Möglichkeiten zur Kontrolle und Steuerung der Kommunikation von Microservices

**Bachelorarbeit**

im Studiengang  
Softwaretechnik und Medieninformatik

vorgelegt von

**Gerrit Wildermuth**Matr.-Nr.: 74734

am 28. Februar 2020   
an der Hochschule Esslingen

Erstprüfer/in: Prof. Dr. Mirko Sonntag   
Zweitprüfer/in: Prof. Dr. Steffen Schober

# Kurzfassung

Der Titel dieser Arbeit weist direkt auf den Hauptteil dieser Arbeit hin. Bei diesem wird sich im Bereich der Steuerung genauer mit der Thematik der Leitung und Management der Kommunikation beschäftigt als auch wie diese gewährleistet werden kann. Zu jeder dieser Thematiken werden hierbei unterschiedliche Ansätze untersucht, welche sich sowohl in der Kommunikationsebene als auch der Implementierungstiefe unterscheiden und in einer Microservice Umgebung eingesetzt werden können. Bevor es allerdings in den Hauptteil geht, wird sich zuerst mit der Thematik von Microservices im Allgemeinen auseinandergesetzt, sowie eine Einführung in weitere Themenbereiche der verteilten Systeme, welche für ein späteres Verständnis benötigt werden.

Der Grund für die Notwendigkeit dieser Arbeit entstammt der Problematik der verteilten Systeme. Um mehr Leistung und größere Systeme zu ermöglichen wurde zu verteilten Systemen gewechselt. Hierbei kamen allerdings auch einige Probleme mit sich, welche sich unter anderem in einer erhöhten Kommunikation auswirkte. Um schneller auf Lastenunterschiede reagieren zu können und weniger Ressourcen zu verbrauchen wird hierbei vermehrt auf Microservices gesetzt. Um hierbei einen Überblick zu bekommen welche Techniken und Bausteine hierbei benötigt werden oder Hilfreich sind um Systeme zu gestalten, welche mit dieser Problematik umgehen, wurde sich mit dieser Thematik befasst und unterschiedliche Möglichkeiten gesucht diese Umzusetzen.

Hierbei wurde im Speziellen der Unterschied zwischen einer eher herkömmlichen Umsetzung in Java mit Spring Boot mit einer alternativen Herangehensweise durch den Einsatz eines Service Meshes untersucht.

**Schlagwörter**: Microservice, Service Mesh, verteilte Systeme, Kommunikation.

# Abstract

This thesis is built around distributed systems and the emerging problems which come along with choosing such architecture. First it is taken a look at the theoretical background of microservices and distributed systems as well as additional systems which are needed later on. After that the main part consists around different techniques and components which are needed or helpful in the context of microservices in distributed systems and the problems which come along with it.

For the different approaches it was specifically looked into Java with Spring Boot and a service mesh approach with Istio.

**Keywords:** microservice, distributed systems, service mesh, techniques, components.

# Inhaltsverzeichnis

Kurzfassung 2

Abstract 2

Inhaltsverzeichnis 3

Abbildungsverzeichnis 6

Tabellenverzeichnis 6

Codeverzeichnis 6

Abkürzungsverzeichnis 8

1 Einleitung 9

2 Microservices 11

2.1 Wieso Microservices 11

2.2 Monolith 12

2.2.1 Einführung 12

2.2.2 Vorteile der Monolithen 12

2.2.3 Nachteile der Monolithen 13

2.3 Microservices 13

2.3.1 Einführung 13

2.3.1 Vorteile der Microservices 14

2.3.2 Nachteile der Microservices 15

2.4 Zusammenfassung was steht jetzt fest was ist klar? 16

3 Verteilte Systeme 17

3.1 Definition 17

3.2 Orchestrierung 17

3.3 Netzwerk/Kommunikation in einem Verteilten System 18

3.4 Von Fehler Potential zu Robustheit 18

3.5 Daten Beständigkeit 19

3.6 Einführung in Kubernetes 19

3.6.1 Kubernetes Komponenten 20

3.7 Vereinheitlichte Systeme 22

3.7.1 Service-Mesh 22

3.7.2 Spring Cloud 25

3.8 Zusammenfassung was steht jetzt fest was ist klar? 28

4 Möglichkeiten zum Management und Gewährleisten eines verteilten Systems 29

4.1 Kommunikations-Leitung 29

4.1.1 API Gateway 29

4.1.2 API Microgateway 31

4.2 Gewährleistung von Services 32

4.2.1 Einleitung 34

4.2.2 Resilient4j VS Failsafe 35

4.2.3 Bulkhead 36

4.2.4 Circuit breaker 40

4.2.5 Retry 43

4.2.6 Timeout? **Fehler! Textmarke nicht definiert.**

4.2.7 Rate Limiting 44

4.2.8 Fallback 45

4.2.9 Zusammenfassung 47

4.3 Kommunikations- Überwachung und Verfolgung 32

4.3.1 Einleitung 32

4.3.2 API Management 32

4.3.3 Distributed tracing 33

4.4 Service discovery 47

4.4.1 Wo sind alle Services? **Fehler! Textmarke nicht definiert.**

4.4.2 Was ist Service discovery 48

4.4.3 Serverseitige discovery 48

4.4.4 Clientseitige discovery 49

4.4.5 Selbstregistrierungs Methode 50

4.4.6 Drittparteiregistrierungs Methode 50

4.5 Status und Daten Verbreitung in verteilten Systemen **Fehler! Textmarke nicht definiert.**

4.5.1 Daten Konsistenz **Fehler! Textmarke nicht definiert.**

4.5.2 Konfiguration Server **Fehler! Textmarke nicht definiert.**

5 Umsetzung eines robusten verteilten Systems mit Hilfe eines Service-Meshes Fehler! Textmarke nicht definiert.

5.1 Das Service-Mesh Istio 25

5.1.1 Istio Architecture **Fehler! Textmarke nicht definiert.**

5.1.2 Istio Notizen **Fehler! Textmarke nicht definiert.**

Zusammenfassung und Ausblick 52

Glossar 55

Ehrenwörtliche Erklärung 56

Stichwortverzeichnis 57

6 Literaturverzeichnis Fehler! Textmarke nicht definiert.

# Abbildungsverzeichnis

Abbildung 1 Monolith gegen Microservices Quelle: Eigene Darstellung 12

Abbildung 2: Container VS virtuelle Maschinen Vorteile der Microservices (Sebastian Eschweiler 2019) 14

Abbildung 3: Kubernetes Industrie Annahme (StackRox 2019) 17

Abbildung 4: Container deployment in Kubernetes (Ushio 2018) 19

Abbildung 5: Kubernetes Komponenten und deren Interaktion (Ushio 2018) 21

Abbildung 6: Deplyoment Yaml (Kubernetes) 22

Abbildung 7: Nutzung von Bibliotheken VS Auslagerung in Sidecars (Prinz 2019) 23

Abbildung 8: Service Mesh Architektur (Prinz 2019) 24

Abbildung 9: Spring Annahme (Paraschiv 2018) 26

Abbildung 10: API Gateway Übersicht (Richardson 11/20/2019) 30

Abbildung 11: API Gateways Backends for frontends (Richardson 11/20/2019) 31

Abbildung 13: Schiff Links Ohne Bulkhead (IBM 11/28/2019) 36

Abbildung 14: Schiff Rechts mit Bulkhead (IBM 11/28/2019) 36

Abbildung 14: Fehlerhafter Service ohne Bulkheads (Eigenkreation) 37

Abbildung 15: Fehlerhafter Service mit Bulkhead (Eigenkreation) 37

Abbildung 16: Implementierungswaage (Eigenkreation) 38

Abbildung 13: Circuit Breaker Zustände (Fowler 2014) 41

Abbildung 14: Retry Pattern (Microsoft) 44

Abbildung 14: Rate Limiter (Storozhuk 2018) 45

Abbildung 15: API Managment Gateway Overview (Posta 2019) 33

Abbildung 16: Distributed Tracing (Kyma) 34

Abbildung 17: Wiso ist Service discovery notwendig (Richardson 2015) 47

Abbildung 18: Server-Side discovery Pattern (Richardson 2015) 48

Abbildung 19: Client-side discovery Pattern (Richardson 2015) 49

# Tabellenverzeichnis

Tabelle 1: Unterschiede zwischen Abbildungen und Tabellen **Fehler! Textmarke nicht definiert.**

Tabelle 2: Beispiele für Überschriftebenen **Fehler! Textmarke nicht definiert.**

Tabelle 3: Aufstellung der wichtigsten Formatvorlagen der Dokumentvorlage **Fehler! Textmarke nicht definiert.**

# Codeverzeichnis

[Codeteil 1: Bulkhead Serverseitige Implementierung in Java mit resilience4j in Plain Java 39](#_Toc31640628)

[Codeteil 2: Bulkhead Fehlermeldung wenn der Threadpool vollgelaufen ist. 39](#_Toc31640629)

[Codeteil 3: Resilient4j mit Spring Boot 2. 39](#_Toc31640630)

[Codeteil 4: Methode welche mit Resilient Annotationen des Circuit breakers versehen ist. 43](#_Toc31640631)

[Codeteil 5: Circuit breaker Einstellungen in der application.properties 43](#_Toc31640632)

[Codeteil 6: Method mit Resilient4j Retry 44](#_Toc31640633)

[Codeteil 7: Methode mit Fallback 46](#_Toc31640634)

[Codeteil 8: Aufzurufende Fallback Methode 46](#_Toc31640635)

# Abkürzungsverzeichnis

ALP Arbeits-, Lern- und Präsentationstechniken

HBI Hochschule für Bibliotheks- und Informationswesen

HdM Hochschule der Medien

HSE Hochschule Esslingen

# Einleitung

Diese Arbeit beschäftigt sich im Besonderen mit der Kommunikation zwischen bzw. von Microservices und möchte hierbei unterschiedliche Möglichkeiten für Techniken und Bausteine im Kontext der verteilten Systeme untersuchen. Zuallererst wird sich dafür mit theoretischen Hintergründen von Microservices, verteilten Systemen und weiteren benötigten Grundlagen befasst welche für ein Verständnis später benötigt werden. Die Notwendigkeit dieser Arbeit entsteht dem Problem, welches fast unweigerlich auftaucht, wenn sich mit verteilten Systemen beschäftigt wird. Durch ein verteiltes und vernetztes System kann bei weitem mehr Kommunikation entstehen wie dies bei bisherigen Systemen der Fall war. Dieser Kommunikationsanstieg kann dazu führen das deutlich mehr Fehler auftreten, sich deutlich mehr um den Kommunikationsfluss gekümmert werden muss, es nicht mehr möglich ist manuell einzelne Serviceadressen weiterzuverteilen und ein Management einzelner APIs immer schwieriger wird. Es gibt im Umfeld der verteilten Systeme durchaus noch weitere Probleme, um welche sich gekümmert werden könnte, jedoch werden diese im Umfang dieser Arbeit nicht näher betrachtet.

Es werden hierbei unterschiedliche Möglichkeiten geboten, um ein Robusteres verteiltes System zu ermöglichen und mit Fehlern wie Kettenreaktionen, Ressourcen Erschöpfung, Netzwerkunterbrechungen oder Methoden welche zufällig einen Fehler werfen, entsprechend umzugehen.

Bevor allerdings Kommunikationsfehler auftreten können benötigt es die Kommunikation. Um die Kommunikation zu ermöglichen, benötigen wir Kenntnis über unsere Services im Speziellen über ihren Aufenthaltsort. Damit wir diesen in einem sich möglicherweise schnell verändernden System noch finden können benötigen wir eine dynamische Lösung zum Lösen dieses dynamischen Problems.

Nachdem wir Kenntnis über den Aufenthaltsort unserer Services haben muss der Kommunikationsfluss in unserem verteilten System wieder ermöglicht bzw. hergestellt werden. Hierfür wird sich mit Bausteinen beschäftigt, welche helfen sich mit diesem Problem zu befassen und es möglich machen unsere Anfragen durch unser System durchzuleiten, um sie genau dort hinzuführen, wo sie hingehören.

Nachdem der Kommunikationsfluss wiederhergestellt ist und die APIs wieder zugänglich sind, wird es daraufhin wichtig zu kontrollieren, wer Zugriff auf unser System hat. Sowohl Nutzer als auch Entwicklern benötigen eigene Möglichkeiten sich mit unserem System zu verbinden und mit diesem zu arbeiten, unbewusst oder bewusst.

Um diese Probleme zu bändigen, sollen sowohl unterschiedliche Möglichkeiten auf der Kommunikationsebene als auch in der Implementierungstiefe geboten werden, wobei wir uns hierbei auf die Programmiersprache Java beschränken um den Rahmen nicht zu sprängen. Die hierbei Verwendung findenden Umgebungen sind zum einen Spring Boot/Cloud, als auch das Service Mesh Istio welche beide im Verlauf der Arbeit noch erklärt werden.

Im Folgenden wird eine Einführung in Microservices gegeben, mit einer Gegenüberstellung zu einem Monolithischen Ansatz, welcher dabei helfen soll herauszufinden, ob ein Microservice Ansatz der Richtig für das entsprechende Problem ist, sowie einige weitere Probleme aufzeigt welche hiermit einherkommen können.

# Microservices

Im folgendem wird sich mit Microservices befasst und Erklärt woher sie kommen worauf sie Aufgebaut sind und eine Einführung in die Thematik dieser gegeben. Des Weiteren wird eine Gegenüberstellung mit einem Monolithischen System geben wodurch einige weitere vor und Nachteile dieser beiden Systeme veranschaulicht werden um dem Leser die Möglichkeit zu geben selbst eine Entscheidung zu treffen welches eher das richtige System für seinen Zweck ist.

## Wieso Microservices

Der Ursprung von Microservices entstammt aus Problemen welche mit bisherigen monolithischen Architekturen nicht mehr zu bewältigen waren. “The many things that you would like to see happening in a good software environment couldn’t be done anymore. “The many things that you would like to see happening in a good software environment couldn’t be done anymore; (Vogels 6/30/2006). Diese Systeme konnten ab einem Zeitpunkt/Größe nicht länger Wachsen/sich entwickeln. “It couldn’t evolve anymore. The parts that needed to scale independently were tied into sharing resources with other unknown code paths.” (Vogels 6/30/2006). Die monolithische Architektur ist in diesem Fall an ihre Grenzen gekommen und eine neue Architektur musste die alte ersetzten. Eine Architektur welche es möglich macht Softwarekomponenten schnell, unabhängig und vor allem Isoliert zu bauen und zu Skalieren. Die Microservice Architektur ist hierbei eine solcher Architekturen welche es ermöglicht diese Punkte umzusetzen und ist ebenso eine sehr stark wachsende Architektur „Microservices have become mainstream in the enterprise“ (Dimensional Research 2018).

## Monolith

### Einführung

Ein Monolith ist im Deutschen ein einzelner Stein. In unserem Fall spiegelt es ein zusammenhängendes Softwaresystem wider, welches alle Komponenten in sich vereinigt. Diese Komponenten wie in Abbildung 1 zu sehen ist können in unterschiedliche Teile aufgeteilt sein aber das System wird als einheitliches Paket veröffentlicht. Diese eng verbundene Bauweise ist ihr Segen wie auch ihr Fluch zugleich. Die einzelnen Teile des Systems können dadurch direkt lokal Kommunizieren und so einen sehr hohen Datendurchsatz erreichen. Auf der andern Seite kann nur die einzelne Maschine auf der das System veröffentlicht wurde weiter verstärkt werden, nicht aber das System auf mehrere Standorte verteilt und individuell Skaliert werden. Außerdem muss bei jeder Änderung auch das gesamte System neu veröffentlicht werden, Teilbereiche separat zu veröffentlichen ist hierbei nicht möglich.

