Hysterie in verteilten Systemen – Hystrix im Einsatz

Benjamin Wilms, codecentric AG

Ob man sich nun in einer modernen Microservice-Architektur bewegt oder an bestehende Legacy-Backends binden muss, in beiden Fällen ist mit dem Verhalten und den Fehlern dieser Systeme zu leben und damit umzugehen. Das Hystrix-Framework erfreut sich immer größerer Beliebtheit. Service-Aufrufe werden isoliert, um die eigene Anwendung vor den Fehlern ihrer Abhängigkeiten zu schützen. Timeouts und Exceptions, die zum Absturz der eigenen Anwendung führen können, werden verhindert.

Das Thema „Resilient Software" findet immer mehr Aufmerksamkeit und wird dabei von mehreren Global Playern wie zum Beispiel Netflix stetig vorangetrieben. Der Artikel geht unter anderem der Frage nach, was eine resiliente Software ist und wie man diese erreichen kann.

Eines der obersten Ziele ist es, eine fehlerfreie und vor allem immer korrekt arbeitende Applikation zu entwickeln. Der Anwender sollte von technischen Fehlern nichts mitbekommen, im schlimmsten Fall kann ein sehr aufmerksamer Anwender dies nur an einem langsam reduzierten Service-Level merken. Dabei spricht man von „graceful degradation", das System hält den Betrieb so weit als möglich aufrecht und reduziert nur die Funktionalität. Dies hört sich im ersten Moment einfach an, bedeutet aber auch, dass technische Fehler Teil der Fachlichkeit werden, was eine starke Zusammenarbeit mit den Verantwortlichen für das fachliche Verhalten der Applikation erfordert. So sind Fragen nach möglichen Fallbacks, statischen Defaults und/oder finalen Fehlerseiten zu klären.

Um die genannte Stabilität zu erreichen, werden wir uns einiger Projekte aus dem Hause Netflix bedienen und anhand derer eine Beispielanwendung erstellen. Die in diesem Artikel beschriebene Anwendung ist auf GitHub unter *„https://github.com/MrBW/resilient-transport-service"* verfügbar und kann gerne für erste Erfahrungen mit Hystrix und Co. verwendet werden. Alle notwendigen Informationen zum Aufsetzen der Demo-Anwendung sind dort dokumentiert.

Transport my Package

Die genannte Beispiel-Anwendung ist in der Lage, einen Transportauftrag entgegenzunehmen und einen sogenannten „Booking-Request" zu verarbeiten. Anhand einer Kundennummer wird ermittelt,

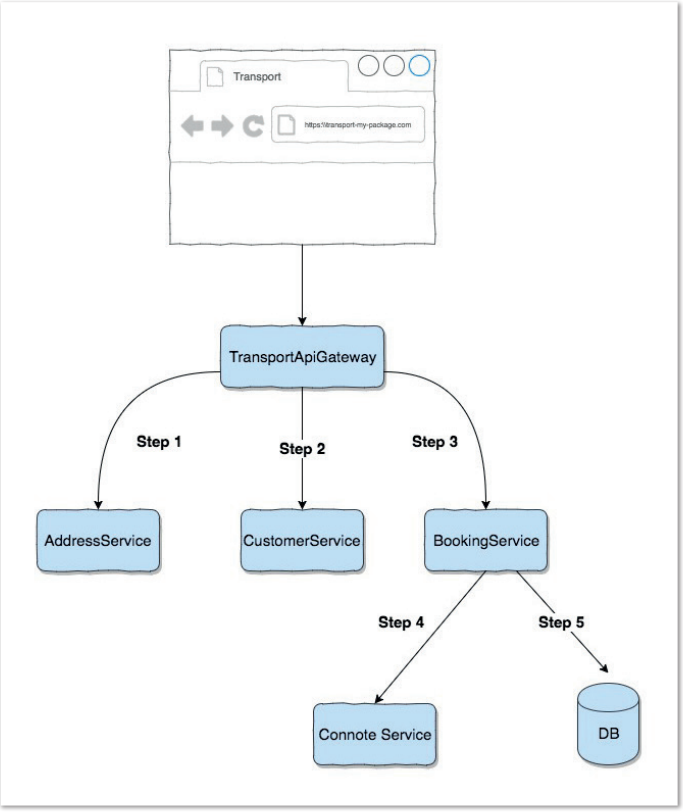

Abbildung 1: Architektur-Diagramm

ob es diesen Kunden gibt, ob die Abhol- und Zustelladresse valide ist und ob diese von unserem fiktiven Unternehmen „Transport my Package" angefahren werden kann. Im Anschluss an diese Schritte wird eine Frachtbriefnummer („Connote") generiert und eine finale Buchung ausgelöst *(siehe Abbildung 1)*.

