Inhaltsverzeichnis

1 Abstract

Zu Umsetzung von Cloud Nativen Anwendungen liegt Spring Boot aktuell vor Java EE. Für die Integration von Spring Boot Anwendungen in Cloud Umgebungen unterstütze das auf Spring basierende Spring Cloud Projekt. Die folgende Arbeit gibt zu Beginn einen Überblick über die von Spring Cloud beschriebenen Komponenten und dessen Implementierungen. Anschließend wird das Projekt `Spring Cloud Netflix` und dessen Implementierung von `Service Registration and Discovery`, namens Eureka, genauer beleuchtet. Dazu wird eine Simple Beispiel Implementierung gegeben welche den Aufbau, Ablauf und Kommunikation eines `Service Registration and Discovery` Netzes darstellt und erklärt. Anschließend werden dessen Stärken und Schwächen gegenübergestellt. Zum Ende wird ein kurzes Fazit zu Spring Cloud gegeben.

2 Spring Cloud

2.1 Einführung

Spring Boot ist einer der am weit verbreitetsten und meist verwendeten Java Frameworks um Microservices zu Entwickeln. Zusätzlich setzen viele Firmen heutzutage auf Cloud Infrastrukturen um sich nicht mit dem Betrieb und Wartung eigener Server auseinandersetzen zu müssen. Hierbei müssen die Anwendungen beliebig skalierbar sein, elastisch auf Last reagieren und dabei dennoch bis zum Ende Fehlertolerant sein. Hier kommt nun Spring Cloud zur Hilfe.

Das Spring Cloud Projekt ist ein Projekt des Spring Teams das eine Reihe von verschiedenen Mustern Verteilter Systeme als einfache Java Bibliothek bereitstellt. Spring Cloud ist dabei nur ein Development Kit und keine Cloud Lösung. Es stellt einem eine Vielzahl an Essentiellen Bibliotheken zur Entwicklung von Cloud Applikationen bereit [6, S. 205]. Da Spring Cloud auf das Spring Framework aufbaut und es zum größten Teil die Art und Weise der Entwicklung von Applikationen beibehält gestaltet sich die Entwicklung von Spring Cloud Anwendungen ähnlich einfach wie die Entwicklung von Spring/Spring Boot Anwendungen. Viele Dinge lassen sich meist schon mit einzelnen Java Annotationen lösen welche Spring Typisch viel im Hintergrund bewirken. Am Ende lassen sich Spring Cloud Anwendungen in gewohnter Spring Weise schnell und effizient entwickeln.

2.2 Komponenten

Spring Cloud lässt sich grob in 12 Komponenten unterteilen welche jeweils eine bestimmte Aktion/Eigenschaft eines Verteilten Systems adressieren.

Distributed configuration Gute Konfiguration ist sehr wichtig um sein Verteiltes System unter Kontrolle zu halten. Mit einer Dezentralen Konfiguration wie Manuelles Konfigurieren jedes einzelnen Microservices würde das komplette Gegenteil von Kontrolle erzeugen geschweige denn bei sehr großen Systemen viel zu viel Aufwand mit sich bringen. Am besten wäre es die Konfiguration von einer Stelle aus Verwalten zu können. Somit behält man zu jeder Zeit sein System, ob klein oder groß, unter Kontrolle. Die Komponente `Distributed configuration` nimmt sich genau diesem Thema an.

Service registration and discovery Microservices die nichts voneinander wissen sind nutzlos erst recht wenn diese miteinander interagieren sollen. In der immer populäreren werdenden Welt der Container Orchestration von Kubernetes und Docker bleiben die Adressen der jeweiligen Anwendungen zu keiner Zeit statisch sondern ändern sich ständig. Ein Beispiel wäre ein von Kubernetes gestarteter Container welcher nicht zwingend immer die gleiche Adresse (hostname:port) zugeordnet bekommt. Somit ist zu keiner Zeit garantiert das eine angefragte Adresse auch der gewollte Microservice ist. `Service registration and discovery` adressiert dieses Problem.

Distributed messaging Bereitstellung von zuverlässigen messaging Systemen wie Kafak, RabbitMQ, Redis und weitere [6, S. 209].