Abbildung 1 Monolith gegen Microservices Quelle: Eigene Darstellung

### Vorteile der Monolithen

* Der größte Vorteil eines Monolithen ist, die nahe Verknüpfung der einzelnen Bestandteile und somit kann auf weitläufige Kommunikation verzichtet werden und einiges an Fehlern und Arbeitsaufwand vermieden werden. Es ist dadurch möglich schneller zu Ergebnissen zu kommen, was sich vor allem zu Beginn eines Projektes bemerkbar macht da keine zusätzlichen Systeme benötigt werden um z.B. das System zu veröffentlichen.
* Durch die zusammenhängende Architektur, gestaltet es sich einfacher, das System auf genau einem Server zum Laufen zu bringen. Dies ist dadurch möglich, dass keine weiteren Verbindungen, Ports oder ähnliches festgelegt werden müssen da alles auf einem und demselben System läuft und keine weiteren verwendet werden können.
* Die Datenbank ist an einem Punkt wodurch Daten Konsistenz deutlich einfach bzw. überhaupt erreicht werden kann.

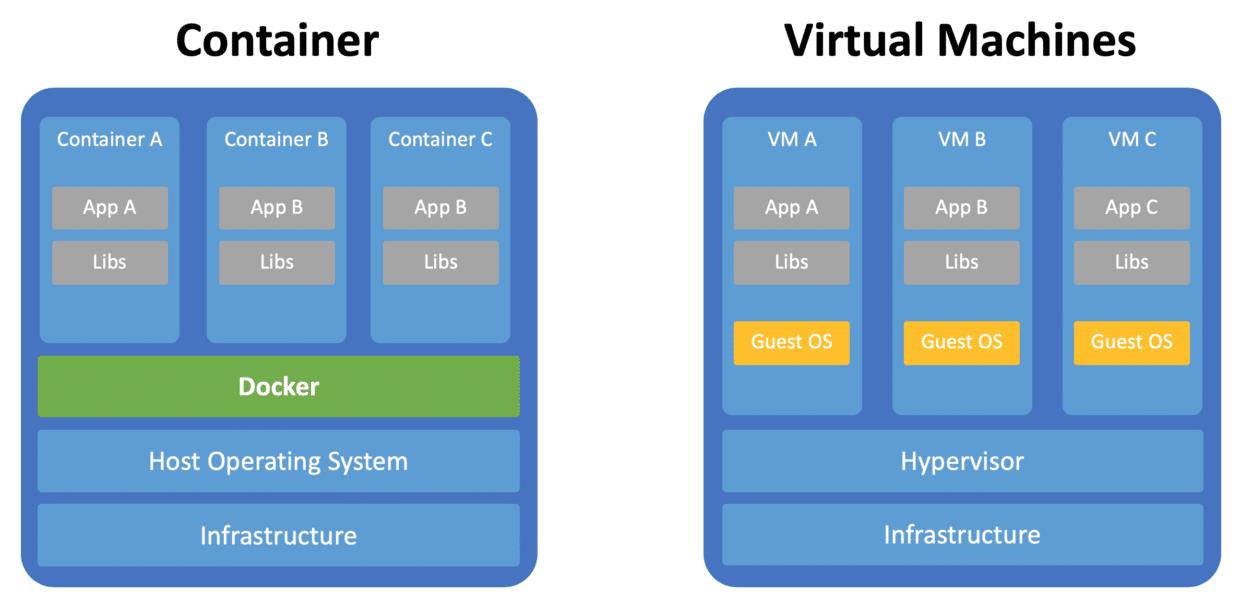
### Nachteile der Monolithen

* Jede Änderung des Systems, egal ob klein oder groß, hat zur Ursache, dass das gesamte System neu bereitgestellt werden muss. Dies spielt keine große Rolle, solange die Anwendung noch klein ist doch je größer sie wird, umso länger dauert dieser Prozess.
* Ein weiteres Problem von zu groß werdenden zusammenhängenden Systemen ist, dass durch fehlende Isolierung jede Änderung ungeahnte Folgen haben kann. Dadurch wird es schwieriger neue Technologien in ein bestehendes System zu integrieren.
* Je größer die Anwendung wird, also umso mehr Zeilen Code sie enthält und zusätzliche Features implementiert werden, umso wertvoller werden Entwickler die sich bereits mit dem System auskennen. Ebenso kann es auch immer schwieriger für neue Entwickler werden einzusteigen und eigene Beiträge beizusteuern.
* Da die einzelnen Teilbereiche stark miteinander verbunden sind und oft gleichzeitig an vielen Orten gearbeitet werden muss, können sich zu viele Entwickler leicht in die Quere kommen. Dies hat zur Folge, dass früher eine maximale Anzahl an Entwicklern erreicht wird.

## Microservices

### Einführung

Verschiedene Bereiche der Anwendung werden aufgeteilt und ausgelagert in eigene Services. Jeder Service kommuniziert über Schnittstellen mit anderen Services bzw. bietet diese an. Jeder Service ist eine eigenständige leichte, also schnell startende und wenig Platz benötigende Anwendung. Für eine bessere Performance und Isolierung wird häufig eine einzelne Datenbank siehe Monolith VS Microservices in Abbildung 1 in viele einzelne Datenbanken aufgeteilt. Es hat sich hierbei etabliert, dass Microservices für gewöhnlich in Containern auftreten. „The best choice for running a microservices application architecture is application containers.” (Golden 5/2/2017) Container bilden hierbei ein Gehäuse für eine entsprechende Anwendung mit all ihren Abhängigkeiten, Laufzeitumgebungen, System Werkzeugen, System Bibliotheken und Einstellungen (Abbildung 2). Die Container bilden isolierte Bereiche dar mit ihren eigenen Namenräumen, Benutzern und Ressourcen. Sie stellen hierbei allerdings keine vollwertigen Virtuellen Maschinen dar und Besitzen kein eigenes Betriebssystem oder einen eigenen Kernel, dies ist allerdings auch gleichzeitig der Grund für ihre Schlankere Bauweise wie man im Vergleich in Abbildung 2 sehen kann. „Just from an efficiency perspective, containers are a far better choice for a microservices architecture than are VMs.” (Golden 5/2/2017)

Abbildung 2: Container VS virtuelle Maschinen Vorteile der Microservices (Sebastian Eschweiler 2019)

### Vorteile der Microservices

* Die Anwendung in kleinere handlichere Teilbereiche zu zerteilen hilft dabei den Code leichter zu verstehen und sich besser auf das Wesentliche zu konzentrieren, zu Entwicklern und zu Pflegen.
* Je größer die Anwendung wird, umso eher können ganze Teams auf einzelne Services angesetzt werden und diese Entwickeln, anstatt sich um die gesamte Anwendung kümmern zu müssen.
* Vorausgesetzt die Services sind nicht eng miteinander verzahnt, können Entwicklerteams freier Entscheiden was für Technologien, Frameworks oder sprachen sie benutzten.
* Die jeweiligen Bereitstellungen, können durch eine Isolierung, vom restlichen System getrennt werden, was es einfacher macht Änderungen durchzuführen. Tests, bleiben ebenso auf kleinere Services begrenzt, wodurch sie schneller Veröffentlicht werden können.
* Skalieren der Services wird Möglich und durch die Ressourcensparende Bauweise Kostensparender. Im Gegensatz zum monolithischen System, wo jeweils das einzelne System verstärkt wird, werden im Microservice Kontext nur diejenigen Skaliert welche wirklich benötigt werden, ohne unnötige Ressourcen zu verschwenden.
* Durch die starke unabhängige Modularität, können Updates je nach Service vollständig unbemerkt durchgeführt werden und es ist nicht mehr notwendig das gesamte System Abzuschalten. Durch diesen beschleunigten Prozess können Entwickler auch schneller auf Änderungen oder Fehler reagieren.
* Wenn Fehler in Services keine Kettenreaktion herbeiführen, wird eine höhere Robustheit erreicht und das gesamte System kann stabiler laufen.
* Durch die Verwendung von Service discovery können automatisch skalierte Services sofort verwendet werden.
* Durch die Verwendung von Management Komponenten wird es einfacher die eigenen APIs zu beherrschen.
* Services können, durch die sprachliche Trennung, einfach in anderen Projekten wiederverwendet werden.

### Nachteile der Microservices

* Da Microservices in der Regel in einem verteilten System zuhause sind und dadurch über das Netzwerk getrennt wird es bei größeren Systemen deutlich schwieriger Fehler zu finden. Dies wird dadurch noch verstärkt, wenn die Services von unterschiedlichen Teams entwickelt werden.
* Wenn Services gebaut werden kann die Unterteilung der logischen Prozesse zu weit gehen, wodurch diese zu sehr auseinander gezerrt werden was die Nützlichkeit der Verteilung überschreiten und die Prozesse unnötig verlangsamt.
* Die Notwendigkeit, der Verwendung von Docker oder Kubernetes um das verteilte System zu steuern bzw. Bereitzustellen, möchten Manche nicht auf sich nehmen. Da z.B. sie nicht noch weitere Technologien benutzen möchten oder das Know-how fehlt.
* Mit vielen Abhängigkeiten zwischen den Services, kann es herausfordernd werden, ein Ende zu Ende Test durchzuführen.
* Kommunikation zwischen Services, kann sehr teuer werden, wenn z.B. an der falschen Stelle auf synchrone anstatt asynchrone Kommunikation gesetzt wird, was zu lange blockenden Anfragen führen kann.
* Datenbank Anfragen, können dazu führen das zusätzliche Datenbanken angesprochen werden müssen, welche zu anderen Services gehören. Dies kann zu Problemen führen(wenn diese z. B. mehrfach kaskadierend ist).

## Zusammenfassung

Zum Ende dieses Abschnittes sollten einige Aspekte verstanden worden sein. Zum ersten was Microservices sind und wie ihre Verbindung mit Containern sind. Weiterhin sollte klar sein was ein Service ist und was sein Äquivalent in einem Monolithen ist. Zum dritten was es mit der Monolithischen Architektur auf sich hat und wo die Unterschiede zur Microservice Architektur liegen. Außerdem sollte es möglich sein eine Entscheidung zwischen diesen beiden Architekturen zu fällen, um den passenderen Architekturstil für das jeweilige Projekt zu finden. Dem entsprechend sollte auch ein Überblick über die Risiken und Vorteile der beiden Architekturstile vorhanden sein.

Im Folgenden werden einige Aspekte von verteilten Systemen erläutert und eine tiefere theoretische Grundlage für die weitere Arbeit gegeben, wobei hierbei besonderer Bezug zu Microservices genommen wird und die genauere Problematik in verteilten Systemen aufzeigen wird.

# Verteilte Systeme

## Definition

Nach (Tanenbaum 2006) ist ein verteiltes System ein Zusammenschluss unabhängiger Computer welche dem Nutzer als ein einziges zusammenhängendes Systems erscheinen. In unserem Fall, betrachten wir dieses verteilte System als ein zusammenhängendes System aus Microservices. Dem entsprechend trifft unsere vorherige Definition der Microservices hier auf die einzelnen Komponenten/Computer zu.

## Orchestrierung

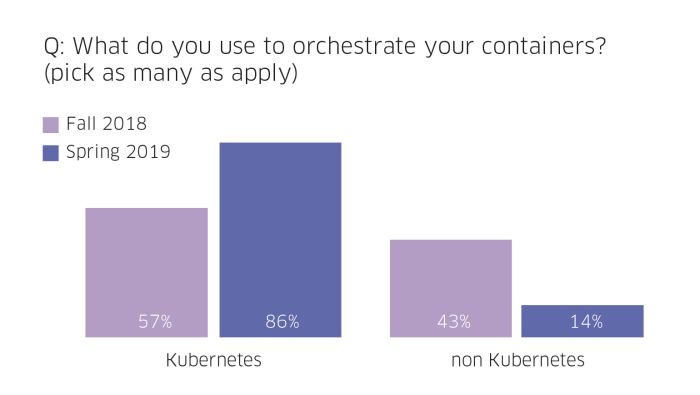
Es wird besonders wichtig, wenn ein verteiltes System mit einer Microservice Architektur verwirklicht wird, dass durch die deutlich größere Anzahl an Services und die dadurch entstehenden Schnittstellen diese gut gemanagt werden können. Durch die Verwendung von Containern und Autoskalierungsoptionen können Serviceinstanzen sich deutlich schneller vermehren. Für diese zum einen deutlich größere Anzahl an Services und zum anderen große Anzahl an Instanzen, benötigen wir neue Werkzeuge, um dieses Wachstum zu beherrschen und möglichst zu automatisieren. Diese Werkzeuge werden in der Fachsprache Orchestratoren gennant und werden dafür benutzt diese Services automatisiert zu konfigurieren, koordinieren und zu managen.

Abbildung 3: Kubernetes Industrie Annahme (StackRox 2019)

Sie unterscheiden sich hierbei natürlich in Funktionalität und Handhabung. In unseren Untersuchungen, werden wir zu der Orchestrierungs Plattform Kubernetes zurückgreifen, da sie zum Zeitpunkt der Thesis die am weitesten verbreitete Plattform ist und immer mehr Annahme in der Industrie erfährt siehe Abbildung 3.

## Kommunikation in einem Verteilten System

Der wohl größte Unterschied, zu einem nicht verteilten System, ist die Kommunikation. Durch die zusätzlichen Wege dauert die Kommunikation deutlich länger und kommt ab einer gewissen Größe an Ihre Grenzen. Sie kann außerdem immer wieder Fehlschlagen und gewährleistet nie eine hundert Prozentige Sicherheit. Durch diese Limitationen werden unterschiedliche Technologien benötigt und angewendet. Es können z. B. verzögerte Wiederholungen ausgeführt werden, Teilinformationen abgerufen oder gar noch gecachte Informationen weiter verwendet werden.

## Von Fehler Potential zu Robustheit

Im Kontext von verteilten Systemen geht es nicht mehr darum falls oder ob etwas fehlschlägt, sondern wann etwas fehlschlägt. Das heißt, wir rechnen fest damit, dass Fehler auftreten werden und Bereiten die Services darauf vor. Dies trifft besonders zu, da die erhöhte Anzahl und Vielfalt an Kommunikation, über das Netzwerk dies extra fördert. Doch, durch dieses Wissen, können wir entsprechende Komponenten und Sicherheitsmechanismen in unterschiedlichen Schichten einbauen, um diese Systeme Robuster zu gestalten.

Einige Fehlerquellen:

1. Es können Fehler im Netzwerk auftreten.
2. Die Netzwerkbandbreite kann überschritten werden.
3. Anfrage Wiederholungen können Lasten noch vergrößern.
4. Nachrichten Puffer können überlaufen.
5. Fehler in einzelnen Systemen, können weitere oder gar das gesamte System zum Einsturz bringen.
6. Es können immer wieder Topologische Änderungen auftreten.
7. Es können sich Datenbank Duplikate einschleichen.

(Rotem-Gal-Oz)

Einigen dieser Fehlerquellen werden später noch angeschaut und behandelt. Hierbei werden unterschiedliche Lösungsansätze geprüft und eingeordnet.

## Daten Beständigkeit

Bei einem monolithischen System ist es üblich, dass es eine zentrale Datenbank gibt, wo jeglicher Datenverkehr gebündelt wird bzw. stattfindet. In einem verteilten System wird ein anderer Ansatz verwendet und jedes System verwaltet seine eigene Datenbank. Demnach findet die Kommunikation größtenteils über APIs statt und nicht über eine geteilte Datenbank.

Da die jeweiligen Systeme eigenständige Datenbanken haben, können diese spezialisiert werden. Dies erlaubt verschieden Datenbank Systeme, in einem einzigen System zu verwenden.

Das Thema der verteilten Datenbeständigkeit mit all ihren Tücken und konsistenten Zuständen wird im weiteren Teil der Arbeit nicht weiter erläutert und sollte selbständig nacherforscht werden.