Gehen wir zunächst die notwendigen Schritte eines Booking Request durch. Schritt 1 ist die Validierung der Abhol- und Zustelladresse und Schritt 2 die Überprüfung des angemeldeten Kunden. Beide Schritte werden parallel ausgeführt und liefern die benötigten Daten, um mit Schritt 3 fortzufahren. Nur bei einem Erfolg beider Schritte kann die eigentliche Buchung durchgeführt werden.

Im Schritt 3 übernimmt der Booking Service die zuvor in den Schritten 1 und 2 ausgelesenen und validierten Daten und bereitet eine Buchung vor. Schritt 4 ist die Erzeugung einer eindeutigen Frachtbriefnummer, unter der die Sendung geführt wird. Im Schritt 5 erfolgt das Fortschreiben der Buchung, wenn eine Frachtbriefnummer erzeugt werden konnte. Im Fehlerfall wird eine entsprechende Meldung zurückgegeben.

Architektur

Jede Komponente ist dabei eine schlanke Spring-Boot-Applikation, die in einem eigenen Docker-Container betrieben wird. Dadurch ist es möglich, die Skalierung als eine mögliche Fallback-Strategie bereits während der Entwicklung zu verwenden. Hystrix versetzt einen in die Lage, für jeden Service-Call einen entsprechenden Fallback bereitzustellen und den eigentlichen Service-Aufruf von Hystrix absichern zu lassen.

In diesem Zusammenhang wird auch immer gerne von „Bulkheads" (Schotts) gesprochen, in die man eine Applikation aufteilen kann, um kaskadierende Fehler zu verhindern. Diese Aufgabe übernimmt Hystrix indirekt, da man auf technische Fehler und definierte Timeouts reagieren kann. Somit sind diese mit Hystrix abgesicherten Bereiche isoliert, was die Stabilität des Gesamtsystems deutlich erhöht.

Reicht Hystrix für die Absicherung und das Definieren von Bulkheads aus? Die Antwort lautet ganz klar – nein! Bulkheads sind eines der elementarsten Resilience-Pattern, wenn nicht gar das elementarste.

Die meiste Zeit bei der Entwicklung eines neuen Systems wird dafür verwendet, die fachlichen Verantwortlichkeiten zu ermitteln und zu verstehen. Erst mit diesem Wissen ist es möglich, die einzelnen Bausteine (Module) so zu schneiden, dass keine langen Aufrufketten entstehen und die Module möglichst unabhängig voneinander arbeiten können. Nun ein Blick auf die beteiligten Akteure zur Absicherung der Beispielanwendung.

Hystrix

Hystrix wurde von Netflix entwickelt und als eines von etwa dreißig Open-Source-Projekten veröffentlicht; es erfährt in den letzten Jahren eine immer größere Beliebtheit. Wann und wofür man Hystrix verwenden soll, ist eine der ersten Fragen, die es zu beantworten gilt.

Hystrix kann mit Java, Java EE und Spring eingesetzt werden und wird als Dependency im Projekt eingebunden. Egal in welcher Galaxie des Java-Universums man sich aufhält, es gibt immer die drei Möglichkeiten, Hystrix im Code zu verankern.

Als erste Variante kommt das Command-Pattern *(siehe „https://github.com/Netflix/Hystrix/wiki/How-To-Use")* zum Einsatz, wobei man für jeden externen Service-Aufruf ein eigenes Command erstellt und dort seinen Service-Aufruf mit passendem Fallback bereitstellt. Die Fallbacks sind die hohe Kunst des Ganzen und müssen mit der Fachlichkeit zusammen entwickelt werden. Hier trifft die Fachlichkeit auf technische Fehler, die ihr vom Entwickler erklärt werden und die zu einem definierten Fallback führen müssen.