Routing Eine Anwendung besteht meist aus vielen Teilen wie z.b. eine Web Oberfläche, ein Anmelde Service, einen Shop und so weiter. Nun müssen alle eingehenden Anfragen auf diese Teile der Anwendung richtig verteilt werden. Dies kann mittels eines sogenanntem Router erfolgen welcher z.b. die Anfrage `/` auf die Web Oberfläche weiterleitet oder `/user/login` auf den Anmelde Service weiterleitet.

Loadbalancing Um nicht dauerhaft alle eingehenden Anfragen, für einen speziellen Service, in eine Warteschlange einzureihen ist es sinnvoll die Last auf mehrere Replikate, eines Services, zu verteilen um eine schnellere Verarbeitungszeit zu erzielen. Meist wird dies auch vom Container Orchestrator übernommen dennoch bietet Spring Cloud hier eine eigene Lösung an um eine Lastverteilung auch in nicht Orchistrierten Systemen zu verwenden. Auch eine Client-seitige Lastverteilung ist hier möglich.

Service to Service calls Adressiert die Unterhaltung von zwei oder mehr Services untereinander um einen Datenaustausch/Anfrage zwischen ihnen zu ermöglichen.

Security Eine Sicherheits Komponente darf in der heutigen Zeit natürlich nicht fehlen und wird mit der Security Komponente in Spring Cloud adressiert. Unter diese Komponente fallen dann Sachen wie das OAuth2 Protokoll wo z.b. der Authentifizierungs Server in seinem eigenem Service läuft, aber auch das erstellen von Mustern wie `single sign on`, `token relay` und `token exchange` wird hier spezifiziert [5].

Global locks Leadership Election and cluster state Wird zum Cluster Management und Koordinierung großer Systeme verwendet. Es erlaubt auch Globale Locks um z.b. eine Sequenz Generierung durchzuführen [6, S. 209].

Cloud Support In der heutigen Zeit gibt es eine Vielzahl an bereits existierenden Cloud Providern die einem Ihre hochverfügbaren Cloud Infrastrukturen bereitstellen und sich dabei um Betrieb dieser kümmern. `Cloud Support` will es vereinfachen Anwendungen auf existierenden Cloud Infrastrukturen wie AWS, Azure und Co. zu deployen und dessen Mechanismen als Schnittstelle anzubieten.

Big Data Support Unterstützung von `Big Data` Lösungen um Data Services und Data flows zu ermöglichen.

Distributed Tracing Interessant für DevOps Teams welche ihre verteilten Systeme Überwachen/Debuggen wollen. Hierbei ist es möglich in einem verteiltem System den Fehler oder eventuelle Performance Einbußen zurück zum Ursprung zu verfolgen um diese anschließend am richtigen Punkt beheben zu können.

Circuit Breakers Adressiert die Implementierung des `circuit breaker` Musters welches z.b. beim auftreten von mehreren Fehlern bei einem speziellen Service die Verbindung zu diesem abbricht und direkt einen Fehler zurück gibt oder auf einen Backup Service weiterleitet um weitere Überlastung zu vermeiden.

2.3 Haupt Projekte

Die Haupt Projekte sind Projekte unter dem Schirm von Spring Cloud die eine der oben genannten Komponenten implementieren. Die Grauen Zellen in unten stehender Tabelle 2.1 stellen die Komponenten da und die in der Zeile dazu befindlichen Zellen die Projekte die jeweils diese Komponente implementieren.

Distributed Configuration	Config	Spring Cloud Bus	Netflix Archaius
Service Registration and Discovery	Netflix Eureka	Consul	Netflix ZooKeeper
Distributed messageging	Streams		
Routing	Netflix Zuul		
Load Balancing	Netflix Ribbon		
Service to Service Calls	Netflix Feign		
Security	Security		
Global locks Leadership Election and cluster state	Cluster		
Cloud Support	Connectors	AWS	Cloud Foundry
Bid Data Support	Data Flow		
Distributed Tracing	Sleuth		
Circuit Breakers	Netflix Hysterix		

Tabelle 2.1: Spring Cloud Main Projects [6, S. 207]

Wie in der Tabelle 2.1 oben festzustellen ist stellt der Streaming Konzern Netflix Implementierungen für den Großteil der Komponenten bereit.
Weitere Spring Cloud Projekte die in die Tabelle 2.1 nicht direkt eingeordnet werden können da diese meist Spezielle Muster zur Verfügung stellen, Verbindung zu bestimmten Systemen herstellen oder Spring Cloud Komponenten unterstützen.