## Einführung in Kubernetes

Wie es bereits zuvor im Thema 3.2 Orchestrierung erwähnt wurde, wird zur Durchführung dieser Thesis die Orchestrierungsplattform Kubernetes als runtime (Laufzeitumgebung) für das verteilte System verwendet. Im Folgenden werden die einzelnen Komponenten von Kubernetes erklärt um ein grobes Verständnis darüber zu erlangen wie sie Funktionieren und miteinander Interagieren.

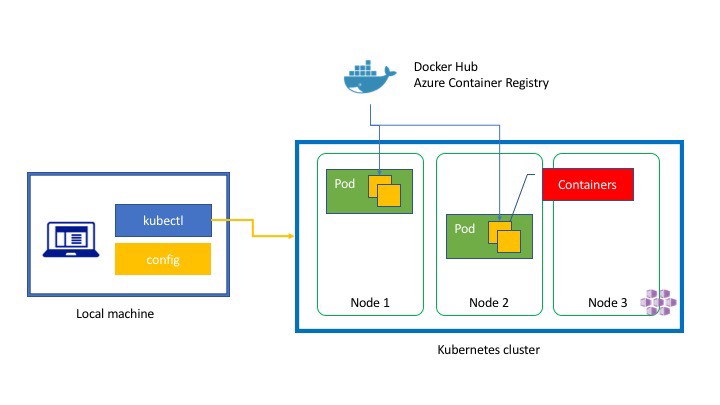


Abbildung 4: Container deployment in Kubernetes (Ushio 2018)

Um Container in Kubernetes verwenden zu können, müssen diese zuerst für diese Konfiguriert werden. Hierfür werden Yaml/Json Dateien benötigt welche diese spezifizieren und Unterschiedliche Parameter für diese Festlegen wie z.B. den Namen, den Port, das zu verwendende Image usw. Durch die Konfiguration weiß Kubernetes um welche Container Images es sich handelt und bezieht bzw. Konfiguriert diese Automatisch. Für diesen Prozess wird die CLI(Command Line Interface) Kubectl verwendet welche lokal Eingerichtet ist und in einer Konfigurationsdatei die nötigen Informationen für das momentane Cluster gespeichert hat. Die CLI Kubectl wird für die Interaktion mit dem Cluster verwendet und ist die Schnittstelle zu diesem.

### Kubernetes Komponenten

#### Pods

Ein Pod ist immer Bestandteil eines Nodes und wird automatisch über diese verteilt. Pods können einen oder mehrere Container beinhalten. (Abbildung 4: Container deployment in Kubernetes (Ushio 2018)) Jeder Container kommt mit einer Spezifikation welche für die Repräsentation und Einstellungen im Cluster beinhaltet. Falls Pods mehrere Container beinhalten, befinden sich diese im selben Netzwerk und können darüber miteinander kommunizieren, so wie auch einen möglichen Zugriff auf den gleichen Speicher vornehmen.

Mehr zu Multi-Containerkommunikation unter: <https://linchpiner.github.io/k8s-multi-container-pods.html>.

#### Nodes

Nodes zu Deutsch Knoten sind die Arbeitsmaschinen in einem Cluster, also entweder Virtuelle Maschine oder vollständiger Rechner. Die jeweiligen Nodes, werden von der Zentralen Komponente verwaltet und beinhalten alles was für den Betrieb der Pods benötigt wird. Dies umfasst die Container Runtime, das Kubelet und den Kube-Proxy.

#### Kubelet

Die Kubelet ist die am niedrigstehenden Komponente in Kubernetes und ist dafür verantwortlich was auf den einzelnen Nodes läuft. Sie stellt sicher, dass die jeweiligen Container laufen und nicht irgendwann einmal abgestützt sind. Der API Server spricht mit dem Kubelet in den jeweiligen Nodes. Die Kubelet ist dafür zuständig Informationen über die Container und deren Anwendung zu liefern und gibt diese an den API Server weiter.

#### Kube Proxy

Der Kube Proxy läuft auf jedem Node. Er spricht mit dem API Server und legt Traffic Regeln in IP Tabellen fest, leitet Traffic um und filtert das Netzwerk.

#### Replica Set

Das Replica Set stellt sicher das immer eine festgelegte Anzahl an Pods, eines Typs vorhanden und Verfügbar ist (Orangene Linien in Abbildung 5: Kubernetes Komponenten und deren Interaktion (Ushio 2018)). Jeder Pod wird außerdem mit einer owner Referencer ausgestattet wodurch diese Konkreten Replica Sets zugewiesen werden. Da die Replica Sets von den Deployments verwaltet und benutzt werden, kann es sein das diese niemals direkt manipuliert werden müssen, sondern dies durch die Spezifikation der Deployments schon erledigt hat.

#### Deployment

Abbildung 5: Kubernetes Komponenten und deren Interaktion (Ushio 2018)

Ein Deployment bestimmt welche Pods und Replica Sets Erstellt werden und beschreibt welchen erwünschten Status diese haben. Der Deployment Controller führt hierbei die eigentliche Arbeit aus. Der Controller verändert das Deployment auf eine kontrollierte weiße bis der erwünschte ist Zustand erreicht ist. Deployments können später noch upgedatet, zurückgesetzt, skaliert, gestoppt und wieder fortgesetzt werden.

Da das Erstellen von Deployments besonders wichtig ist, wird hierauf nun noch ein Besonderer Blick anhand eines Beispiels geworfen.

Deployments werden wie schon zuvor erwähnt mit Yaml oder Json Dateien erstellt, im Folgenden handelt es sich um eine Yaml Datei da dies generell der Standard ist.

Kind: Definiert um was es sich handelt, in diesem Fall ein Deployment.

.metadata.name: Definiert den Namen des Deplyoments.

Spec.replicas: Definiert wie viele Pods des Deployments erstellt werden sollen.

Spec.selector.matchlabels: Labels werden generell als Key:Value Speicher verwendet. In diesem Fall macht es dem Deployment möglich herauszufinden welche Pods zu ihm gehören.

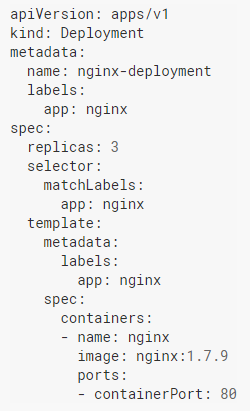
Template.spec.\*: Definiert welches Image, mit welcher Version, auf welchen Container Port verwendet werden soll. Images werden Standartmäßig von Docker geladen, es werden allerdings auch alle anderen gängigen Provider unterstützt.

Abbildung 6: Deplyoment Yaml (Kubernetes)

#### Service

Services sind eine abstrakte Art Anwendungen in Pods für das Netzwerk freizugeben. Sie dienen außerdem als Service discovery Mechanismus, wodurch die Anwendungen nicht verändert werden müssen. Den Pods werden eigene IP Adressen zugewiesen und DNS Namen für Gruppen an Pods verteilt, worüber anschließend eine Lastenverteilung ausgeführt werden kann.

## Vereinheitlichte Systeme

### Service-Mesh

#### Warum sind Service-Meshes entstanden

In einer Microservices Umgebungen bestehen unsere Anwendungen für gewöhnlich aus mehreren über das Netzwerk verbundenen Services. Jeder dieser Services besteht aus einem oder mehreren Containern, welche im Falle von Kubernetes (vorheriges Kapitel) in Pods laufen. Wenn nun Anfragen in unser System eingehen, warten diese für gewöhnlich auf Antworten von einigen Services welche jeweils dynamisch Skaliert sind. Um nun diese Kommunikation reibungsfrei zu ermöglichen und für Entwickler einfacher zu gestalten wurden die Service-Meshes erstellt.

Ein Service Mesh übernimmt also zum einen die Kommunikation zwischen Services und zum anderen hilft es auch beim Managen der Services und Komponenten welche diese ermöglichen. Das Service Mesh hilft also dabei Microservices zu benutzen und die Nachteile möglichst auszugleichen bzw. mit diesen umzugehen um das volle Potenzial einer Microservice Umgebung auszunutzen in dem es das Netzwerk Management einfacher macht.

#### Was macht ein Service Mesh

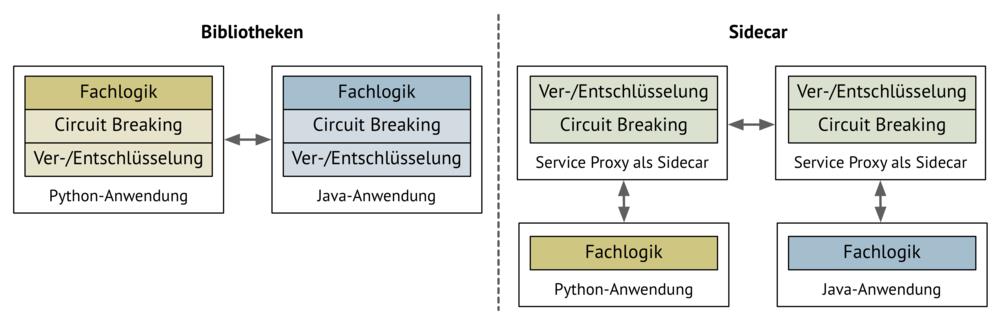
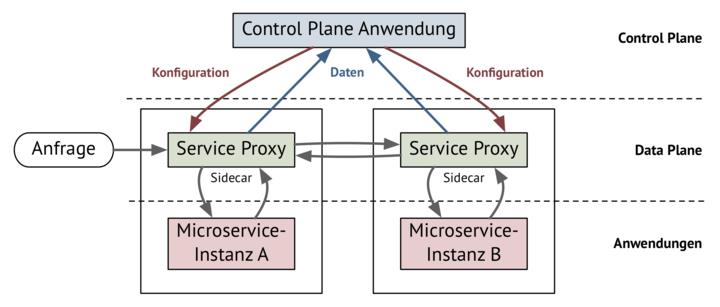
Als Voraussetzung für ein Service Mesh, ist eine Microservice Architektur, da sie genau für diese Umgebung gebaut sind und eine Infrastruktur Komponente wie Kubernetes( Orchestrierung) benutzten können. Diese sind eine allgemeine Voraussetzung für die Service Meshes sind Bsp. Linkerd 2 ist nur mit Kubernetes als Orchestrierung Lauffähig. Das jeweilige Service Mesh baut auf diesen Bestehenden Infrastrukturen auf und Erweitert sie Bzw. Versucht sie einfacher benutzbar zu machen. Ein Service Mesh benutzt hierfür üblicherweise das Sidecar Pattern, welches von dem Englischen Beiwagen stammt und auch sehr ähnlich funktioniert. Hierbei werden die Sidecars an bereits bestehende Container geheftet und haben dem entsprechend auch denselben Lebenszyklus. Sidecars übernehmen sprach-unabhängig unterstützende Funktionen für diese Container. Typische Funktionen welche durch Sidecars abgedeckt werden, umschließen Metriken, Routing, Ver- bzw. Entschlüsselung, Service discovery, Health checks, Robustheit Mechanismen sowie Autorisierung und Authentisierung. Die Sidecars Kommunizieren per Localhost über ein Standard Netzwerkprotokoll mit den Containern und bilden einen weiteren zwischen Hop, wo ihre Funktionen angewendet werden. In Abbildung 7 ist eine Gegenüberstellung zur herkömmlichen Herangehensweise zu sehen mit einer Auslagerung in Sidecars zu sehen.

Abbildung 7: Nutzung von Bibliotheken VS Auslagerung in Sidecars (Prinz 2019)

Alle Services mit ihren Sidecar Proxys werden zusammen als Data Plane bezeichnet und bildet ein von zwei großes Teilen eines Service Meshes. Sidecars sind jedoch keine neue Erfindung, sondern gibt es sie schon eine ganze Weile, was neu ist, ist das sie von einem zentralen Punkt verwaltet und automatisch verteilt werden.

Der Zweite große Teil eines Service Meshes bildet die Control Plane. Die Control Plane bildet das zentrale Gehirn eines Service Meshes, es kommuniziert mit den einzelnen Service Proxys. Das bedeutet die Control Plane verteilt Konfigurationen welche in ihr eingestellt wurden an die Proxys, siehe Abbildung 8 rote Pfeile. Weiterhin sammelt die Control Plane Monitoring Informationen der Proxys ein, siehe Abbildung 8 blaue Pfeile um sie an ein entsprechendes System weiterzugeben wo sie dann Verarbeitet und in z.B. einem Prometeus(eines von möglichen Monitoring Dashboards) Dashboard wiedergeben werden kann. Im Weiteren ist eine Übersicht der Interaktion in der Architektur zu Sehen.

Abbildung 8: Service Mesh Architektur (Prinz 2019)

#### Was spricht gegen die Nutzung von Service Meshes

Service Meshes besitzen noch einige Nachteile, welche jeweils abgewogen werden sollten um Herauszufinden, ob es sich lohnt in diese zu Investieren oder eben nich. Im Moment ist ein größerer Nachteil, dass die jeweiligen Technologien relative Neu sind und erst eingeschätzt werden müssen wie sie sich in kleineren oder vor allem in größeren Projekten beweisen. Die nicht unbedeutende Auswahl an unterschiedlichen Anbietern für Service Meshes macht dies nicht gerade einfacher und eine Auswertung der Technologie unübersichtlicher. Man kann sich ein eigenes Bild z. B. über das Service Mesh Landscape von (Layer5 10/28/2019) machen. Zwei der weiter verbreiteten Service Meshes sind Istio, jetzt in der Version 1.4 und Linkered, jetzt in der Version 2.6. Beide gelten soweit als Produktionsfähig

Ein weiterer Nachteil kommt durch die zusätzlichen Komponenten hinzu, welche die Komplexität der Umgebung erhöhen kann und zusätzliche Kosten verursacht, welche auf das Projektbudget drücken.

Erhöhte Latenzzeit, welche durch zusätzliche Proxys vor jedem Container entsteht, wodurch speziell die innere Kommunikation durch 2 weitere Hops verlangsamt wird.

Für Projekte welche keine Microservice Architektur benutzen, machen Service Meshes ebenso weniger Sinn da sie ja gerade dabei helfen sollen diese zu managen, Stattdessen führt die einführen eines Service Meshes eher zu einem Overhead an Funktionalität und Ressourcen verbrauch.

### Das Service-Mesh Istio

## Istio ist ein Service-Mesh, es wird hier nicht noch einmal darauf eingegangen was ein Service-Mesh ist, da dies bereits im Punkt 3.7 Vereinheitlichte Systeme

Service-Mesh erläutert wurde. Istio ist zuallererst Vollständig **Open-Source** und bringt indem Zuge alle seine Vor und Nachteile mit sich. Durch die Benutzung von Istio ist es möglich services zu Veröffentlichen und sie mit unterschiedlichsten Techniken auszustatten(welche Techniken dies genau beinhaltet wird später erläutert) welche mit wenigen bis keinen Code Änderungen in Kraft treten können

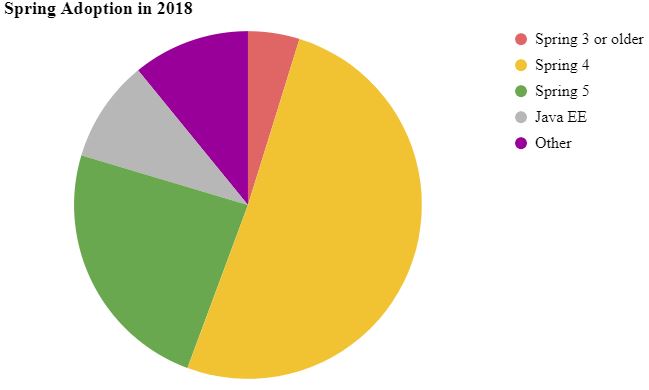
Die im speziellen verwendeten Komponenten für die einzelnen Aspekte bzw. vielleicht auch die Möglichkeiten welche hier verfügbar sind. Ingress …

Wo sind die Unterschiede im speziellen in Istio.