Ein Entwickler hat ebenso die Möglichkeit, Hystrix-Javanica *(siehe „https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica")* zu nutzen und somit anstatt mit einer eigenen Klasse mit einer einfachen „@HystrixCommand"-Annotation zu arbeiten. In der Spring-Galaxie gibt es, für Spring typisch, auch die Möglichkeit, mit einer einfachen „@HystrixCommand"-Annotation zu arbeiten. Spring baut dabei auf der Hystrix-Javanica-Implementierung auf. Zu erwähnen ist, dass man seine Fallback-Methode als einfachen String in der Annotation angibt und somit erst zur Compile-Zeit einen möglichen Tippfehler im Methodennamen erkennt.

Das Schöne an Hystrix ist, dass Netflix eine Unmenge an Konfigurationsmöglichkeiten vorgesehen hat und für jeden Parameter einen passenden Default bereitstellt. Somit ist es möglich, Hystrix aus dem Stand heraus zu nutzen und mit den Defaults erstmal loszulegen. In der Beispiel-Anwendung verwenden wir Hystrix immer dann, wenn wir uns an einen Remote-Service binden müssen und diesen nicht asynchron aufrufen können *(siehe Abbildung 2)*.

Archaius

Archaius stammt ebenfalls aus dem Hause Netflix und steht auch als

Open-Source-Projekt zur Verfügung. Hystrix baut intern auf Archaius auf und bietet damit die Möglichkeit, eine dynamische Konfiguration zur Laufzeit bereitzustellen. Dies ist ein klarer Vorteil bei den ersten Schritten mit Hystrix und versetzt einen in die Lage, Einfluss auf die definierten Timeouts zu nehmen oder den Circuit Breaker im laufenden Betrieb vom Status „geschlossen" auf „offen" zu setzen. Warum sollten wir das tun? In einer Integrationsumgebung können wir so unsere Fallback-Strategie überprüfen und auf ihre Standfestigkeit hin testen. Ebenso können wir dafür sorgen, dass unsere Fallbacks von Hystrix nicht mehr im Fehlerfall verwendet werden, und testen, ob unser Gesamtsystem damit umgehen kann. Keiner möchte in einer modernen, verteilten Servicelandschaft dreißig oder fünfzig Services neu einrichten, nur weil man mit zwei Sekunden anstatt mit einer Sekunde als Timeout arbeiten möchte.

In der gezeigten Demo befindet sich ein CoreOS-etcd-Server *(siehe „https://coreos.com/etcd")*, in dem die Hystrix-Konfiguration abgelegt ist, die so bequem über das REST-API des etcd-Servers geändert werden kann. Der etcd-Server kann in einem Cluster betrieben werden und bietet ein paar Gimmicks, mit denen man zum Beispiel ein Feature-Toggle implementieren kann. Hier wird eine Property mit dem neuen Wert „true" via REST-Call gesetzt und dies zeitlich auf fünf Minuten begrenzt. Danach nimmt die Property wieder ihren Ursprungswert „false" an. Dank Archaius können wir darauf via Callback reagieren und das neue Feature aktivieren beziehungsweise deaktivieren.

