerWithEureka: false* und *fetchRegistry: false* deaktivieren zum Beispiel die Möglichkeit zur Replikation zwischen

mehreren Eureka-Servern im Standalone-Modus.

Nach dem Start gibt Eureka den Clients ein Zeitfenster von fünf Minuten für die Registrierung; beim Beispiel wird die Wartezeit mit *waitTimeInMsWhenSyncEmpty: 0* auf "Null" gesetzt. Der Eureka-Server ist standardmäßig mit einem Schutz versehen, sodass nur berechtigte Instanzen auf ihn zugreifen können. Für die Beispielkonfiguration lässt sich der Schutzmechanismus mit *security: basic: enabled: false* ausschalten.

Der Service braucht für den Start noch eine Main-Methode; sie wird in der Klasse *EurekaServerApplication.java* angeboten. Durch die Annotation *@EnableEurekaServer* erkennt Spring Cloud, dass es sich um einen Eureka-Server handelt, und startet den entsprechenden Dienst.

```
package com.maibornwolff.server;

import org.springframework.boot.SpringApplication;
import org.springframework.cloud.client.SpringCloudApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

import java.io.IOException;

@SpringCloudApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

Jetzt funktioniert der Eureka-Server, an dem sich ein Service registrieren kann. Als nächsten Schritt bindet man den Eureka-Client im HELLOWORLD-SERVICE ein.

Den HelloWorld-Service erstellen

Auch der HelloWorld-Service ist ein Spring-Boot-Service, der anhand spezieller Annotationen einen Eureka-Client enthält. Für den Service benötigt man neben der Referenz auf das Parent-POM *spring-cloud-starter-parent* die Artefakte *spring-cloud-starter-eureka* und *spring-boot-starter-web*. Letzteres ist notwendig, um einen REST-Service zu erstellen.

```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>1.0.1.RELEASE</version>
    <relativePath/>
</parent>

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

</dependencies>
```

In der Datei *application.yml* werden zwei Informationen festgelegt: die URL des Eureka-Servers und der Port, unter dem der Hello-World-Service verfügbar ist.

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

server:
    port: 8888
```

Der Applikationsname, den man in der Datei *bootstrap.yml* angibt, ist zugleich der logische Name des Microservices – in diesem Beispiel *helloWorld-service*:

```
spring:
    application:
        name: helloWorld-service
```

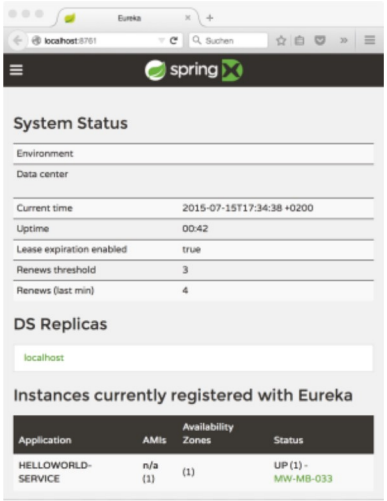
Der nächste Schritt implementiert den REST-Service. Unter der Ressource *hello* gibt der Service *"Hello cloudy world"* als Antwort zurück.

```
HelloController.java

package com.maibornwolff.client;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "Hello cloudy world";
    }

}
```

Damit der REST-Service unter seinem logischen Applikationsnamen registriert wird, muss man neben den Spring-Boot-Annotationen *@SpringBootApplication* und *@EnableAutoConfiguration* die Annotation *@EnableEurekaClient* angeben.

```
HelloWorldServiceApplication.java

package com.maibornwolff.client;

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

@SpringBootApplication
@EnableAutoConfiguration
@EnableEurekaClient
public class HelloWorldServiceApplication {

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

}
```

Nach dem Start registriert sich der Service am Eureka-Server. Das lässt sich in der Web-GUI des Eureka-Servers (*http://localhost:8761*) nachvollziehen. Unter der Liste der registrierten Instanzen ist der HELLOWORLD-SERVICE jetzt sichtbar (s. Abb. 2).



Nach erfolgreicher Registrierung erscheint der Hello-World-Service in der Web-GUI des Eureka-Servers (Abb. 2).