### Spring

#### Warum Spring

Um Microservices zu bauen wird heutzutage sehr häufig das Open Source Framework Spring benutzt(vor allem im Java Sprachraum) „Over the last decade, Spring Framework has become the dominant framework in the Java community“. (Dashora 2019). Dieses Framework hat sich zum Ziel gesetzt die Arbeitsweise mit Java deutlich benutzerfreundlich zu machen und einiges an Arbeit für den Entwickler zu übernehmen. Dies geschieht hauptsächlich durch eine automatische Konfiguration im Hintergrund und den exzessiven Einsatz von Annotationen. Durch diese Leistung wurde eine gute Alternative zum bisher weit verbreiteten und zuvor Präsenten Java Enterprise Edition kurz JEE Framework geliefert. Wie in Abbildung 9 zu sehen ist, hat das Spring Framework das zuvor vorherrschende Java EE Framework größtenteils abgelöst.

Abbildung 9: Spring Annahme (Paraschiv 2018)

Da das Spring Framework unteranderem von Netflix mit entwickelt wurde, welche ebenfalls eines der größten verteilten Systeme weltweit besitzen und es zur damaligen Zeit noch keine etablierten Technologien gab, diese nicht praktikabel oder ausreichend waren um ein robustes System in diesem Ausmaße zu etablieren haben sie hier ihre eigenen Werk-zeuge gebaut. Diese Tools bilden einen wesentlichen Bestandteil dessen, was mit einer herkömmlichen Java Programmierung an Robustheit erzeugt werden kann. Dies beinhaltet eine zum einen Infrastruktur Lösungen wie Service Discovery, Distributed Tracing, verteilte Konfiguration oder Clientseitiges Load balancing. Als auch Funktionen zur Festigung der Robustheit wie Circuit Breaker, Timeouts oder auch Retry Regeln. Fast jede dieser Komponenten, kann im Java Code spezifisch konfiguriert/Code außen herum geschrieben werden und bietet dadurch deutlich mehr Möglichkeiten für Entwickler mit entsprechenden Problemen umzugehen um z. B. Fallbacks im Circuit Breaker einzustellen. Da nie das Java Umfeld verlassen wird fällt es Entwicklern oft deutlich leichter Probleme direkt zu lösen ohne ein neues System wie ein Service Mesh zu erlernen und bietet gleichzeitig auch mehr Möglichkeiten mit Problemen umzugehen als Konfigurationen in einem Service Mesh dies täten.

#### Spring Cloud

Spring Cloud ist eine Sammlung an Werkzeugen und Frameworks, welche Lösungen für gewöhnliche Muster oder Techniken in einem verteilten System bietet. Durch die Verwendung von Spring Cloud können Services und Anwendungen entwickelt werden, welche unterschiedlichste Muster oder Techniken implementieren um in einem verteilten System koordiniert zu operieren. Diese Anwendungen ermöglichen es außerdem in jeglicher verteilten Umgebung gut zu funktionieren, wie der lokalen Entwicklungsumgebung oder einem Datenzentrum. Die Features von Spring Cloud umschließen hierbei Konfigurationsmanagement, Service discovery, circuit breaker, intelligentes routing, micro proxys (side-cars), load balancing und einiges mehr.

### Vergleich Istio gegenüber Spring Cloud

Im Folgenden wird ein Vergleich zwischen Istio und Spring Cloud aufgezeigt. Beide werden später in der Arbeit noch mehr im Detail verglichen, hier wird nun zuerst ein mehr genereller Ansatz verfolgt, um Entscheidungen zu erleichtern und allgemeinere Informationen zu erhalten.

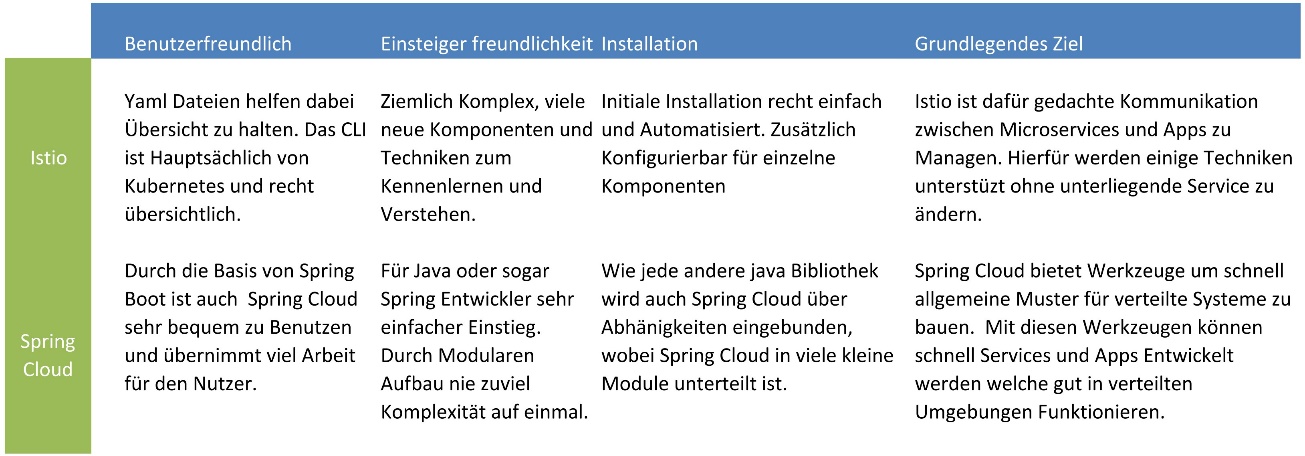


Tabelle 1: Istio oder Spring Cloud 1

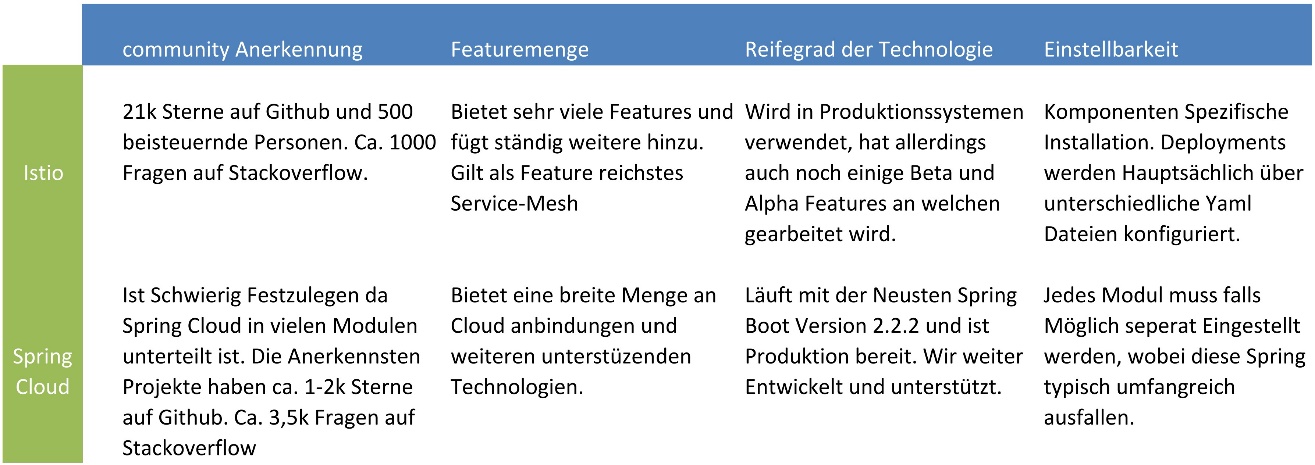


Tabelle 2: Istio oder Spring Cloud 2

Wenn man sich noch nicht mit Istio auskennt, kann es einiges an Arbeit kosten sich einzuarbeiten was einen späteren Wechsel erschwert was auch der Fall ist, wenn bereits Erfahrungen mit ähnlichen Technologien gemacht wurden. Im Gegensatz dazu fällt eine Initiale Einarbeitung bei Spring Cloud nicht so sehr ins Gewicht, falls bereits Erfahrungen aus dem näheren Umfeld vorhanden sind. Es sollte Möglichst früh klar sein auf welcher Plattform die Anwendung laufen soll, um nicht spezifische Komponenten zu erstellen, welche bereits auf der Plattform existieren. Dies ist vor allem deswegen wichtig um zu entscheiden, ob spezifische Lösungen benötigt werden oder passende Optionen bereits vorhanden sind.

## Zusammenfassung

Zum Schluss dieses Kapitels werden nun noch einmal die Kernpunkte aufgegriffen und Festgehalten was für den weiteren Verlauf der Arbeit wichtig ist. Es ist hierfür äußerst wichtig zu Verstehen welche Problematiken ein verteiltes System mit sich bringt und das die Kommunikation hierbei den entscheidenden Unterschied ausmacht. Weiterhin sollte erkannt worden sein welche entscheidende Rolle die Robustheit in einem verteilten System einnimmt um den Problematiken vorbeugend entgegenzuwirken. Zum weiteren Verständnis ist es außerdem vorteilhaft das ein grobes Verständnis der Orchestrierungsplattform generell und im speziellen in Bezug auf Kubernetes vorhanden ist. Um die im weiteren Verlauf folgenden Beispiele zu versehen ist es äußerst relevant einen Überblick über die Service-Meshes und die Spring Umgebung zu besitzen, wobei im Falle der Service-Meshes speziell die Eigenheiten von Istio bekannt sein sollten.

Im Folgenden werden unterschiedliche Möglichkeiten erläutert um die unter anderem hier beschriebenen Probleme in verteilten Systemen bzw. in einer Microservice Umgebung anzugehen und zu lösen oder den Umgang zu erleichtern. Hierbei werden einzelne Möglichkeiten herausgenommen und spezielle Beispiele gegeben.

# Möglichkeiten zum Management und Gewährleisten eines verteilten Systems

Im Folgenden werden Microservices in verteilten Systemen und die hierbei auftauchenden Probleme einzeln untersucht und für die jeweiligen Ansätze falls vorhanden unterschiedliche Lösungsmöglichkeiten vorgestellt. Es wird hierbei besonders auf die Steuerung und Kontrolle der Kommunikation eingegangen bzw. wie diese überhaupt erst ermöglicht wird. Um die im Anschluss kommenden Beispiele nachvollziehen zu können ist sehr zu empfehlen sich zuerst mit den Theoretischen Grundlagen in den vorherigen Kapiteln auseinander zu setzen um den Rahmen zu verstehen und ein Verständnis für die unterliegenden Systeme zu erhalten.

## Kommunikations-Leitung

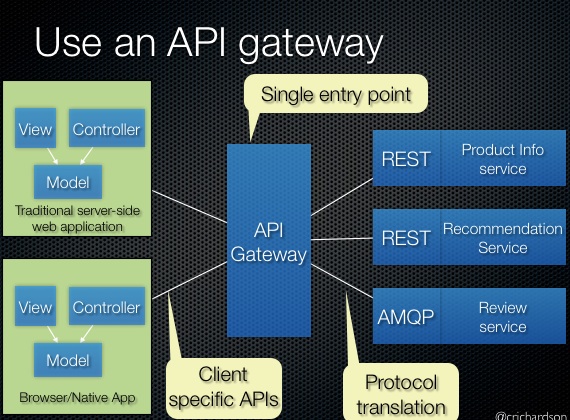
### Einleitung

In einer Microservice Architektur werden Informationen von vielen unterschiedlichen Punkten bezogen, um dies bei einer wachsenden Anzahl an Services weiterhin ermöglichen zu können, wird eine oder mehrere Komponenten benötigt diese Kommunikation möglichst automatisiert zu Steuern. Des Weiteren kommunizieren externe Nutzer mit APIs unserer Anwendung wodurch es Schwierig wird diese im Ort zu verschieben oder umzubenennen, da die Nutzer an diese fest gebunden sind. Die gesamte Kommunikation sollte Protokoll und Sprachlich unabhängig passieren. Für diese Probleme gibt es unterschiedliche Ansätze in der Architektur und etliche Anbieter worüber später ein grober Überblick geben wird.

### API Gateway

#### Was ein API Gateway erreichen soll

* Services können unterschiedliche, auch Web inkompatible, Protokolle nutzen siehe Abbildung 10.
* Clients benötigen Daten von vielen unterschiedlichen Quellen/Microservices, was dem Client allerdings nicht bekannt sein soll (keine Informationen über internen Datenverkehr) siehe Abbildung 10 rechts.
* Eine Client entsprechende Daten Bereitstellung sollte Erreicht werden (Desktop vs. Mobile) siehe Abbildung 10 links.
* Services können sich sowohl in der Anzahl als auch im Ort bzw. der Adresse ändern und sollen vom Clients versteckt sein, demnach keine direkte Verbindung zu ihm haben.
* Es soll auf Hardware/Netzwerk Limitierungen, von unterschiedlichen Arten von Clients, mit Entsprechenden API reagiert werden können, welche z.B. längere Wartezeiten zulassen (Abbildung 10 im Gateway).

Abbildung 10: API Gateway Übersicht (Richardson 11/20/2019)

#### Ein Gateway als Lösung

Das Gateway dient als einziger Eingangspunkt für alle Nutzer, ist dies nicht der Fall können Nachrichten unbemerkt an die APIs geladen. Hierbei übernimmt das Gateway den gesamten Nachrichtenverkehr in beide Richtungen, dieser wird entsprechend der Einstellung des API Gateways, nur zu den entsprechenden Services durchgeschleift oder an mehrere Services ausgefächert. Das API Gateway kann jedem Nutzer eine andere API zur Verfügung stellen und dadurch die entstehenden Bedürfnisse spezifischer handhaben. Es ist auch, entsprechend abhängig von welchem Anbieter das Gateway stammt möglich, Autorisierung, Authentisieren, Logging, Monitoring usw. in dem Gateway zu verwenden und somit die Services zu entlasten bzw. Dritt Software nicht zu benötigen.

#### Backends for Frontends

Einen etwas anderen Ansatz nimmt die Variante welche „Backends for Frontend“ genannt wird. Für jedes Frontend (sowohl Web Client als auch Mobile oder 3rd Party) wird ein eigener Gateway erstellt welcher auch nur für dieses verantwortlich ist siehe Abbildung 11. Das hat zum Vorteil, dass es zum einen keinen alleinigen Flaschenhals mehr gibt, zum anderen das die Gateways zweckmäßig getrennt und eine klarere Struktur haben. Durch ihre erhöhte modulare Gestaltung können sie außerdem besser Skaliert werden und können klarer auf die Bedürfnisse ihres speziellen Clients eingehen. Zusätzliche Funktionalitäten sind natürlich immer noch möglich.