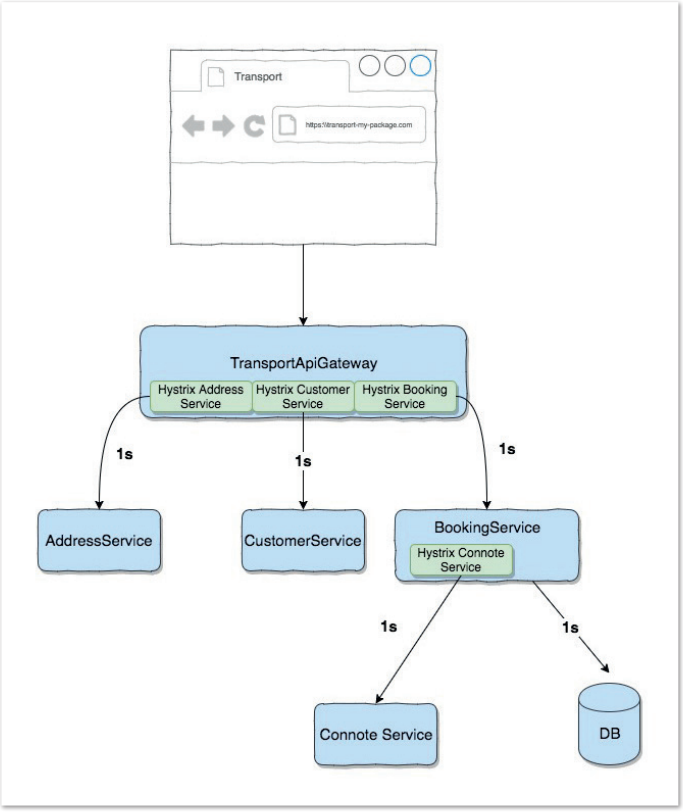

Abbildung 2: Architektur-Diagramm Hystrix

Eureka – Service Discovery

Wie finden sich nun unsere einzelnen Module untereinander und wie verwalten wir deren Zustand und auch die Anzahl der verfügbaren Instanzen? Das Mittel der Wahl ist eine Service Discovery, bei der sich jeder Service registriert. Ein anderer Service kann sich alle aktuellen Instanzen bei der Service Discovery abholen und für den Aufruf verwenden. Damit ist es allerdings noch nicht getan – in unserem Beispiel verwenden wir Eureka *(siehe „https://github.com/netflix/eureka")* aus dem Hause Netflix.

Eureka führt für uns auch einen Health Check der registrierten Services durch, nimmt automatisch nicht mehr antwortende oder fehlerhafte Services aus der Liste und benachrichtigt alle registrierten Consumer darüber. Eureka ist so aufgebaut, dass die Consumer eine lokale Kopie der registrierten Services vorhalten und diese periodisch aktualisieren. Somit sind die Services nach dem Start auch ohne eine laufende Instanz eines Eureka Servers in der Lage weiterzuarbeiten.

Ribbon

Wie könnte es auch anders sein, auch Ribbon *(siehe „https://github.com/Netflix/ribbon")* kommt aus dem Hause Netflix und ermöglicht im Zusammenspiel mit Eureka unter anderem ein Client-Side-Load-Balancing. Die verfügbaren Instanzen sind über Eureka bekannt und werden im einfachen Round-Robin-Verfahren angesprochen. Sollte Eureka feststellen, dass eine Instanz nicht mehr korrekt antwortet, wird diese vom Eureka-Client nicht weiter verwendet.

Dank Spring ist es möglich, mit einem „LoadBalanced Rest"-Template *(siehe „https://spring.io/guides/gs/client-side-load-balancing")* zu arbeiten, um das Client-Side-Load-Balancing durchzuführen. Als Endpoint wird dabei der Name des in Eureka registrierten Service

```
...
@LoadBalanced
  @Bean
  RestTemplate restTemplate(){
    return new RestTemplate();
  }

  @Autowired
  RestTemplate restTemplate;

@RequestMapping("/hi")
  public String hi(@RequestParam(value="name", defaultValue="You") String name) {
    String greeting = this.restTemplate.getForObject("http://greeting-service/greeting", String.class);
    return String.format("%s, %s!", greeting, name);
  }
...
```