- Spring Cloud Open Service Broker
- Spring Cloud Stream App Starters
- Spring Cloud Task
- Spring Cloud Task App Starters
- Spring Cloud Starters
- Spring Cloud CLI
- Spring Cloud Contract
- Spring Cloud Gateway
- Spring Cloud OpenFeign
- Spring Cloud Pipelines
- Spring Cloud Function

3 Spring Cloud Netflix

Viele der in Spring Cloud definierten Komponenten findet man im Netflix OSS (Open Source Software) center wieder. Da Netflix zu den ersten gehörte die im Microservice Umfeld aktiv wurden ist die Bibliothek dementsprechend Fortgeschritten [6]. Etwas mehr als die Hälfte der in Tabelle 2.1 erwähnten Implementierungen von Netflix sind im Wartungsmodus, somit werden keinen neuen Features mehr hinzugefügt sondern alleine bestehende Fehler ausgebessert [3]. Nachfolgend wir einer der Haupt Komponenten im Spring Cloud Netflix Projekt beleuchtet.

3.1 Service Discovery - Eureka

Der `Registration and Discovery Service` von Netflix ist ein auf REST (Representational State Transfer) basierender Service Namens **Eureka**. Eureka wird von Netflix hauptsächlich in AWS (Amazone Web Services) verwendet um Services zu lokalisieren [2]. Unterteilt ist es dabei in den **Eureka Server** welcher Registrierungen annimmt und Auskunft über registrierter gibt und denn **Eureka Client** welcher von jedem Services benötigt wird der Teil der Lokalisierung sein will. Zusätzlich beinhaltet der Eureka Client noch ein auf `Round Robin` basierende Lastverteilung [2].

Ablauf Alle Eureka Client Services melden sich zu Beginn bei dem Eureka Server an. Jeder Client meldet sich einmal alle 30 Sekunden um dem Eureka Server seine Anwesenheit mitzuteilen und um eine Aktualisierte Liste aller im Eureka Netz aktiven Services zu bekommen. Der Ansatz des erhalten einer Liste aller aktiven Services wurde gewählt um Schnellere Reaktionszeiten zu erhalten und um hohe Last auf denn Eureka Servern zu vermeiden.

Peer Awareness Eureka erlaubt zusätzliche Instanzen welche sich gegenseitig registrieren dies nennt man `Peer Awareness`. Damit alle Instanzen auch über jede Registrierung Bescheid wissen müssen diese lediglich über eine Ecke miteinander verbunden sein damit die Synchronisierung klappt [4, Peer Awareness]. Zu beachten ist das ein großes Netz solcher Discovery Services eine gewisse Zeit benötigt bis eine neue Information bis an das andere Ende des Netzes wandert. Pro zusätzlichen Eureka Server auf dem Weg benötigt die Information 30 Sekunden mehr um zum anderem Ende auf direktem Weg zu gelangen.

3.2 Beispiel

Nachfolgend wird ein kurzes Beispiel gegeben was ein kleines Netz an Microservices darstellt die über einen Discovery Service verbunden sind. Das Beispiel beinhaltet 3 Services einen **Discovery-Service** für die Lokalisierung der Services, einen **HelloWorld-Service** welcher auf eine GET Anfrage antwortet und einen **Client-Service** welcher beim HelloWorld-Service anfragt ohne dessen Adresse beim Start statisch mitgeteilt zu bekommen.

Hierbei kommt von Netflix der Eureka Server für den Disocvery Service bzw. Eureka Client für die beiden anderen Services zum Einsatz. Die Abbildung 3.1 zeigt den groben Aufbau der drei Services. Hierbei stellen die Grauen Rechtecke jeweils einen Service da. In Jedem dieser Services sind in hellerem Grau die wichtigsten Komponenten eines jedem Service dargestellt. Die möglichen Aktionen zwischen den Services ist mittels Pfeilen und dessen damit verbundene Aktion dargestellt.