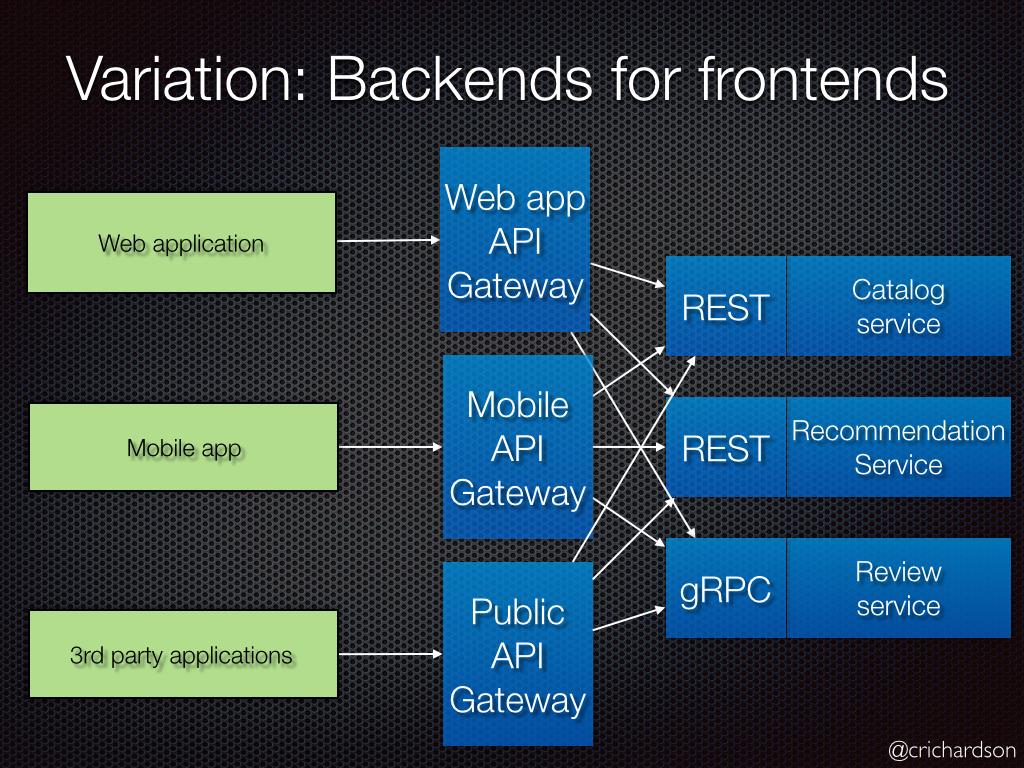


Abbildung 11: API Gateways Backends for frontends (Richardson 11/20/2019)

### API Microgateway

Der API Microgateway, ist üblicherweise ein Proxy (Kommunikationsschnittstelle), entweder extern oder als sidecar, welcher vor einem Microservice sitzt. Er bietet die Möglichkeit Regeln und Sicherheitsabfragen gezielter am Service durchzuführen, den Datenverkehr direkt am Service zu überwachen, Service discovery(Erkennung) zu übernehmen und die Stabilität im Allgemeinen zu erhöhen. Solange das Gateway, micro also klein genug bleibt und nicht zu groß und schwer wird können beliebige Funktionalitäten hinzugefügt werden. Man sollte immer bedenken, dass der Microgateway in den Häufigsten fällen im selben Bereich wie der Service läuft und infolgedessen, dessen Startzeit mit beeinträchtigt. Doch durch diese enge Bauweise wird er auch sehr wiederverwendbar, welche direkt zu einer besseren Skalierbarkeit führen kann. Hierbei werden nicht die API Gateways, welche am äußeren Rand des Systems sitzen ersetzt, sondern Kommunizieren vielmehr mit diesen und erhalten Sicherheit Informationen über z. B. API Services. Der API Microgateway wird inzwischen vor allem in Service Meshes wie zuvor Erklärt verwendet und findet dadurch eine große Anwendungsfläche.

## Kommunikations- Überwachung und Verfolgung

### Einleitung

Bereits vor der Entstehung der Microservice Architektur wurden verteilte Systeme so groß, dass die Übersicht über diese verloren ging bzw. die Durchsuchung immer unproduktiver wurde. Dies beinhaltet zum einen, einen Überblick über die aufgerufenen Services eines einzelnen Aufrufs, zum anderen den Überblick über die unterschiedlichen APIs in einem komplexen System welche sich in Versionierung und Typ unterscheiden. Es werden neue zentrale Punkte benötigt, welche sich mit diesen Problematiken beschäftigen und den Entwicklern einen Zugangspunkt gewähren, durch welchen sie die Möglichkeit haben diese auf einfach weiße zu managen und den Überblick wieder herzustellen.

### API Management

#### Teilen zu unseren Bedingungen

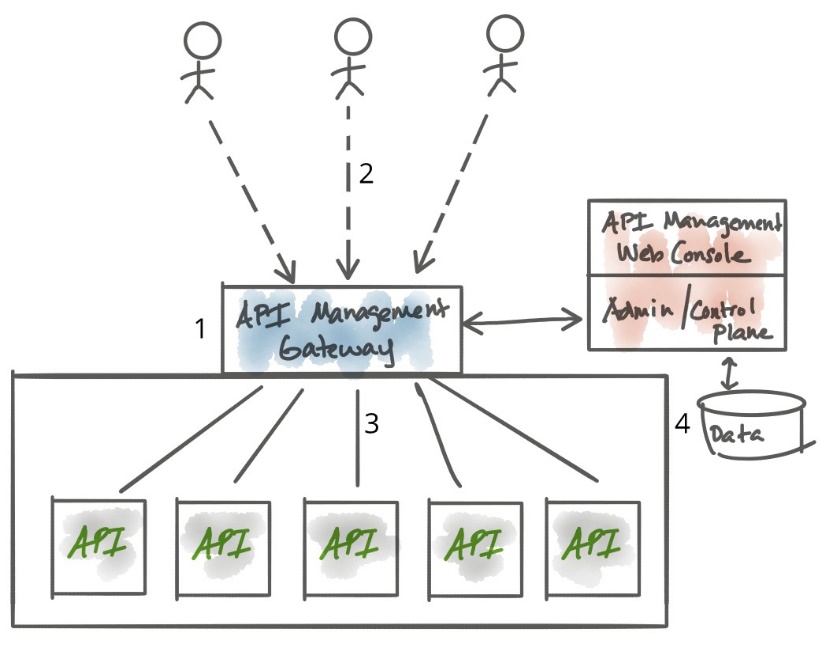
Einen der zuvor erwähnten zentralen Punkte bildet der API Management Gateway. Dieser Management Gateway deckt entsprechend dem Anbieter gewisse Funktionalitäten ab und soll das Leben von Entwicklern einfacher machen in dem es z.B. APIs verwalten lässt.

We want to solve the problem of “we have these existing, curated, APIs that we want to share with others but share them on our terms”. (Posta 2019)

#### Das API Management Gateway

Der API Management Gateway ist eine Komponente, welche eine weitere Schicht Bildet oder den bisherigen Gateway komplett ersetzt (dies kann auch auf von der Größe des Systems abhängen). Er managt, wann existierende APIs für Konsumer erreichbar/benutzbar sind und notiert dessen Nutzung, etabliert Regeln und hält fest, für wen sie gelten, der Sicherheitsfluss wird festgelegt und ergibt Freigaben für die Nutzung (Autorisierung, Authentisierung). Alle APIs Abbildung 12 [3] werden von diesem Gateway gemanagt, katalogisiert und verwaltet, wodurch diese vergeben, gefunden und effektive kontrolliert werden können.

Die Managementkomponente kann als Einstiegspunkt in unser System dienen wie es in Abbildung 12 [2] zu sehen ist, muss dies allerdings nicht. Es kann so auch ein dezentralisierter Ansatz verwendet werden, wodurch Gateways direkt vor jede API gesetzt werden und somit das System besser Skalierbar ist.

Abbildung 12: API Management Gateway Overview (Posta 2019)

Ein Punkt bei dem hierbei aufgepasst werden muss, ist keine Businesslogik in diese Schicht Abbildung 12 [1] einfließen zu lassen. Da das API Management eine geteilte Komponente ist, hat es Tendenzen dazu ein all wissend, all verarbeitendes Konstrukt zu werden, wo jeglicher Datenverkehr für Änderungen an APIs durchfließen muss, was erneut in einem organisatorischen Flaschenhals enden kann. Da auch die Management Komponente Skaliert werden muss sollte hier möglichst versucht werden keine Abhängigkeiten zu anderen Komponenten zu erzeugen und genau aus diesem Grund auch eine eigene lokale Datenbank zur Verfügung zu stellen Abbildung 12 [4].

### Distributed tracing

#### Was ist Distributed tracing

Bei distributed tracing geht es darum Anfragen, über mehrere Systeme hin, zu Verknüpfung. Die Verknüpfung geschieht durch zugehörige eindeutige IDs, welche im http Kopf mitgeliefert werden. Die IDs werden in Zusammenhang gebracht umso denn gesamten weg einer Anfrage nachvollziehen zu können. Das heißt es ist leichter möglich herauszufinden wo ein Fehler tatsächlich entstanden ist und welche Systemkomponente verantwortlich ist.

#### Wieso ist Distributed tracing wichtig

Mit der Architekturänderung von Monolithen zu Microservices haben sich einige neue Herausforderungen ergeben. So sind wenige lokale Module in viele verteilte Services umstrukturiert worden und was damals noch lokal gedebugt werden konnte, verteilt sich nun über das Netzwerk hinweg über etliche Hops.



Abbildung 13: Distributed Tracing (Kyma)

Da niemand gerne viel Zeit damit verbringt sich durch etliche Netzwerklogs durchzuwühlen wird eine automatisierte Lösung benötigt. Um genau dieses Problem zu beherrschen benötigt es neue Analysewerkzeuge und hier kommt Distributed Tracing ins Spiel.

#### Nachteile

Wie so vieles, hat auch distributed tracing einige Nachteile. Die anfallenden Logs können immense Groß werden und sind fast nicht komprimierbar was vor allem an ihrer notwendigen Einzigartigkeit liegt und je nach Systemgröße sehr groß/viele werden können „Tracing is high-volume and high-cardinality“ (McAllister 2019, 6:5-6:10). Es gibt einige „Standards“ (anstatt einen)was dazu führen kann, dass es einen Anbieterwechsel besonders schwer macht, "When you follow widely adopted standards you get to avoid vendor lockin which is actually pretty important inside of this space" (McAllister 2019, 15:15-15:25). Eine Codeanpassung für Distributed tracing kann sehr umfangreich sein sodass es niemand zweimal durchführen möchte. "Trust me instrumenting all of your code once is more than enough" (McAllister 2019, 15:37-15:42).

## Gewährleistung von Services

### Einleitung

Mit einer steigenden Anzahl an Microservices und der damit erhöhten Kommunikation entstehen unweigerlich neue Probleme. Um diesen vorbeugend entgegenzuwirken und vollständig einsatzbereite robuste Systeme zu erreichen, benötigen wir neue Techniken welche im Folgenden genauer betrachtet und die damit verbunden Probleme beleuchtet werden.

### Resilient4j VS Failsafe

Für die Implementierung in Java wird hierbei auf Resilient4j gesetzt und im Folgenden, zweierlei Alternativen aufgezeigt sowie vor und Nachteile dieser.

Resilient4j ist hierbei jenes neue Bibliothek welches in Zukunft die vermutlich größte Akzeptanz erfahren wird und somit denn bisherigen Platzhirsch Hystrix von Netzflix ablöst, da dieses nicht länger aktive weiter Entwickelt wird.

Eine Alternative zu Resilient4j bietet hierbei Failsafe.

* Failsafe hat hierbei keine weiteren Abhängigkeiten welche später zu Problemen führen können (unterschiedliche Versionen oder Überschneidungen mit Eigenen Abhängigkeiten), wohingegen Resilient4j eine Abhängigkeit auf Vavr besitzt.
* Failsafe bietet die Möglichkeit über ein SPI(Service Provider Interface) zusätzliche Regeln zu implementieren.
* Failsafe lässt asynchrone Aufrufe jederzeit abrechen und unterbrechen.
* Failsafe unterstützt asynchrone API Einbindungen, wodurch die Vollendung von Fremden Code immer noch durch Callbacks bestätigt wird.
* Beide Bibliotheken bezeichnen sich als Leichtgewichtig.
* Beide werden von einigen Firmen verwendet, wobei Failsafe hier deutlich mehr aufzählt.
* Beide Bibliotheken sind Opensource allerdings sind an Resilient4j ca. 5-8 Personen mit größerem Anteil Beteiligt, wohingegen bei Failsafe dies allerhöchstens 2 Personen sind und von einer Person getragen wird.
* Resilient4j hat 5k Sterne auf Github wohingegen Failsafe 3,1k Sterne besitzt.
* Resilient4j bietet eine deutlich ausführlichere Dokumentation.
* Resilient4j bietet spezielle Möglichkeiten in andere Bibliotheken wie Spring Boot, welche die Nutzung vereinfachen.
* Resilient4j stellt Einstellungen über Konfigurationsdateien ein und unterscheidet diese im Namen.
* Resilient4j bietet zusätzliche Robustheitsmöglichkeiten wie das Bulkhead und das RateLimiting.

#### Einschätzung

Diese beiden Bibliotheken unterscheiden sich hierbei vor allem in dem Ansatz, dass Resilient4j eine breitere reihe an Möglichkeiten bietet und diese einfacher benutzbar macht und Failsafe auf der anderen Seite einen Ansatz wählt welcher mehr Eigenentwicklung benötigt aber auch mehr Freiraum gibt, wobei asynchrone Programmierung besonders stark unterstützt wird.

### Bulkhead

#### Was sind Bulkheads

Bulkhead ist ein Begriff, welcher aus dem Schiffsbau kommt und beschreibt eine Technik, bei der das Schiff in Segmente unterteilt wird. Bei einem Leck können diese Segmente separat geschlossen werden, um somit zu verhindern, dass das gesamte Schiff voll Wasser läuft siehe Abbildung 14 und Abbildung 15. Feuerschutztüren im Brandschutz dienen demselben Zweck und verhindern die Rauchverbreitung im Haus.

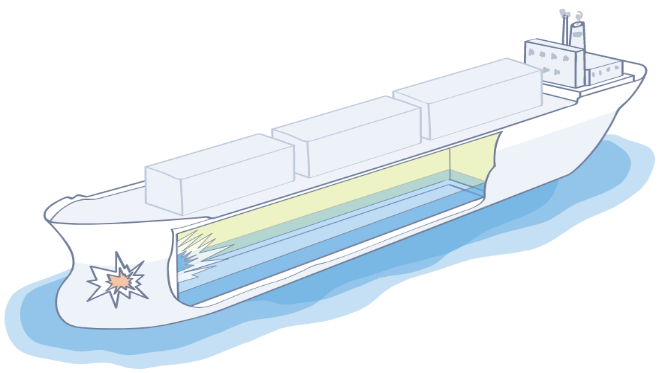
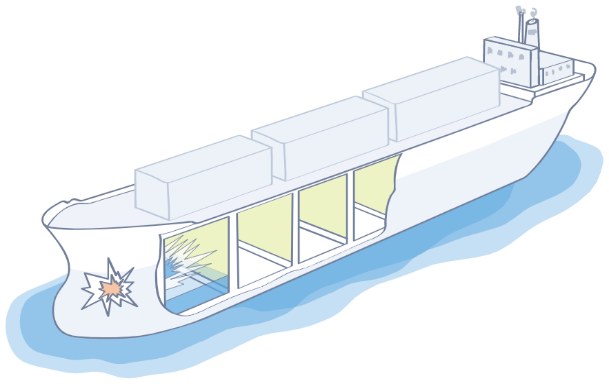
Abbildung 14: Schiff Links Ohne Bulkhead (IBM 11/28/2019)

Abbildung 15: Schiff Rechts mit Bulkhead (IBM 11/28/2019)

So wie im Schiffsbau und im Brandschutz, wird das Bulkhead Pattern (in Deutsch Schott Muster) auch in der Microservice Architektur verwendet, um zu verhindern, dass das gesamte System durch einen einzigen Fehler zum Einsturz gebracht wird. Dies geschieht im Falle von Microservices auf die Art und Weise, dass einzelne Bahnen im Prozessablauf von Beginn an voneinander Technisch getrennt/isoliert werden. Das bedeutet, dass ein Service nicht auf dieselbe Warteschlange oder Threadpool zugreift wie ein anderer Service sondern diese separat zugewiesen werden.

Durch dieses Pattern wird einerseits zwar der Schaden nicht verhindert, er wird allerdings davon abgehalten sich auszubreiten wodurch alle anderen Services weiterlaufen können. Andererseits dient es dazu nur jeweils so viel Arbeit auf einmal abzuarbeiten wie dies Eingestellt ist, um immer reagieren zu können. Wenn man beim Bild des Schiffes bleibt, wiederspiegeln die Kammern jeweils einen Service. Jede Kammer hat eine gewissen Anzahl an Ressourcen und jeder Service kann nur so viel Einnehmen/Fluten wie seine Kammer hergibt/bereitstellt.