Den Service über einen Discovery-Client nutzen

Entwickler können den registrierten Service zum Beispiel in einer Kommandozeilen-Applikation nutzen, hier mit dem Namen Discovery-Client. Die Applikation ruft den HELLOWORLD-SERVICE anhand seines logischen Namens auf und gibt das Ergebnis auf der Kommandozeile aus. Dafür

ist der Discovery-Client auch als Eureka-Client zu annotieren. Er benötigt somit die folgenden Dependencies.

```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>1.0.1.RELEASE</version>
    <relativePath/>
</parent>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Für die Implementierung des Discovery-Clients benötigen Entwickler die Spring-Boot-Annotation *@SpringBootApplication* und die Spring-Cloud-Annotation *@EnableEurekaClient*. Damit der SpringApplicationBuilder weiß, dass es sich um eine Kommandozeilen-Applikation handelt, gibt man Folgendes an:

```
new SpringApplicationBuilder(DiscoveryClientApplication.class)
                .web(false)
                .run(args);
```

Dadurch wird die *run*-Methode der Klasse *RestTemplateExample* beim Start aufgerufen. Sie implementiert den *CommandLineRunner*. Die ganze Magie des Service-Bindings erfolgt im Spring RestTemplate. Hier benutzt man in der URL den logischen Namen "helloWorld-service" statt der tatsächlichen URL "localhost:8888".

```
DiscoveryClientApplication.java

package com.maibornwolff.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.feign.EnableFeignClients;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableEurekaClient
public class DiscoveryClientApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(DiscoveryClientApplication.class)
                .web(false)
                .run(args);
    }
}

@Component
class RestTemplateExample implements CommandLineRunner {

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public void run(String... strings) throws Exception {
        // use the "smart" Eureka-aware RestTemplate
        String exchange =
                this.restTemplate.getForObject(
                        "http://helloWorld-service/hello",
                       String.class);

        System.out.println(exchange);
    }

}
```

Test: Hello cloudy world

Beim Start des Discovery-Clients fragt der Eureka-Client die Service-Registry des Eureka-Servers ab. Vor dem tatsächlichen REST-Aufruf überprüft der Client, ob unter dem logischen Namen ein registrierter Service existiert, und ersetzt ihn durch den physikalischen Endpunkt, mit dem wiederum der REST-Aufruf durchgeführt wird.

Die Ausgabe der Konsole lautet, wie bei der Implementierung des REST-Services definiert, *Hello cloudy world*. Das zeigt, dass die Service-Registrierung funktioniert und die Services unter ihrem logischen Namen gefunden werden.

Als kleine Erweiterung lässt sich durch Anpassung des Ports eine weitere Instanz des Hello-World-Services starten. Der Eureka-Client sorgt nun für ein **Round Robin Load Balancing [7]** zwischen den beiden Services. Beendet man einen der Dienste, erkennt der Eureka-Server das und leitet die Anfragen nur noch an den aktiven Service weiter.

Fazit

Spring Cloud Netflix bietet viele qualitativ hochwertige Komponenten zur Entwicklung zum Betrieb einer Microservice-Architektur. Leider hält die Dokumentation nicht immer Schritt mit den schnellen Releasezyklen im Projekt. Viele Konfigurationsoptionen sind nicht dokumentiert und nur durch Analyse des Quellcodes zu entdecken. Die Spring-Cloud-Contributer definieren folgerichtig die Dokumentation als wichtiges Ziel auf der Roadmap für die nächsten Versionen.

Dieser Artikel leistet einen kleinen Beitrag zu diesem Ziel: Wie gezeigt, lässt sich mit der Eureka-Komponente von Spring Cloud Netflix in wenigen Minuten eine funktionierende Service-Registry für Microservices aufbauen. (ane)

Stefan Janser
arbeitet als Senior Software Engineer bei MaibornWolff. Der Diplom-Informatiker beschäftigt sich mit der Entwicklung und Integration großer verteilter Systeme mit Cloud- und Webtechniken.

URL dieses Artikels:
https://www.heise.de/developer/artikel/Eureka-Microservice-Registry-mit-Spring-Cloud-2848238.html

Links in diesem Artikel:

[1] http://variety.com/2015/digital/news/netflix-bandwidth-usage-internet-traffic-1201507187/
[2] https://netflix.github.io/
[3] http://cloud.spring.io/spring-cloud-netflix/
[4] http://projects.spring.io/spring-cloud/
[5] http://martinfowler.com/bliki/CircuitBreaker.html
[6] http://spring.io/guides/gs/spring-boot/
[7] https://de.wikipedia.org/wiki/Round_Robin_(Informatik)