Listing 1

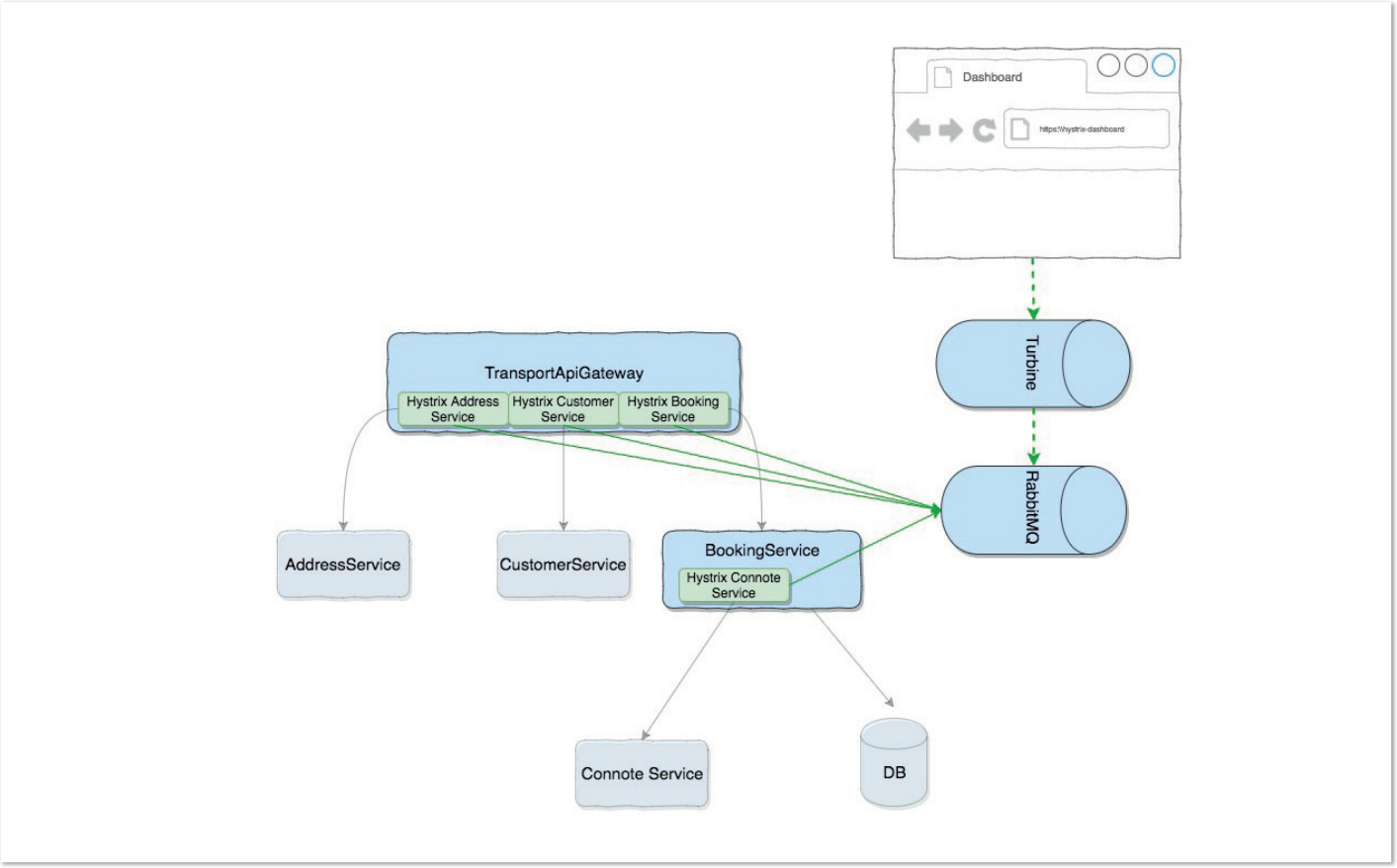

Abbildung 3: Architektur-Diagramm-Metriken via RabbitMQ

verwendet, dieser wird vom RestTemplate mit der entsprechenden URL und dem zugeteilten Port ersetzt. Somit ist der Entwickler völlig frei von der Umgebung, in der der Service ausgeführt wird, und muss sich nicht mit passenden Ports beziehungsweise URLs herumschlagen *(siehe Listing 1)*. Wie in dem Beispiel der Spring-Kollegen *(siehe „https://spring.io/guides/gs/client-side-load-balancing")* zu erkennen ist, verwendet das RestTemplate den Service-Namen „greeting-service", der zuvor in Eureka registriert worden ist.

AOP Chaos Monkey

So langsam wächst unsere kleine Beispielanwendung und es wird Zeit für ein wenig mehr Leben. Es ist doch sehr ernüchternd und langweilig für einen Entwickler, eine neue Anwendung zu gestalten und nie zu erleben, wie die Anwendung mit zufälligem Chaos umgeht. Ein ungutes Bauchgefühl bleibt, sollte man die entwickelte Anwendung nie unter Stress und im Zusammenspiel gesehen haben. Den erfahrenen Entwickler erkennt man dann am spontanen Fernbleiben zur Release-Einführung. Viel besser wäre es doch, die Anwendung durch einen kleinen Helfer und ein paar seiner bösen Freunde in einen instabilen Zustand zu versetzen, um zu sehen, ob die unabhängigen Module und die daraus entstehenden Bulkheads, der Einsatz von Hystrix, Eureka und Ribbon auch wirklich funktionieren.