#### Microservices Probleme ohne Bulkheads

Wenn in einem Softwaresystem ein Fehler verursacht wird stürzt in der Regel das gesamte System ab. In einem verteilten System heißt dies, dass wenn ein Fehler in einem Teilsystem verursacht wird, alle verbundenen Services oder gar das gesamte System abstürzt. Am Beispiel der Abbildung 16 wirft der Dienst A Fehler wodurch dieser Anfragen sofort beantwortet und erneut Anfragen erhält. Durch diesen Prozess werden jegliche Ressourcen des Threadpools und damit der CPU vereinnahmt wodurch andere Dienste keine Möglichkeit mehr haben selbst zu Operieren.

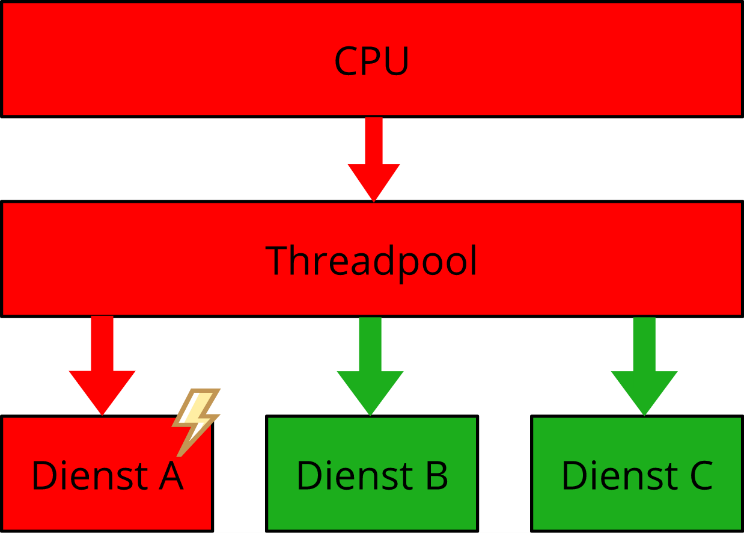


Abbildung 16: Fehlerhafter Service ohne Bulkheads (Eigenkreation)

#### Microservices mit Bulkheads

Abbildung 17: Fehlerhafter Service mit Bulkhead (Eigenkreation)

Wenn Services mit der Bulkhead Technik ausgestattet sind sieht das verhalten mehr wie in Abbildung 17 zu sehen ist aus. Hier wurde jeder Service mit einem eigenem Threadpool ausgestattet, wodurch zwar einer dieser Pools ausgeschöpft und damit Entsprechend viel Last auch auf die CPU übertragen wurde, die Restlichen Services jedoch können zur selben Zeit ohne Beeinträchtigung weiterlaufen.

#### Prinzipien des Bulkhead Patterns

1. Teile möglichst nichts:

Soweit es möglich ist, sollte, wenn Services in eigene Fehlerzonen isoliert werden keine Datenbanken, Firewalls, Speicher und Rechenleistung etc. teilen. Aus Kostengründen kann man es auch nur auf die Services heruntergebrochen werden.

1. Vermeide synchrone Aufrufe zu anderen Services:

Synchrone Service zu Service Kommunikation erweitert die Fehlerzone eines Bulkheads. Es können Fehler und Trägheit mit synchronen Aufrufen übertragen werden und somit den Schutz welcher durch ein Bulkhead gewährt wird Verletzt werden.

#### Umsetzung

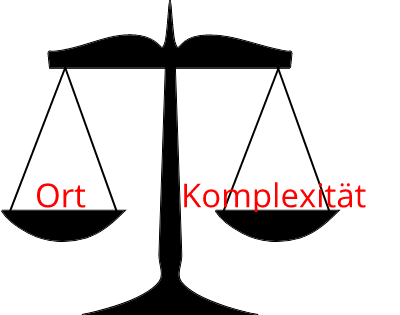
Es gibt verschiedene Möglichkeiten ein Bulkhead in ein Softwaresystem einzubauen. Dies unterscheidet sich einmal im Ort der Implementierung und zum anderen in der Komplexität der Implementierung. Verschiedene **Orte** wären in diesem Falle eine direkte Implementierung im Code, eine Nutzung von Drittanbieter Software, eine Einschränkung durch Firewall regeln oder durch Konfiguration in einem Service Mesh.

Abbildung 18: Implementierungswaage (Eigenkreation)

Die **Komplexität** der Implementierung meint hierbei wie spezifisch auf das Geschehen des Prozesses Einfluss genommen werden kann. Diese beiden Eigenschaften sind oft wie zwei Seiten einer Wage Abbildung 18, falls es bevorzugt ist eine Implementierung direkte im Code durchzuführen so kann man Automatisch auch spezifischere Lösungen Produzieren. Ist ein Ansatz in einem höheren Level gewünscht, wodurch auch weniger Code selbst geschrieben werden muss, so kann oft auf bereits bestehenden Funktionen zurückgegriffen werden. Durch die Verwendung von Funktionen in höheren Ebenen wird die Umsetzung allerdings auch automatisch limitiert, so können teilweise nur Funktionen wie Ratelimiting eingebunden werden. Da kein direkter Zugriff auf Prozessinterne Ressourcen besteht, stellt sich hierbei auch die Frage, kann überhaupt von einem Bulkhead die Rede sein, wenn eine Anwendung nicht in unterschiedliche Ressourcen abschnitte getrennt werden kann.

##### Beispiel: Eine Code Umsetzung in Java mit dem Resilient4j.

Wie man im Code Beispiel Codeteil 1 im Ersten Abschnitt zu sehen ist, sind in einem in Java Implementierten Bulkhead sehr spezifische Änderungen möglich wodurch zum einen dieser Service in seinen eigenen Threadpool eingeschlossen wird und zum anderen dieser Threadpool durch spezifische Einstellungen eingeschränkt werden kann. Durch diese direkte Implementierung können solche spezifische Eigenschaften für jeweilige Routen/APIs festgelegt werden ohne die Bekannte Entwicklungsumgebung bzw. Sprache zu verlassen. Darüber hinaus hilft Resilient4j dem Entwickler noch dabei effizienteren Code zu schreiben indem es, falls ein Thread Pool Bulkhead verwendet wird, asynchrone Kommunikation erzwungen wird welche in einer Microservice Umgebung bereits Standard seien sollte. Falls man die Synchrone Semaphore Version des Bulkheads verwendet fallen allerdings auch einige Einstellungsmöglichkeiten weg, wodurch man nicht einen so genauen Einfluss nehmen kann.

ThreadPoolBulkheadConfig threadPoolBulkheadConfig = ThreadPoolBulkheadConfig.*custom*()  
 .maxThreadPoolSize(4)  
 .coreThreadPoolSize(2)  
 .queueCapacity(8)  
 .keepAliveDuration(Duration.*ofMillis*(100))  
 .build();  
  
ThreadPoolBulkheadRegistry threadPoolBulkheadRegistry = ThreadPoolBulkheadRegistry.*of*(threadPoolBulkheadConfig);  
**threadBulkhead** = threadPoolBulkheadRegistry.bulkhead(**"threadBulkhead"**);  
  
Runnable runnable = ()-> **todoRepository**.findAll().forEach(todo -> todos.add(todo.getTodo()));  
**­­­­threadBulkhead**.executeRunnable(runnable);

Codeteil 1: Bulkhead Serverseitige Implementierung in Java mit resilience4j in Plain Java

io.github.resilience4j.bulkhead.BulkheadFullException: Bulkhead 'threadBulkhead' is full and does not permit further calls

Codeteil 2: Bulkhead Fehlermeldung wenn der Threadpool vollgelaufen ist.

@GetMapping(path = **"resilience"**)  
@Bulkhead(name=**"BACKEND"**,type = Bulkhead.Type.***THREADPOOL***)  
**public** CompletableFuture<ModelAndView> getItems()  
{  
 String MethodName = **"Standard"**;  
 **return** CompletableFuture.*completedFuture*(getTodos(MethodName));  
}

Codeteil 3: Resilient4j mit Spring Boot 2.

Durch die Verknüpfung von Resilient4j mit Spring Boot 2, wird die Verwendung der resilient Mechanismen um einiges einfacher als zuvor im reinen Javacode wie in Codeteil 3 zu sehen ist. Die alleinige Voraussetzung zur Nutzung des Threadpool Bulkheads anstelle des Semaphoren Bulkheads ist die, das ein CompletableFuture Objekt benutzt wird, was sehr einfach realisierbar ist indem es von der Methode zurückgegeben wird. Das CompletabelFuture Objekt wird in Java für asynchrone Verarbeitung verwendet und erleichtert dies seit Java 8 mit einer reichhaltigen API. Die Einstellungen für das Bulkhead, werden in einer application.properties oder .yml Datei festgelegt und bieten viel Spielraum zum Einstellen der Funktionalität. Falls man mehrere resilient Funktionalitäten verbinden möchte, geschieht dies von Haus aus in folgender Reihenfolge:

Retry ( CircuitBreaker ( RateLimiter ( Bulkhead ( Function ) ) ) ).

##### Bulkhead in Kubernetes/Istio

Beide Fälle befinden sich in der Netzwerk-Ebene, wodurch ein Eingreifen in Prozesse einer Anwendung praktisch unmöglich wird. Dementsprechend handelt es sich, wenn immer von Bulkheads im Zusammenhang mit einem der Beiden gesprochen wird nicht um Bulkheads, sondern viel eher um ein Rate Limiting.

Rate Limiting wiederum wird später in 4.3.6 Rate Limiting genauer betrachtet.

### Circuit breaker

#### Kurzschlüsse in Softwaresystemen

Kurzschlüsse geschehen, wenn zu viel Last/Strom auf einer elektrischen Leitung ist und diese infolge dessen durchbrennen würden, was zu einem Feuer führen kann. In einem Softwaresystem sieht das etwas anders aus, Kurzschlüsse geschehen hier wenn so viel Last an einem Service anliegt das dieser zum Absturz gebracht wird und dieser infolge dessen versucht neu zu starten. Während der Startphase des Services wird nun weiterhin so viel Last auf den Service ausgeübt das dieser sofort wieder Abstürzt und sich nicht wieder erholen kann wodurch ein dauerhafter Kurzschluss oder nicht erreichbarer Service entsteht.

#### Was sind circuit breaker

Circuit breaker, in Deutsch Sicherung, kommen ursprünglich aus dem elektrischen Bereich. Sicherungen sind kleine Drähte oder Widerstände welche bei einer gewissen Leistung durchbrennen, bevor Hauptleitungen z. B. in der Wand durchbrennen und so ein Feuer auslösen können. Dies hat ursprünglich öfters zu Hausbränden geführt. Genauso werden sie als Notschalter benutzt, um größere Schäden zu verhindern.

Dieses Prinzip, würde nun anhand eines Programmiermusters in der Software übernommen, umso eine erhöhte Stabilität zu gewehrleisten. Wie in der herkömmlichen Art und Weiße, gibt es auch hier mehrere zustände des „Schaltkreises“. Er kann Offen, geschlossen oder Zusätzlich halb-offen sein. Die Einzelnen Zustände werden nun anhand des Folgenden Bildes (siehe Abbildung 19) erklärt.

**Closed** ist zuallererst der Zustand, welcher den Normalen betrieb widerspiegelt [1], dieser wird nur verlassen, wenn eine entsprechende Anzahl an Fehlern in einer festgelegten Zeit überschritten wird ([2] threshold). Verlassen bedeutet in den Open Status zu wechseln [3]. Hierbei ist es wichtig zu unterscheiden um welche Arten von Fehlern es sich dabei Handelt. So kann es sich z.B. nur um Verbindungsfehler, um Zeitüberschreitungen oder andere Fehler handeln.

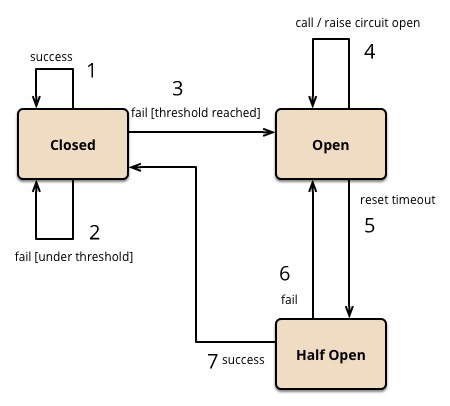
**Open**, blockt erst einmal jeglichen Datenverkehr [4] und schaltet erst nach einer gewissen Zeit auf Half-Open um [5]. Hierbei werden sofort Fehler Meldungen zurückgesendet. In dieser Zeit soll sich der Service, von dem ankommenden Datenverkehr erholen können oder die Zeit haben neu zu starten. Die Wiederherstellung der Verbindung wird in der Regel von externen System übernommen und findet nicht im Circuit breaker statt.

Abbildung 19: Circuit Breaker Zustände (Fowler 2014)

**Half-Open** testet den Service. Es werden hierbei ein paar Anfragen angenommen. Die Anzahl ist normalerweise geringer als im Ursprünglichen Closed Status. Wenn dieser Test Erfolgreich verlaufen ist, wird wieder in den Closed Zustand gewechselt [7]. Falls die Anfragen weiterhin Fehlschlagen wird wieder zurück in den Open Status gewechselt [6].

#### Unterschiedliche Arten/Ebenen von Circuit breakern

Man kann im Allgemeinen in Zwei unterschiedliche Arten von Circuit Breakern unterscheiden. Diese unterscheiden sich je nachdem in Managementaufwand und möglicher Einflussname, welche mit erhöhtem Programmieraufwand Hand in Hand geht.

#### Zentral Verwaltete Circuit breaker

Beschreibt einen von einer Middleware verwalteten Circuit breaker. Normalerweise ist dies ein API Gateway, ein Service Mesh oder ein Reverse Proxy. In diesen Fällen geht der gesamte Datenverkehr durch diese Middleware und wird dort weitergeleitet. Das größte Problem ist die Gefahr, dass dies zu einer einzelnen Schwachstelle für das gesamte System werden kann. Die Infrastruktur Architekten sollten hier immer besonders darauf achten das diese Schwachstellen möglichst Ausfallsicher betrieben werden. Auf der anderen Seite hilft es den Entwicklern, da sie sich weniger mit diesem zusätzlichen Konstrukt beschäftigen bzw. dieses Entwickeln müssen.

#### Unabhängig Verwaltete Circuit breaker

Ein Unabhängig Verwalteter Circuit breaker steht für einen im Service selbst implementierten Circuit breaker welcher den Ankommenden Datenverkehr Regelt. Ein großer Vorteil hierbei ist, dass man auf diese Art und weiße keine einzelnen großen Schwachstellen mehr hat, da diese nach unten auf die Services verschoben werden, was speziell in hoch Verfügbaren Systemen Wichtig wird. Der Nachteil ist die erhöhte Komplexität welche für die Entwickler hinzu kommt, was vor allem der Fall ist wenn unterschiedliche Service in verschiedenen Programmiersprachen geschrieben werden oder Entwickler noch neu in dem Bereich der weiterführenden Fehler sind. Es gibt allerdings in allerlei Sprachen, gut geschriebene Bibliotheken welche weit verbreitet sind und Benutzt werden, was dabei hilft diese Probleme zu bändigen.

#### Umsetzung

Da Circuit breaker mit Nachrichten arbeiten, können diese auf fast allen gängigen Ebenen eingesetzt werden. Hierbei unterscheiden sie sich dadurch wo sie ausgeführt werden, wie spezifisch diese Eingestellt werden können und wie einfach die Verwendung ist. Es kann schlau sein einen Circuit breaker möglichst früh im eigenen Netzwerk einzubauen um darunterliegende Systeme zu schonen.

