

**Bachelorarbeit**  
**In der Angewandten Informatik**

**Technologische Ansätze zur Umsetzung  
einer Microservice-Architektur**

Prototypische Implementierung einer Anwendung  
zur Verwaltung der IT-Kontaktmesse an der  
Fachhochschule Erfurt

**Benjamin Swarovsky**

Abgabedatum: 01.11.2021

**Prof. Dr. Steffen Avemarg**

**Dipl.-Inf. Steffen Späthe**

## Kurzfassung

Ein weit verbreitetes Software-Architekturmuster stellt heutzutage die Microservice-Architektur dar. Folgende wissenschaftliche Arbeit beschreibt Technologien wie zum Beispiel Frameworks und Bibliotheken, welche für die erfolgreiche Verwirklichung einer solchen Architektur eingesetzt werden können. Unter anderem werden Grundlagen wie Algorithmen, Protokolle und Entwurfsmuster genannt, auf denen diese Technologien basieren. Beispielhaft wird der Einsatz der Technologien anhand eines Softwareprototyps demonstriert. Dieser dient zur Umsetzung einer Anwendung, zur Verwaltung der IT-Kontaktmesse an der Fachhochschule Erfurt. Der Prototyp wird anhand einer Microservice-Architektur realisiert. Es wird dargestellt, wie diese Architektur auf Grundlagen der Anforderungen, welche an das Verwaltungssystem gestellt werden, angefertigt wird. Am Ende erfolgt eine Auswertung, inwieweit die vorgestellten Technologien implementiert werden konnten. Zusätzlich wird die Bedeutsamkeit dieser Technologien, für die erfolgreiche Verwirklichung einer vollständigen Anwendung, welche über den Funktionsumfang des Prototyps hinausgehen würde, beurteilt.

## Abstract

The microservice architecture is a popular software architecture pattern nowadays. The following scientific work describes technologies such as frameworks and libraries, which can be used for the successful implementation of such an architecture. Among other things, fundamentals such as algorithms, protocols and design patterns on which these technologies are based are named. The use of the technologies is demonstrated using a software prototype as an example. This is used to implement an application, to manage the IT contact fair at the Erfurt University of Applied Sciences. The prototype is implemented using a microservice architecture. It is shown how this architecture is made based on the requirements that are placed on the management system. At the end there is an evaluation to what extent the presented technologies could be implemented. In addition, the importance of these technologies for the successful implementation of a complete application that would go beyond the functional scope of the prototype is assessed.

# Inhaltsverzeichnis

Kurzfassung.....	II
Abstract .....	III
I. Abkürzungsverzeichnis .....	VI
II. Abbildungsverzeichnis.....	VII
1 Einleitung.....	1
1.1 Problemstellung.....	1
1.2 Ziel.....	2
2 Grundlagen.....	3
2.1 Microservices.....	3
2.1.1 Vorteile .....	3
2.1.2 Nachteile.....	3
2.2 Frameworks.....	4
2.3 Kommunikation zwischen Microservices.....	5
2.3.1 Synchrone Kommunikation .....	5
2.3.2 Asynchrone Kommunikation.....	6
2.4 Load Balancer .....	7
2.5 Service Discovery.....	8
2.6 API Gateway.....	10
2.7 Autorisierung und Authentifizierung .....	12
2.8 Circuit Breaker Pattern .....	14
2.9 Distributed Tracing .....	15
2.10 User Interface .....	16
2.11 Container .....	17
3 Anforderungsanalyse .....	19
3.1 Aufgabenstellung.....	19
3.2 Qualitätsziele .....	20
3.3 Stakeholder .....	21
4 Architekturentwurf.....	23
4.1 Lösungsstrategie .....	23
4.2 Systemkontext (Ebene 0) .....	25
4.3 Domain Driven Design .....	26
4.4 Bausteinsicht (Ebene 1) .....	27
4.5 Bausteinsicht (Ebene2) .....	29
4.6 Domainmodell.....	31
4.6.1 Messestammdatenservice.....	31

4.6.2	Firmenservice .....	31
4.6.3	Newsletterservice .....	32
4.6.4	Vortragsservice .....	32
5	Implementierung .....	33
5.1	Spring Framework .....	33
5.2	Build-Automatisierung mit Maven .....	34
5.3	Frontend mit Thymeleaf .....	35
5.4	Spring Cloud API Gateway .....	39
5.5	Eureka Discovery Service .....	40
5.6	Keycloak und Spring Security .....	42
5.7	Synchrone Kommunikation mit Feign Client .....	47
5.8	Resilience4J Circuit Breaker .....	49
5.9	Docker Container .....	52
5.10	Distributed Tracing mit Jaeger .....	54
6	Auswertung .....	58
7	Zusammenfassung und Ausblicke .....	61
III.	Literaturverzeichnis .....	IX
	Selbstständigkeitserklärung .....	XIV