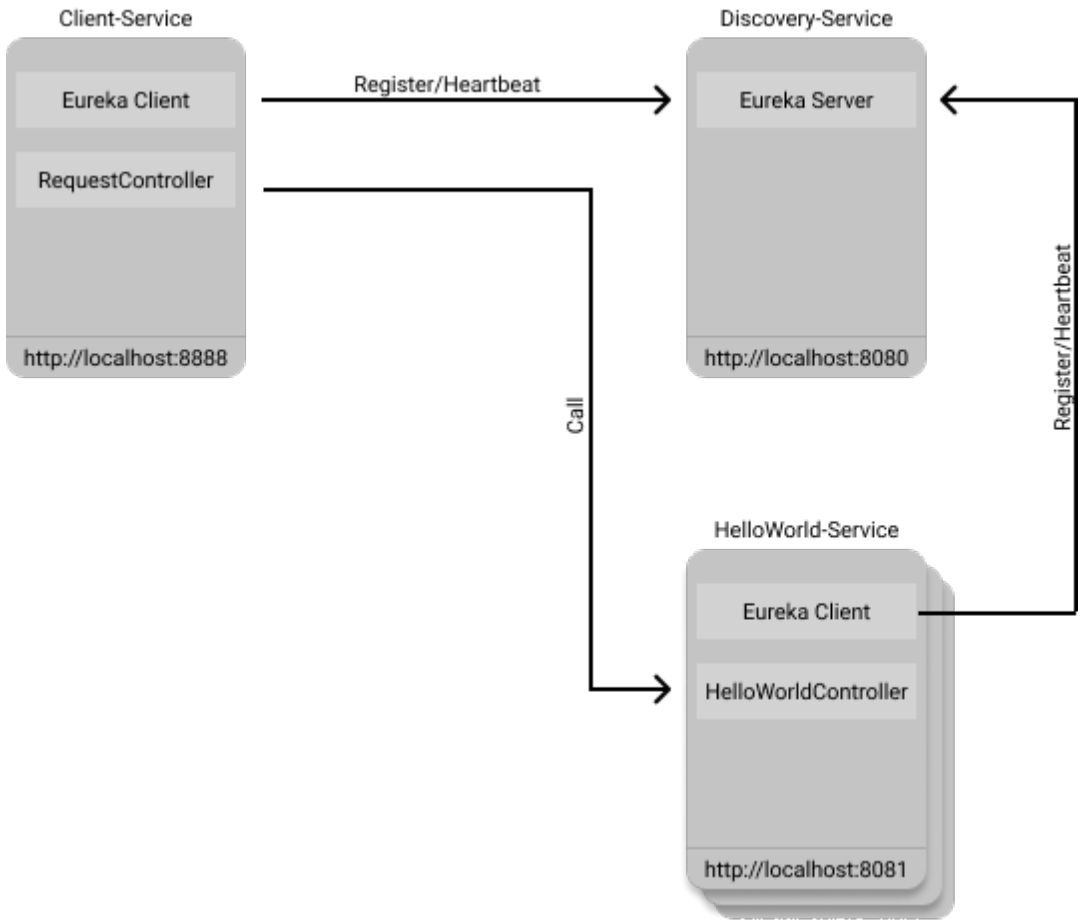

Abbildung 3.1: Aufbau und Verbindung der Beispiel Services

3.2.1 Aufbau

Nachfolgend wird erklärt wie man jeden der drei Services einrichtet. Das Beispiel wurde dabei auf einem MacBook Pro 2017 (macOS Mojave Version 10.14.4) mit 3.1 GHz und 8 GB Ram entwickelt und ausgeführt. Zum Kompilieren wurde das Java 8 JDK 8u211 verwendet. Zur Vereinfachung wurde stets der `spring initializr` unter `https://start.spring.io/` verwendet um benötigte Abhängigkeiten hinzuzufügen. Selbstverständlich ist auch ein manuelles hinzufügen der Abhängigkeiten möglich.

Discovery-Service einrichten Um den Eureka Clients eine Registrierung zu ermöglichen muss die Abhängigkeit des **Eureka Server** in den Discovery Service eingebunden werden. Der folgende Ausschnitt der `build.gradle` Datei zeigt die benötigte Abhängigkeit.

```
...
dependencies {
    implementation
        'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
    ...
}
...
```

Als nächstes wird die Hauptklasse des Discovery-Services erstellt:

```
@SpringCloudApplication
@EnableEurekaServer
public class DiscoveryApplication {

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

}
```

Die **@EnableEurekaServer** Java Annotation ist das einzigste was man für einen Simplen aber dennoch voll funktionsfähigen `Service Registration & Discovery Service` benötigt. Natürlich wird Spring Typisch im Hintergrund sehr viel Konfiguriert welches man über die `application.yml` überschreiben kann und was im Nachfolgendem Auszug auch geschieht.

```
spring.application.name: 'discovery'
server.port: 8080

eureka:
  instance.hostname: 'localhost'
  client:
    fetch-registry: false
    register-with-eureka: false
    serviceUrl.defaultZone:
        'http://${eureka.instance.hostname}:${server.port}/eureka/'
```

Die `defaultZone` Variable definiert auf welchem Endpunkt der Eureka Server Anfragen entgegen nehmen soll. Mit den folgenden Variablen `register-with-eureka` und `fetch-registry` wird noch die Möglichkeit der Replikation zwischen mehreren Eureka Instanzen abgeschaltet.