##### Eine Code Umsetzung in Java mit Resilient4j und Spring Boot 2.

Um Resilient4j mit Spring Boot 2 benutzen zu können, müssen lediglich einige Abhängigkeiten erfüllt werden sowie die entsprechenden Methoden mit CircuitBreaker Annotation und entsprechenden Namen versehen werden siehe Codeteil 4: Methode welche mit Resilient Annotationen des Circuit breakers versehen ist.Codeteil 4. Die jeweiligen Namen sowie Einstellungen für jegliche Eigenschaften werden in einer application.properties oder application.yml Datei festgelegt wie im Codeteil 5 zu sehen ist, wobei BACKEND den Namen des Circuit breakers wiederspiegelt.

@GetMapping(path = **"cb"**)  
@CircuitBreaker(name=**"BACKEND"**)  
**public** ModelAndView getItems()  
{

Codeteil 4: Methode welche mit Resilient Annotationen des Circuit breakers versehen ist.

**resilience4j.circuitbreaker.instances.BACKEND.wait-duration-in-open-state**=**500  
resilience4j.circuitbreaker.instances.BACKEND.permitted-number-of-calls-in-half-open-state**=**2**

Codeteil 5: Circuit breaker Einstellungen in der application.properties

##### Circuit breaker in Istio

Für die Anwendung in Istio gehen wir davon aus, dass die Pods automatisch mit sidecars Injiziert werden und keine gegenseitige Authentifizierung (mutual TLS) aktiviert ist.

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

  name: httpbin

spec:

  host: httpbin

  trafficPolicy:

    connectionPool:

      tcp:

        maxConnections: 1

      http:

        http1MaxPendingRequests: 1

        maxRequestsPerConnection: 1

    outlierDetection:

      consecutiveErrors: 1

      interval: 1s

      baseEjectionTime: 3m

      maxEjectionPercent: 100

Codeteil 6: Destination Rule für einen Service in Istio

Im Codeteil 6 sieht man eine yaml Datei welche eine Destination Rule für einen Service definiert. Der Servicename ist in diesem Fall httpbin und läuft under den zuvor beschriebenen Parametern. Es gibt keinen Wort wörtlichen Circuit breaker in Istio, weshalb regeln welche z.B. in Spring durch die application.properties Datei festgelegt wurde hier nun direkt festgelegt wird. Circuit breaker werden in Istio außerdem direkt für einen ganzen Services festgelegt und können nicht für spezifische Route definiert oder spezifiziert werden.

### Retry

#### Das zu Lösende Problem

Wenn Anwendungen über das Netzwerk miteinander kommunizieren, kann es immer wieder vorkommen, dass gewisse Fehler auftreten. Anfragen können verloren gehen, das Ziel kann momentan nicht erreichbar sein oder ein Service vorübergehend ausgelastet sein. Solche Fehler verschwinden mit der Zeit für gewöhnlich von alleine. Wir brauchen allerdings einen Mechanismus, der dies Möglichst einfach umsetzbar macht, ohne zu große weitere Probleme zu verursachen.

#### Lösungsansatz für das Problems

Es steht nun also fest, das Fehler immer wieder auftreten und das wir mit ihnen umzugehen haben. Wenn unsere Anwendung also feststellt, dass ein Fehler aufgetreten ist, als sie versucht hat einen Aufruf auszuführen haben wir eine Reihe an Möglichkeiten mit diesem umzugehen:

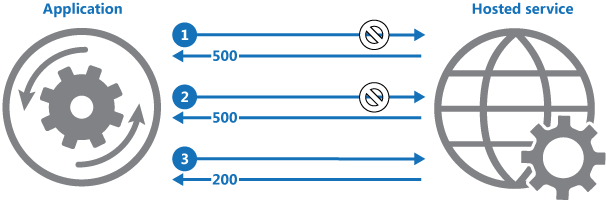
* Abbruch: Falls erkennbar ist das der Fehler sich nicht mit einer Wartezeit beheben lässt oder mit dem Kommunikationsweg zu tun hat, sollte die Aktion abgebrochen werden und eine Exception ausgelöst werden. Wenn Beispielsweise ein Login mehrfach mit den Falschen Daten ausgeführt wird, macht es keinen Sinn diesen überhaupt abzusenden.
* Retry: Wenn der Fehler nur selten vorkommt und keine Rückschlüsse auf konkrete Fehler aufweist, sollte dieser einfach sofort wiederholt werden, da hier mit erhöhter Wahrscheinlichkeit das Netzwerk die Tatsächliche Fehlerquelle ist, was bedeutet das der Fehler nicht nochmal auftreten wird Bsp. Abbildung 20.

Abbildung 20: Retry Pattern (Microsoft)

* Retry mit Verzögerung: Falls es sich um einen eher gewöhnlicherer Fehler handelt, z. B. der Angefragte Service oder das Netzwerk sind Überlaufen. Ist die beste Option die Anfrage mit ein gewissen Verzögerung erneut zu Senden.

#### Umsetzung

Die Retry Funktion von Resilient4j in Verbindung mit Spring Boot 2 ist sehr einfach einzubinden, bietet auch einiges an zusätzlichen Einstellungen, welche bei anderen alternativen so nicht vorhanden sind.

@Retry(name=**"BACKEND"**,fallbackMethod = **"secondFallback"**)  
@GetMapping(path = **"retry"**)  
**public** ModelAndView getItems()  
{

Codeteil 7: Method mit Resilient4j Retry

Istio Retry, Kubernetes Retry

### Rate Limiting

Rate Limiting Komponenten sind Circuit Breakern ziemlich ähnlich, in der Weise das sie die ankommenden Anfragen limitieren. Doch anders als der Circuit Breaker, wird die Auswirkung eines Rate Limiter, erst ab einer bestimmten Skalierung bemerkbar und hat auch nicht eine so starke Auswirkung wie dieser.

"If you’ve ever worked with APIs for some huge products you know that they have rate limiting applied to almost any operation. Examples: [Facebook](https://developers.facebook.com/docs/graph-api/overview/rate-limiting), [Twitter](https://developer.twitter.com/en/docs/basics/rate-limiting), [Google Analytics](https://developers.google.com/analytics/devguides/config/mgmt/v3/limits-quotas)…” (Storozhuk 2018).

Allerdings, ganz im Gegenteil zur Resultierenden Schlussfolgerung, sind sie umso Wichtigere für kleinere Anwendungen und man sollte sich vermehrt überlegen sie Einzubauen.

Abbildung 21: Rate Limiter (Storozhuk 2018)

Sie helfen Anfragespitzen zu verhindern was im Speziellen für kleinere Anwendungen wichtig ist. Anfragespitzen werden entweder in **Warteschlangen** abgelegt und später abgearbeitet oder einfach **abgelehnt,** siehe Abbildung 21 die blauen Flächen werden nach hinten verschoben und später verarbeitet. Nicht nur kann man nach unterschiedlichen Anfragen Typen filtern und diese Separat handhaben, kategorisieren und unterschiedliche Limits für die entsprechenden Gruppen geben. Durch diese Techniken wird eine erhöhte Verfügbarkeit und Verlässlichkeit erreicht, wodurch die Anwendung darauf vorbereitet wird Skalierbar zu sein. Aus diesem Grund wird Rate Limiting in vielen API Gateways oder Proxys schon direkt mitgeliefert was im Teil 4.1 Kommunikations-Leitung beschrieben wurde.

### Fallback

#### Was Passiert bei Fehlern?

Wenn Services ausfallen ist es das einfachste, einfach einen Fehler für den Nutzer auszuwerfen. Es ist allerdings auch die wohl unschönste Antwort welche einem Nutzer geliefert werden kann und da wir unsere Systeme in der Regel auch für unsere Nutzer bauen, sollten sie auch die höchste Priorität haben.

#### Das Fallback Pattern

Das Fallback Pattern ist in diesem Falle mehr eine Ergänzung zu Pattern wie Retry und Circuit breaker welche gewährleisten das bei Fehlern unsere Services sich erholen können bzw. Anfragen bei Fehlern wiederholt werden. Sorgt das Fallback Pattern dafür, dass bei Auftretenden Fehlern auf mögliche Nützliche Informationen zurückgegriffen wird. Fallbacks führen hierbei nur wenig bis gar keine Verarbeitung durch, da wir uns ja bereits im Fehlerzustand befinden. Hinzukommt das die entsprechenden Fallbacks nur eine geringe Wahrscheinlichkeit haben sollten selbst Fehlzuschlagen.

Im Folgenden drei unterschiedliche Möglichkeiten einen Fallback einzubauen.

* Stumm Fehleschlagen: Ist die einfachste und vermutlich am weitest verbreiteten Art eines Fallbacks. Hierbei wird einfach ein null wert zurückgegeben, was dann nützlich sein kann wenn die angefragte Ressource optional ist und in der Antwort einfach weggelassen werden kann.
* Speziell Fehlschlagen: Wird in Fällen benutzt wenn es Fallback Methoden in einer Client Bibliothek gibt. Alternative können wir versuchen Lokal gecachte Daten zurückgreifen umso eine geeignete Antwort zu generieren.
* Schnelles Fehlschlagen: Wird verwendet wenn die Daten benötigt werden und es keine wirkliche alternative Möglichkeit gibt. In diesem Fall stellen wir das wohl des Services über die der Nutzererfahrung und Erlauben dem Service sich zu erholen, sodass der Normale Verlauf wieder hergestellt werden kann.

#### Umsetzung

Die hierbei verwendete Bibliothek ist Resilient4j, mit dem speziellen Module für das Spring Boot 2 Framework, welches die Verwendung der Bibliothek deutlich vereinfacht. Eine Fallbackmethode kann hierbei an jegliche Robustheit Technik abgebracht werden, in diesem Beispiel ist es die Retry Technik. Die Fallbackmethode muss sich hierbei in derselben Klasse befinden, wie der Aufruf siehe Codeteil 7 und dieselbe Struktur besitzen sowie genau einen zusätzlichen Exception Parameter siehe Codeteil 8. Falls mehrere Fallbackmethoden angegeben wurden, wird diejenige aufgerufen welche die Exception aufweist, welche der ausgelösten Exception am ähnlichsten ist.

@Retry(name=**"BACKEND"**,fallbackMethod = **"secondFallback"**)  
@GetMapping()  
**public** ModelAndView getItems()

Codeteil 8: Methode mit Fallback

**public** ModelAndView secondFallback(Exception e) {

cached data, alternative API, static Fallback data …

}

Codeteil 9: Aufzurufende Fallback Methode

### Zusammenfassung

Es wurde nun einige Probleme in verteilten System dargestellt, sowie unterschiedliche Möglichkeiten aufgezeigt, um mit diesen auftretenden Problemen umzugehen und wie diese Lösungen praktisch aussehen können.

Nachdem es nun darum gegangen ist wie Services aufrechterhalten werden können bzw. lauffähig bleiben, geht es nun im Folgenden darum wie Services überhaupt gefunden werden können, wie dies sowohl in der Theorie als auch in der Praxis aussieht und welche unterschiedlichen Möglichkeiten es hierfür gibt.

## Service discovery

### Einleitung

Abbildung 22: Wieso ist Service discovery notwendig (Richardson 2015)

Mit der Microservice Architektur, kommen anstelle von ein paar wenigen schweren übersichtlichen und Möglicher weiße fest zusammenhängenden Services, sehr viele, leichte, kurzlebige und autoskalierte Services zum Einsatz, siehe Abbildung 22 um zu sehen wie dies Aussehen kann. Und jeder dieser neuen Services soll auch die Möglichkeit besitzen miteinander kommunizieren zu können. Wo es bisher noch möglich war Services manuell zuzuweisen, Adressen Festzulegen und diese zu verbreiten, ist nun vollkommen Unvorstellbar und muss automatisiert werden.

### Was ist Service discovery

Um das beschriebene Problem zu lösen wurde die Service discovery Entwickelt. Sie bietet einen oder mehrere Zentrale Orte an, wo die jeweiligen gesuchten Services gefunden werden können. Bevor Services allerdings gefunden werden können, müssen sich diese zuerst einmal anmelden, deshalb wird diese Komponente auch Service Registries genannt. Im selben Schritt wird in diesem Prozess auch eine Lastenverteilung durchgeführt wodurch sie nichtmehr an anderer Stelle zusätzlich benötigt wird. Um diese Technik anzuwenden, gibt es unterschiedliche Herangehensweisen, welche nun angeschaut werden.

### Serverseitige discovery

In dieser Technik wird ein separater Load balancer benutzt, welcher auch als Eintrittspunkt für die alle Clients dient[1]. Der Load balancer fragt bei der Service Registry nach[2] und leitet die entsprechenden Clients an die gesuchten verfügbaren Instanzen weiter[3] (immer ein zusätzlicher Stopp).

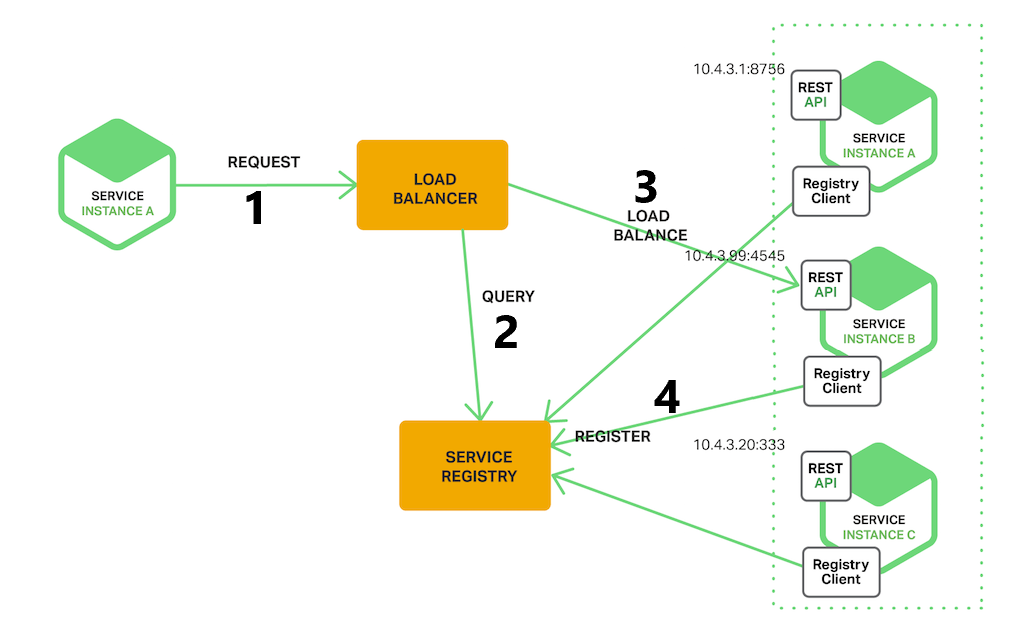


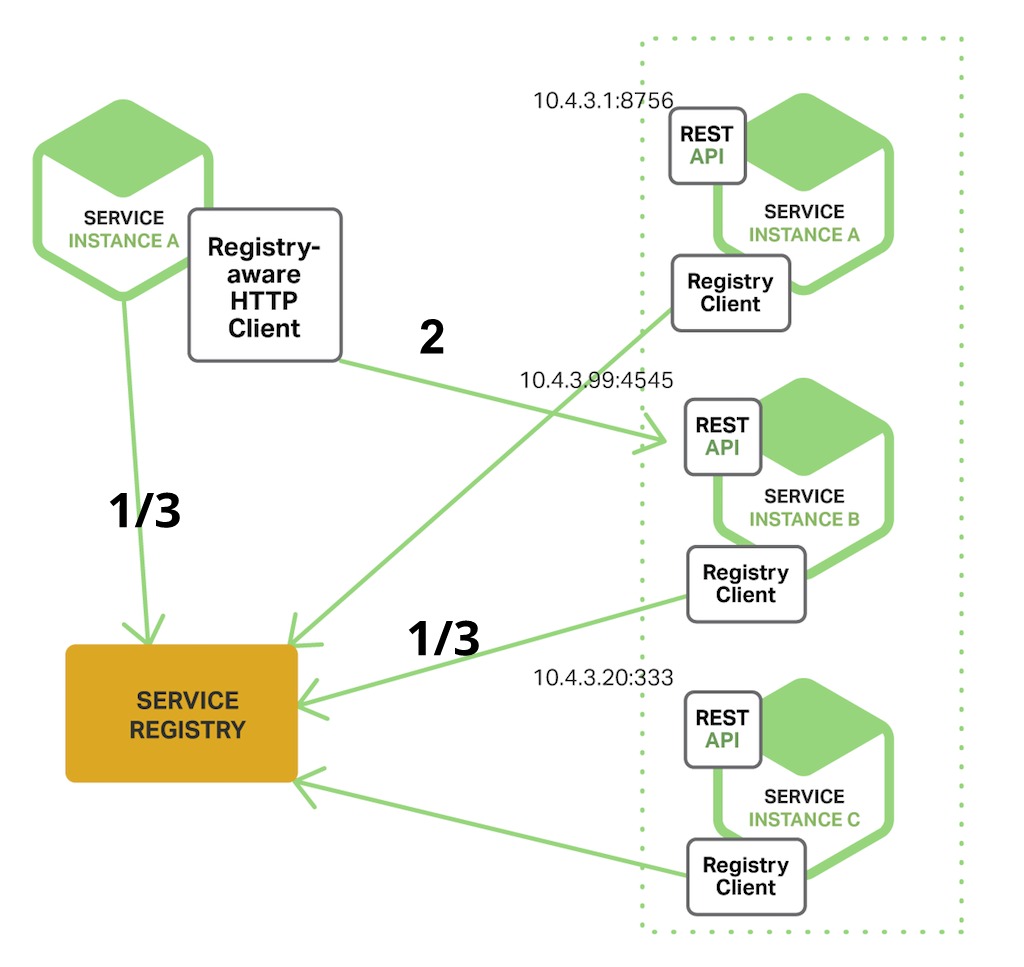
Abbildung 23: Server-Side discovery Pattern (Richardson 2015)

Auf die genauen Registrierungsverfahren [4] wird später eingegangen.