Bei Netflix hat dies zur Simian Army geführt, mit der sich jeder Entwickler messen lassen und deren Angriffen jedes entwickelte System standhalten muss. Einer der bekanntesten Protagonisten ist ein Kollege mit dem passenden Namen „Chaos Monkey". Bei Netflix hat er die Aufgabe, AWS-Instanzen im laufenden Betrieb abzuschießen und so sicherzustellen, dass die Entwickler den aufrufenden Service entsprechend robust, also resilient gebaut haben. Diesen Ansatz findet der Autor sehr interessant und hat sich daher einen eigenen Chaos Monkey für Spring geschrieben. Mit einer einfachen Annotation „@EnableChaosMonkey" kann er in seiner Spring-Boot-Applikation einen Chaos Monkey bereitstellen, den er mithilfe von Archaius dynamisch zur Laufzeit aktivieren kann. Sollte der Chaos Monkey aktiviert sein, wird er per Zufall darüber entscheiden, wie er uns das Leben zur Hölle machen möchte. Folgende Optionen bieten sich hier:

- RuntimeException
- Timeout
- nichts

Im besten Fall haben wir also nichts zu befürchten und wir können auf eine funktionierende Anwendung hoffen. Dies wird mit den aktuellen Einstellungen, mit denen der Chaos Monkey läuft, jedoch nicht allzu oft passieren. Er wird sich an jede Komponente der Beispielanwendung hängen, die einen Service bereitstellt, und dort jede „public"-Methode beeinflussen. Nur so ist es möglich, Hystrix im tatsächlichen Einsatz zu sehen und zu der Erkenntnis zu gelangen, ob sich dessen Einsatz wirklich lohnt.

Hystrix-Metriken

Ein weiteres sehr nützliches Feature von Hystrix sind die Metriken, die wir frei Haus geliefert bekommen. Hystrix muss für die korrekte Zustandsberechnung des Circuit Breaker die Aufrufe überwachen und für einen definierten Zeitraum vorhalten. Diesen Zeitraum können wir, wie von Hystrix gewohnt, an unsere Bedürfnisse anpassen. Dank unseres Chaos Monkey haben wir richtig Leben in der Bude und sehen Hystrix in Action. Dies wird sich auch in den Metriken von

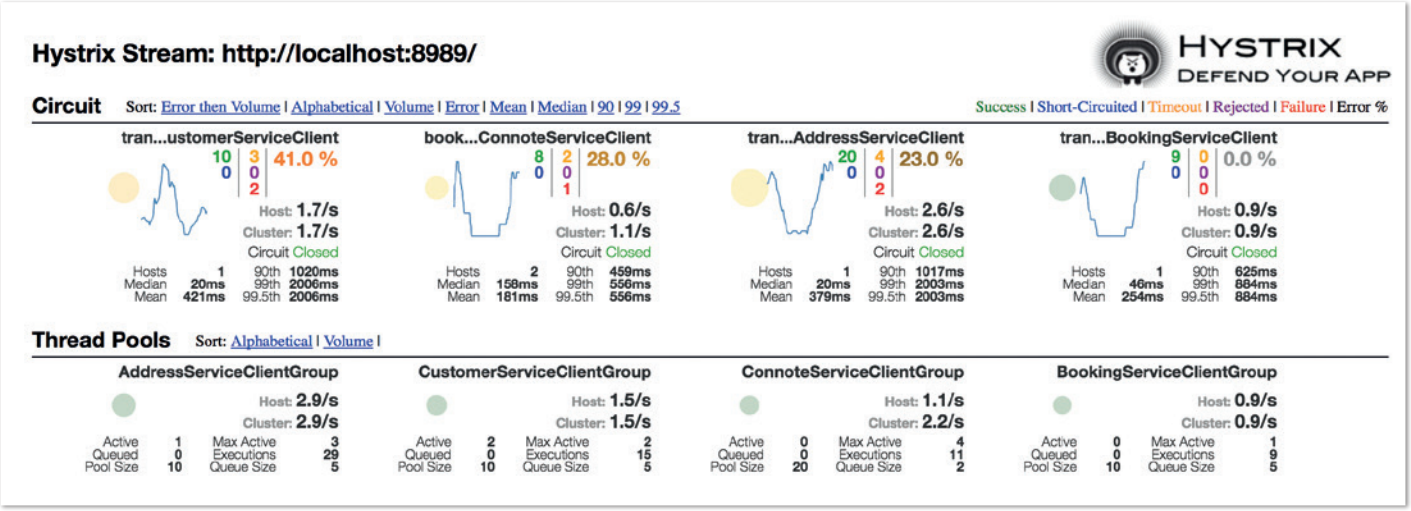

Abbildung 4: Das Hystrix-Dashboard

Hystrix widerspiegeln. Wie können wir uns diese Metriken nun zunutze machen und vor allem: Wie kommen wir an diese sehr wichtigen Daten?