HelloWorld-Service einrichten Nachdem der Service zur Registrierung und Findung steht benötigt man noch Services die darüber erreichbar sein sollen. Um es so einfach wie möglich zu halten wird ein `HelloWorld-Service` erstellt welcher sich beim `Discovery` Service anmeldet und dann auf eingehende REST Anfragen antwortet. Hierzu wird der `eureka-client` hinzugefügt. Der folgende `build.gradle` Ausschnitt zeigt die für den `HelloWorld-Service` benötigten Abhängigkeiten.

```
...
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation
        'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    ...
}
...
```

Anschließend wird in der Hauptklasse eine Annotation hinzugefügt die der Anwendung mitteilt das sie sich bei einem `Eureka Server` anmelden muss.

```
@EnableEurekaClient
...
```

Damit der Service auch weiß wo er sich zu registrieren hat wird in der `application.yml` die Eigenschaft `defaultZone` mit der für den Discovery-Service entsprechenden URL gesetzt.

```
eureka.client.serviceUrl.defaultZone: 'http://localhost:8080/eureka/'
server.port: 8081
```

Bei der Anmeldung muss der, sich Anmeldende, Service seinen Applikations Namen angeben welcher zugleich der logische Name des Services ist über die er gefunden werden kann. Dieser wird in der `bootstrap.yml` Datei angegeben.

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

Nun muss der Service noch eine Funktionalität bieten. Um es einfach zu halten antwortet der Service mit `Hello World! from <ip:port>` wenn er eine GET Anfrage auf `/` bekommt. Dies wird wie in Spring Boot mit einem Restcontroller realisiert.

```
@RestController
public class HelloWorldController {

    @Value("${server.port}")
    Integer port;

    @GetMapping
    public String helloWorld() {
        try {
            String ip = InetAddress.getLocalHost().getHostAddress();
            return "Hello World! from " + ip + ":" + port.toString();
        } catch (Exception ignored) {
            return "Error";
        }
    }
}
```

Client-Service einrichten Nun benötigt man noch einen Client der die Anfragen stellt. Dieser muss sich wie bereits der HelloWorld-Service beim `Eureka Server` anmelden. Somit benötigt er, wie der HelloWorld-Service, die Abhängigkeit des `Eureka Clients` damit in der Hauptklasse die Annotation `@EnableEurekaClient` verwendet werden kann welche die in der `application.yml` definierten Einstellungen verwendet um sich beim `Eureka Server` anmelden zu können. Für die Anmeldung mit Name wird wie im `HelloWorld-Service` die `bootstrap.yml` Datei angelegt welcher den Namen des Clients beinhaltet. Im Prinzip ist oben genanntes Analog zu der Einrichtung des `HelloWorld-Services` mit Ausnahme des RestControllers.

Um nun mit dem `HelloWorld-Services` zu sprechen wird bei einer Simplen GET Anfrage auf den Klienten ein Request mit dem im `bootstrap.yml` definierten Anwendungs Namen des `HelloWorld-Services` ausgeführt. Folgender Code Auszug zeigt dies.

```java
@RestController
public class RequestController {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping
    public String normal() {
        return restTemplate.getForObject("http://helloWorld-service/",
            String.class);
    }

}
```

3.2.2 Ausführung

Nachdem alles Konfiguriert wurde startet man zuerst denn Discovery Service da sich die anderen beiden Services dort zuerst Registrieren müssen. Ein Erfolgreicher Start des Discovery Services beinhaltet folgende Log Zeilen:

```
Setting initial instance status as: STARTING
Initializing Eureka in region <region>
Setting the eureka configuration..
Started Eureka Server
Started DiscoveryApplication in 3.707 seconds (JVM running for 4.432)
```