Wenn Serverseitige discovery benutzt wird kommen einige Vor- und Nachteile mit sich. Durch die Abstraktion des discovery Mechanismus, ist es möglich Clients/Services unabhängig von ihrer Sprache zu benutzten und entfernt die Notwendigkeit diese zu Implementieren. Außerdem stellen einige Bereitstellungsumgebungen(Kubernetes) diese bereits frei zur Verfügung. Das Problem dieser Technik ist, dass durch die Einfügung eines zusätzlichen Load balancers, eine weitere Hochverfügbare Komponente eingefügt wird.

### Clientseitige discovery

Bei der Clientseitigen discovery Methode, sind die Clients dafür verantwortlich, die verfügbaren Service Instanzen zu finden und die Last über sie zu verteilen. Dafür fragen die Clients bei der Service Registry nach [1] und erhalten die verfügbaren Instanzen für die jeweilige Aufgabe. Die Clients, benutzen dann einen Load balancing Algorithmus, um einen Service auszuwählen [2].

Abbildung 24: Client-side discovery Pattern (Richardson 2015)

Auf die genauen Registrierungsverfahren wird später eingegangen [3].

Diese Methode bringt ebenso einige Vor- und Nachteile mit sich. Abgesehen von der Service Registry entstehen keine zusätzlichen hochverfügbaren Komponenten. Die Clients können intelligente und Anwendungsspezifische Load balancing Entscheidungen durchführen. Ein auftretendes Problem, betrifft die Client, Registry Verknüpfung. Diese muss im Client für jede unterschiedliche Sprache bzw. jedes Framework Implementiert werden.

### Selbstregistrierungsmethode

Jede Service Instanz ist in dieser Methode, selbst dafür verantwortlich sich an der Registry Anzumelden siehe eins und drei in der Abbildung 24. Außerdem sendet der Client kontinuierliche Herzschläge/Nachrichten um nicht abgemeldet zu werden. Die Nachteile gleichen denn der Clientseitigen Methode, der Registrierungscode muss für jede Sprache oder jedes Framework erneut geschrieben werden wenn diese nicht schon vorhanden sind.

### Drittparteiregistrierungsmethode

Mit dieser Methode, sind die Clients/Services nicht selbst verantwortlich sich an- oder abzumelden. Stattdessen übernimmt diese Aufgabe eine Drittkomponente namens Service Registrator. Der Registrator überwacht Änderungen der Services entweder durch Herzschläge oder durch Überwachung der Events des Services (Abbildung 23: Server-Side discovery Pattern (Richardson 2015)[4]). Wenn der Registrator neue Services erkennt, Registriert er diese an der Registry und meldet diese wieder ab, sollten diese nicht länger verfügbar sein.

Der große Vorteil eine Drittkomponente zu verwenden, besteht darin das die Services nicht mehr eng mit der Registry verbunden sind. Es ist nicht notwendig für die selbst Registrierung eine Implementierung für jeden unterschiedlichen Service zu schreiben. Der Nachteil der dabei entsteht ist der, dass eine weitere Hoch verfügbare Komponente hinzukommt welche gemanagte werden muss.

### Umsetzung

#### Beispiel für Clientseitige discovery mit Selbstregistrierungsmethode

Anhand von Eureka, welcher von Netzflix gebaut wurde, wird im Folgenden eine Clientseitige discovery mit Selbstregistrierung aufgezeigt wobei sowohl Server als auch Client vorgezeigt wird. Das zu Grunde liegende System ist Java mit dem Spring-Boot Framework für welches Eureka speziell gebaut wurde. Wie bei allen anderen Bibliotheken wird auch für Eureka eine Abhängigkeit über z.B. Maven benötigt (Maven Managet unteranderem die Abhängigkeiten in einem Java Projekt).

Der Discovery Server:

Der Discovery Server ist wie zu sehen ist dank Eureka und Spring Boot äußerst schlank und übernimmt uns fast alle Arbeit. Alleine die Notation @EnableEurekaServer wird benötigt um den Server zu starten. Jegliche Einstellungen für den Laufenden Server werden über die application.properties Datei Eingestellt.

@EnableEurekaServer  
@SpringBootApplication  
**public class** DiscoveryServerApplication {

Codeteil 10: Service discovery Server

Der discovery Client:

Der Service discovery Client benötigt hierbei schon etwas mehr Code von der Entwickler Seite siehe Codeteil 10. Zum Anfang wird wie beim Server auch die @EnableDiscoveryClient Notation benötigt. Als Nächstes sehen wir einige Variablen, welche zeigen wie die bisherige hartgecoded Verbindung aussah. Durch die @Autowirded markierte DiscoveryClient Variable wird diese automatisch initialisiert und kann im Anschluss verwendet werden. Die folgende Methode übernimmt den Namen einer Applikation, welche gesucht wird und fragt anschließend über den Initialisierten DiscoveryClient bei der Registry mit dem entsprechenden Service Namen nach. Im Anschluss würde nun das Clientseitige Loadbalancing stattfinden, wobei in diesem Falle eine beliebige Instanz verwendet wird, um im Anschluss dessen Host aufzulösen und zurückzugeben. Weitere spezifische Einstellungen für den Client, werden wie auch beim Server über die application.properties Festgelegt.

Die Registrierung am Server passiert vollkommen automatisch im Hintergrund.

@SpringBootApplication  
@EnableDiscoveryClient  
**public class** TodoUiApplication {  
 String **backendHost**=**"localhost"**;  
 String **backendPort**=**"8080"**;

String **host**= **""**;  
 @Autowired  
 **private** DiscoveryClient **discoveryClient**;  
 @RequestMapping(**"/service-instances/{applicationName}"**)  
 **public** String serviceInstancesByApplicationName(@PathVariable String applica tionName){  
 List<ServiceInstance> nameSpecificInstances = **this**.**discoveryClient**.getInstances(applicationName);  
 Optional<ServiceInstance> optionalServiceInstance = nameSpecificInstances.stream().findAny();  
 **if**(optionalServiceInstance.isPresent())  
 {  
 **host** = instances.stream().findAny().get().getUri().toString();  
 }  
 System.***out***.println(**host**);  
 **return host**;  
 }

Codeteil 11: Service discovery Client

#### Drittparteiregistrierungsmethode

Beispiel anhand von Istio(Kubernetes):

Istio selbst besitzt keinen eigenen Service discovery Mechanismus dafür allerdings eine Service Registry. Für den discovery Mechanismus greift Istio auf das unterliegende Orchestrierungssystem zurück, welches sehr oft Kubernetes ist und aus diesem Grund auch hier verwendet wird.

# Zusammenfassung

Die Ergebnisse auf einen Blick. Was sind die Ergebnisse aus den einzelnen Bereichen.

Was wurde Erreicht. Was wurde nicht Erreicht.

Reflexion und Bewertung der Aufgabenstellung

Was würde beim nächsten Mal besser gemacht werden?

Einleitung begonnene rote Faden in einem Vergleich, ob und wie die anfänglichen Aufgabenstellungen von der Arbeit nun tatsächlich erfüllt wurden.

Wichtigsten Aussagen der Arbeit noch einmal aufgezählt, miteinander in Beziehung gebracht und bewertet.

welche Punkte der Aufgabenstellung nicht oder nur unzureichend behandelt

werden konnten, solange man schlüssig nachweisen kann, dass es sich um sehr komplexe Aufgabenstellungen handelt, die den Umfang der Arbeit gesprengt hätten.

Sinn und Zweck der Arbeit.

Erreicht:

Theoretische Grundlagen und Einleitung Erfüllt.

Überblick über unterschiedliche Aspekte der Kommunikation

Gegenüberstellung von Techniken von Service Meshes zu Ohne Service Mesh.

Nicht Erreicht:

Reale Umsetzung eines Komplexen Service Meshes.

Resilience Mechanismen in Komplexen Service Mesh.

Zusätzlicher Nachforschungen und Möglichkeiten wurden vernachlässigt.

Zusätzliche Service Meshes und deren Anwendung,

# Ausblick

Die Welt soweit man das momentan sagen kann wird auch in Zukunft immer Globaler.

Verteilte Systeme und Microservices werden immer mehr angenommen und verwendet.

Die Notwendigen Werkzeuge und Verständnisse der entsprechenden Techniken werden dadurch immer mehr Wichtiger und notwendig.

Ein Überblick über die Möglichkeiten in einzelnen Bereichen

# Glossar

# Ehrenwörtliche Erklärung

|  |  |  |  |
| --- | --- | --- | --- |
| Name: |  | Vorname: |  |
| Matrikel-Nr.: |  | Studiengang: |  |

Hiermit versichere ich, <Vorname, Name>, dass ich die vorliegende <Bachelor- oder Masterarbeit> mit dem Titel <Titel der Abschlussarbeit> selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken ent­nommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ort, Datum Unterschrift

# Stichwortverzeichnis

Abbildung 25

Literaturverzeichnis

Dashora, Saurabh: *The Rise of Spring Framework | PROGRESSIVE CODER.* URL http://progressivecoder.com/the-rise-of-spring-framework/. – Aktualisierungsdatum: 16.01.2019 – Überprüfungsdatum 13.12.2019

Dimensional Research: *global-microservices-trends-2018.* A SURVEY OF DEVELOPMENT PROFESSIONALS (2018). URL https://go.lightstep.com/rs/260-KGM-472/images/global-microservices-trends-2018.pdf – Überprüfungsdatum 27.01.2020

Eigenkreation

Fowler, Martin: *CircuitBreaker.* URL https://martinfowler.com/bliki/CircuitBreaker.html. – Aktualisierungsdatum: 04.10.2019 – Überprüfungsdatum 14.10.2019

Golden, Bernard CEO Navica: *3 reasons you should always run microservices apps in containers | Learn.* URL https://learn.techbeacon.com/units/3-reasons-you-should-always-run-microservices-apps-containers. – Aktualisierungsdatum: 02.05.2017 – Überprüfungsdatum 22.11.2019

IBM: *Limiting the number of concurrent requests to microservices.* URL https://openliberty.io/guides/bulkhead.html#background-concepts. – Aktualisierungsdatum: 28.11.2019 – Überprüfungsdatum 02.12.2019

Kubernetes: *Kubernetes Deployments.* URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment – Überprüfungsdatum 25.11.2019

Kyma: *Tracing - Docs | Kyma - An easy way to extend enterprise applications on Kubernetes.* URL https://kyma-project.io/docs/components/tracing#details-benefits-of-distributed-tracing – Überprüfungsdatum 07.11.2019

Layer5: *Service Mesh Landscape.* URL https://layer5.io/landscape/. – Aktualisierungsdatum: 28.10.2019 – Überprüfungsdatum 30.10.2019

McAllister, Dave: *GOTO 2019 • Observability, Distributed Tracing & the Complex World • Dave McAllister - YouTube.* URL https://www.youtube.com/watch?v=2nTJSsBngao – Überprüfungsdatum 08.11.2019

Microsoft, dragon119: *Wiederholungsmuster - Cloud Design Patterns.* URL https://docs.microsoft.com/de-de/azure/architecture/patterns/retry – Überprüfungsdatum 03.12.2019

Paraschiv, Eugen: *The State of Java in 2018.* URL https://www.baeldung.com/java-in-2018#spring-adoption. – Aktualisierungsdatum: 06.11.2018 – Überprüfungsdatum 13.12.2019

Posta, Christian: *API Gateways Are Going Through an Identity Crisis.* URL https://blog.christianposta.com/microservices/api-gateways-are-going-through-an-identity-crisis/. – Aktualisierungsdatum: 16.10.2019 – Überprüfungsdatum 03.12.2019

Prinz, Hanna: *Service Mesh – für Microservices unverzichtbar?* URL https://www.informatik-aktuell.de/entwicklung/methoden/service-mesh-fuer-microservices-unverzichtbar.html. – Aktualisierungsdatum: 15.10.2019 – Überprüfungsdatum 05.12.2019

Richardson, Chris: *Microservices Pattern: API gateway pattern.* URL https://microservices.io/patterns/apigateway.html. – Aktualisierungsdatum: 20.11.2019 – Überprüfungsdatum 02.12.2019

Richardson, Chris: *Service Discovery in a Microservices Architecture - NGINX.* URL https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ – Überprüfungsdatum 31.10.2019

Rotem-Gal-Oz, Arnon: *Fallacies of distributed computing Explained.* URL http://www.rgoarchitects.com/Files/fallacies.pdf – Überprüfungsdatum 09.10.2019

Sebastian Eschweiler: *Docker – Beginner’s Guide – Part 1: Images & Containers.* URL https://codingthesmartway.com/docker-beginners-guide-part-1-images-containers/. – Aktualisierungsdatum: 24.02.2019 – Überprüfungsdatum 26.01.2020

StackRox: *The State of Container and Kubernetes Security Spring 2019 : With observations and analysis from AimPoint Group.* 2019

Storozhuk, Bogdan: *Rate Limiter Internals in Resilience4j.* URL https://medium.com/@storozhuk.b.m/rate-limiter-internals-in-resilience4j-48776e433b90#7585 – Überprüfungsdatum 21.10.2019

Tanenbaum, Andrew S.; Maarten Van Steen (Mitarb.): *Distributed Systems Principles and Paradigms.* Zweite Auflage : Pearson Education, 2006

Ushio, Tsuyoshi: *Kubernetes in three diagrams.* URL https://medium.com/@tsuyoshiushio/kubernetes-in-three-diagrams-6aba8432541c. – Aktualisierungsdatum: 02.05.2018 – Überprüfungsdatum 22.11.2019

Vogels, Werner: *A Conversation with Werner Vogels - ACM Queue.* URL https://dl.acm.org/doi/pdf/10.1145/1142055.1142065. – Aktualisierungsdatum: 30.06.2006 – Überprüfungsdatum 21.11.2019