Jede Komponente, in der wir Hystrix einsetzen, kann uns einen HTTP-Stream mit allen Metriken liefern. Dies wäre einer der Wege, die wir gehen können. Da wir aber eine verteilte Anwendung haben und nicht immer wissen können, unter welcher IP und mit welchem Port die Anwendung läuft, wäre es doch viel schöner, wenn die Anwendung uns die Daten über einen definierten Kanal liefert.

In der Beispielanwendung hat sich der Autor für einen Messaging-Weg entschieden und RabbitMQ eingesetzt. Dies hat den Vorteil, dass alle Komponenten die Daten in eine Queue ablegen, an der sich die Hystrix-Turbine *(siehe „https://github.com/Netflix/Turbine/wiki")* als Consumer registriert und die Daten aggregiert zurückliefert *(siehe Abbildung 3)*.

Die Turbine hat dabei die Aufgabe, alle eintreffenden Metriken zu einem zentralen Stream zu aggregieren, der dann vom Hystrix-Dashboard visualisiert oder von anderen Consumern gespeichert, aufbereitet und ausgewertet werden kann. Somit können wir in den Metriken auch erkennen, wie viele Instanzen unseres Service laufen und wie viele Requests wir über alle Instanzen eines Service hinweg haben.

Hystrix-Dashboard

Eines der ersten und vor allem schnell in Betrieb zu nehmenden Monitoring-Tools ist das Hystrix-Dashboard. Damit ist es sehr einfach und schnell möglich, einen umfangreichen Überblick über den Zustand unserer Anwendung zu erhalten. Ein Gesamtbild der verteilten Anwendung erhalten wir aber nur, wenn wir uns als Datenquelle den Turbine-Stream mit allen aggregierten Streams unserer verteilten Anwendung nehmen *(siehe Abbildung 4)*.

Fazit

Der Artikel gibt einen guten und vor allem praxisnahen Überblick zum Thema „Hystrix" und zu den sehr nützlichen Projekten aus dem Hause Netflix. Hystrix ist kein Allheilmittel und kann auch zu einem unerwünschten Verhalten führen, wenn es nicht korrekt konfiguriert wurde. So ist es zum Beispiel möglich, dass sehr viele Requests aufgrund einer falschen Hystrix-Konfiguration in einem Fallback landen. Um dies zu verhindern, müssen die Metriken herangezogen werden, und es erfordert auch viel Zeit und Vorbereitungen, die richtigen Werte der Konfiguration zu ermitteln.

Einige der in der Beispielanwendung verwendeten Tools sind in diesem Artikel leider nicht erläutert, so kommen zum Beispiel auch der Spring-Boot-Admin zur Administration von Spring-Boot-Anwendungen oder das Open-Tracing-Tool ZipKin zum Einsatz, die nicht unerwähnt bleiben sollen. Diese beiden Tools helfen ungemein dabei, die beteiligten Akteure einer verteilten Spring-Boot-Anwendung übersichtlich im Auge zu behalten. ZipKin ermöglicht es, die Datenströme und die bei einem Request beteiligten Instanzen eines Service visualisiert zu bekommen. Dank ZipKin ist es möglich zu sehen, welchen Weg ein Request genommen hat und welcher Service geantwortet hat. Das Thema „Open Tracing" in einer verteilten Spring-Boot-Anwendung ist auf jeden Fall einen Blick Wert und erleichtert die Fehlersuche ungemein.

Benjamin Wilms
benjamin.wilms@codecentric.de

Benjamin Wilms arbeitet als Senior IT Consultant und Developer bei der codecentric AG. Seine aktuellen Schwerpunktthemen sind Skalierbarkeit und Resilience in verteilten Anwendungen. Er teilt und diskutiert seine Ideen regelmäßig auf Konferenzen sowie als Autor von Artikeln und Blog-Posts.