Anschließend kann man denn HelloWorld-Service und den Client Service unbeachtet der Reihenfolge starten. Der Eureka Server gibt den Clients ein 5 Minuten Zeitfenster [1] um sich zu Registrieren anderseits fährt dieser automatisch wieder herunter.

```
Getting all instance registry info from the eureka server
Starting heartbeat executor: renew interval is: 30
Registering application HELLOWORLD-SERVICE with eureka with status UP
Started HelloWorldServiceApplication in 3.29 seconds (JVM running for 4.108)
DiscoveryClient_HELLOWORLD-SERVICE/<address>:8081 - registration status: 204
```

Der Log Ausschnitt des HelloWorld-Services zeigt dass die Registrierung beim Discovery Service erfolgreich war. Der HelloWorld-Service wird nun alle 30 Sekunden einen Heartbeat gegen den Discovery Service ausführen womit er einerseits dem Discovery Service sein `UP` Status mitteilt und anderseits erhält er Informationen über neu bzw. tote Services welche er dann Lokal zum auflösen von Anfragen verwendet.
Eine Übersicht über die aktiv verbunden Services lässt sich über die Eureka Weboberfläche abrufen, wie in folgender Abbildung 3.2 zu sehen ist.

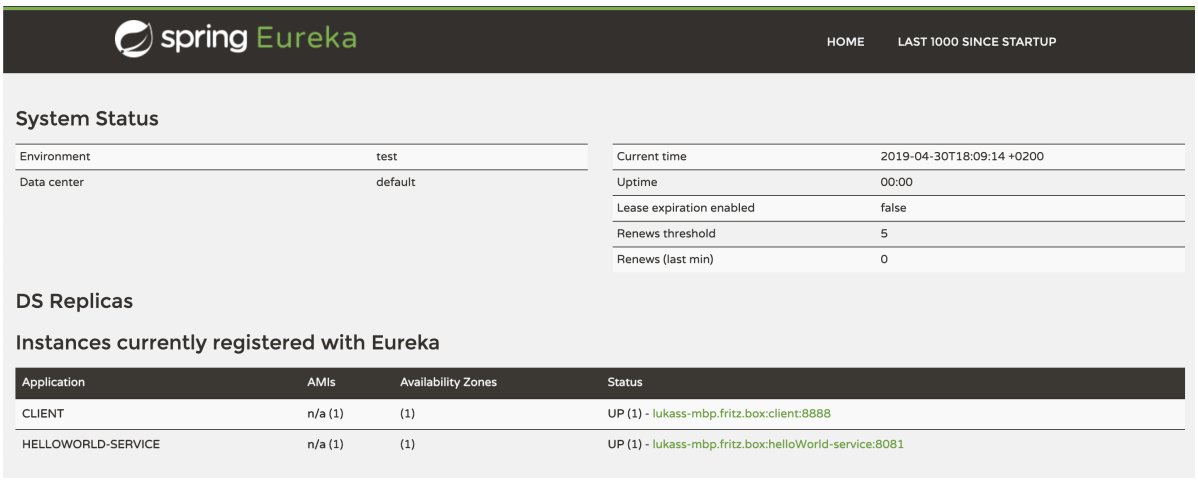

Abbildung 3.2: Weboberfläche des Eureka Servers

Öffnet man nun mit Browser die Adresse des Client Services führt dieser die Anfrage gegen den HelloWorld-Service aus. Die echte Adresse des HelloWorld-Service erhielt der Client-Service bei der Anmeldung bzw. nach einem von ihm ausgeführtem Heartbeat.

Abmelden Jeder Eureka Client Service führt bei einem normalen Shutdown eine Abmeldung beim Discovery Service durch, welche wie folgt anhand eines Logs zusehen ist:

```
Unregistering ...
DiscoveryClient_CLIENT/<address>:8888 - deregister status: 200
Completed shut down of DiscoveryClient_CLIENT
```

Die Abmeldung wird vom Discovery-Service Registriert und der Status wird auf `DOWN` gesetzt:

```
Registered instance CLIENT/<address>:8888 with status DOWN (replication=false)
Cancelled instance CLIENT/<address>:8888 (replication=false)
```

Alle anderen Services, die noch Aktiv sind, bekommen die Information über einen Service der den Status `DOWN` bekommen hat beim nächsten Heartbeat mitgeteilt. Und so läuft grundsätzlich eine Service Registrierung und Findung innerhalb eines mit Eureka verbundenen Systems ab.

3.3 Stärken & Schwächen

3.3.1 Stärken

Zu den Stärken von Netflix Eureka, aber auch allgemein Spring Cloud, zählt die leicht und Spring typische Art der Programmierung von Anwendungen. Viele Dinge lassen sich meist mittels einzelner Annotationen beziehungsweise leichten Änderungen in den properties realisieren. Zusätzlich bleibt die ganze Programmierung innerhalb der Java Welt was das allgemeine Entwickeln von verteilten Java Anwendungen für Entwickler deutlich vereinfacht aufgrund der gewohnten und gleichbleibenden Art. Dies wird auch durch die gute Integration in Spring Unterstützt.

3.3.2 Schwächen

Zu den Schwächen von Eureka zählt vor allem, dass die Service Registrierung beziehungsweise Findung von einem `DNS-Based Service Discovery (DNS-SD)` oder einem Orchestrierten System wie `Kubernetes` übernommen werden kann und die Unabhängig von jeweiligen Frameworks oder Programmiersprachen handeln. Die Limitierung auf die Java Welt kann zugleich Vorteil und auch Nachteil sein im Prinzip Limitiert man sich hier auf eine Sprache für alle Microservices, somit ist ein einfacher Austausch eines Microservices durch einen in einer anderen Sprache geschriebenen Microservice nicht möglich insofern man für den Eureka Service keine Implementierung für andere Sprachen findet oder selbst Entwickelt. Bei größeren Systemen ergeben sich auch Probleme mit der Findung über das gesamte System. Da ein Eureka Server nur alle 30 Sekunden sich ein Update von einem benachbarten Eureka Server holt. Im Prinzip kommen, pro dazwischen geschalteten Eureka Server, 30 Sekunden drauf um von einem Ende bis zum Anderem Ende, auf kürzester Route, des Systems die aktuellen Information durch zugeben welche bei Ankunft schon wieder veraltet sind. Somit ist ein `peer awareness` System niemals an allen Ecken des Eureka Netzes auf dem Aktuellsten Stand insofern sie nicht alle direkt untereinander verbunden sind.

4 Fazit & Ausblick

Die Vernetzung von Services lässt sich auf viele Arten lösen. Die `Service Registration and Discovery` Implementierung `Eureka` ist dabei eine von vielen Möglichkeiten. Eureka lässt dem Entwickler, in einer gleichbleibenden Umgebung, alle Nötigen Einstellung für ein Verteiltes System treffen das sich später selbst finden soll. Spring Boot Entwicklern wird die Art der Programmierung sofort bekannt vorkommen was ihnen somit eine schnellere Entwicklung eines verteilten Systems ermöglicht. Auch die gute Integration in Spring ermöglicht es aktuelle Projekte leichter an Cloud Strukturen anzupassen.

Ein großer Nachteil an Eureka jedoch ist das andere Methoden bzw. Plattformen, wie DNS-SD oder Kubernetes, die gleiche Funktionalität Vielseitiger und Programmiersprachen Unabhängig bereitstellen können.

Schlagwörter wie Microservices, Cloud-Native und weitere prägen aktuell die Entwickler/DevOps Landschaft. Der Betrieb von immer größer werdenden Cloud Systemen mit der Microservices Architektur wird in denn kommenden Jahren vermutlich immer größer werden womit auch die Notwendigkeit an Discovery Services steigt um die Systeme zusammenhalten zu können.

Literaturverzeichnis

[1] Stefan Janser. *Eureka – Microservice-Registry mit Spring Cloud*. URL: https://www.heise.de/developer/artikel/Eureka-Microservice-Registry-mit-Spring-Cloud-2848238.html?seite=all (besucht am 04. 04. 2019).

[2] Netflix. *Eureka at a glance*. URL: https://github.com/%20Netflix/eureka/wiki/Eureka-at-a-glance (besucht am 07. 05. 2019).

[3] Netflix. *Spring Cloud Netflix Github Repository*. URL: https://github.com/spring-cloud/spring-cloud-netflix (besucht am 06. 05. 2019).

[4] Pivotal Software. *Spring Cloud Netflix*. URL: https://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html (besucht am 10. 04. 2019).

[5] Pivotal Software. *Spring Cloud Security*. URL: https://spring.io/projects/spring-cloud-security.

[6] Rajesh R V. *Spring Microservices*. Birmingham: Packt Publishing Ltd., 2016.