## I. Abkürzungsverzeichnis

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CSS Cascading Style Sheets

FH Fachhochschule

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

REST Representational State Transfer

TCP Transmission Control Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

## II. Abbildungsverzeichnis

Abbildung 1 ereignisgesteuerte Kommunikation .....	7
Abbildung 2 Loadbalancing .....	8
Abbildung 3 Clientseitige Erkennung .....	9
Abbildung 4 Serverseitige Erkennung.....	10
Abbildung 5 Kommunikation ohne Gateway .....	11
Abbildung 6 Kommunikation mit Gateway .....	12
Abbildung 7 Circuit Breaker Funktionsweise .....	15
Abbildung 8 Frontend Monolith .....	16
Abbildung 9 Modulares Frontend .....	17
Abbildung 10 Systemkontext .....	26
Abbildung 11 Bounded Contexts .....	27
Abbildung 12 Bausteinsicht Ebene 1 .....	29
Abbildung 13 Bausteinsicht Ebene 2 .....	30
Abbildung 14 Domainmodel Messestammdatenservice .....	31
Abbildung 15 Domainmodel Firmenservice .....	31
Abbildung 16 Domainmodel Newsletterservice .....	32
Abbildung 17 Domainmodel Vortragservice .....	32
Abbildung 18 spring initializr .....	34
Abbildung 19 Discovery Service pom.xml .....	35
Abbildung 20 Besucherservice start.html .....	36
Abbildung 21 Besucherservice guestController.java .....	36
Abbildung 22 Thymeleaf Anweisungen, Besucherservice firmen.html .....	37
Abbildung 23 Ausgabe im Browser, Besucherservice firmen.html .....	38
Abbildung 24 API Gateway application.properties.....	40
Abbildung 25 DiscoveryService (Eureka Server) application.properties .....	40
Abbildung 26 Microservices (Eureka Clients) application.propperties .....	41
Abbildung 27 Eureka Dashboard .....	41
Abbildung 28 Keycloak Konfiguration Anmeldeformular .....	43
Abbildung 29 Keycloak Administrationsoberfläche .....	44
Abbildung 30 Keycloak-Konfiguration, ApiGateway application.properties .....	45
Abbildung 31 URL des Autorisierungsservers in den einzelnen Microservices, application.properties .....	45
Abbildung 32 Spring Security Filterkette, ApiGateway SecurityConfig.java .....	46
Abbildung 33 Loginmaske IT-KoM Verwaltung .....	47
Abbildung 34 Besucherservice FirmenServiceClient .....	48
Abbildung 35 Implementierung FirmenServiceClient und Funktionsaufruf, Besucherservice guestController.java .....	48
Abbildung 36 allCompanies Aufruf, Firmenservice Controller.java .....	49
Abbildung 37 Circuit Breaker Konfiguration, Besucherservice guestController.java .....	50
Abbildung 38 Supplier mit Circuit Breaker dekoriert .....	50
Abbildung 39 Test des Circuit Breakers über eine for Schleife .....	51
Abbildung 40 Firmenservice Controller allCompanies (mit Thread.sleep) .....	52
Abbildung 41 Test des Circuit Breakers Konsolenausgabe .....	52
Abbildung 42 Besucherservice dockerfile .....	53
Abbildung 43 docker Desktop Benutzeroberfläche .....	54
Abbildung 44 JaegerConfig.java .....	55

Abbildung 45 Jaeger Benutzeroberfläche .....	55
Abbildung 46 Trace eines Funktionsaufrufes der Funktion allCompanies vom Besucherservice .....	56
Abbildung 47 Firmenservice Adressangabe .....	57



# 1 Einleitung

## 1.1 Problemstellung

Über Jahre hinweg wurden Softwaresysteme als Monolithen entwickelt. Aufgrund ihrer eng gekoppelten Komponenten bilden solche Systeme eine untrennbare Einheit.<sup>1</sup> Wenn diese Anwendungen mit der Zeit immer größer werden, häufen sich für die Entwickler organisatorische Probleme.<sup>2</sup> Extrem große Systeme sind nicht leicht zu verstehen und zu verwalten. Durch enge Kopplung der Komponenten ist es schwierig Änderungen in der Anwendung einzuspielen. Codeänderungen nehmen Einfluss auf das gesamte System und müssen gründlich koordiniert werden. Diese Problemstellung kann bei einem großen System mit engen verzahnten Komponenten dazu führen, dass bei der Einführung einer neuen Technologie die gesamte Anwendung komplett neu geschrieben werden muss. Beansprucht eine einzelne Funktion zu viel Rechenleistung, dann kann die Performance der gesamten Anwendung darunter leiden. Der Ausfall einer einzelnen Komponente kann zum Ausfall der gesamten Anwendung führen.

Ein weiteres Problem lässt sich gegenüber der Skalierbarkeit erkennen. Skalierbarkeit bedeutet unter anderem die Möglichkeit zu schaffen, bei steigender Systemlast neue Instanzen der Anwendung (zum Beispiel auf verschiedenen Servern) zu erstellen. Dadurch kann die Last verteilt werden. Bei einem Monolithen kann nur das komplette System skaliert werden. Die Skalierung von einzelnen Komponenten ist nicht möglich, weil einzelne Teilbereiche nicht unabhängig voneinander interagieren können. Stehen nur wenige Komponenten einer Anwendung unter hoher Last, dann ist es ineffizient Kopien der kompletten Anwendung zu erstellen. In diesem Fall müssen für jede Instanz die entsprechend hohen Ressourcen bereitgestellt werden.<sup>3</sup>

Gegenüber diesen Nachteilen schafft eine Microservice-Architektur Abhilfe. Seit einigen Jahren erlebt dieses Architekturmuster einen regelrechten Hype. Beispiele für die erfolgreiche Umsetzung einer Microservice-Architektur liefern große Firmen wie zum Beispiel Amazon, Netflix und Zalando. Laut Eberhard Wolf bringt der Hype einen großen Nachteil mit sich. Die Architektur wird oft ausgewählt, weil sie gerade in Mode ist. Microservices sind eines von vielen Architekturmustern, welches je nach Anwendungsfall mehr oder weniger für ein System geeignet ist. Über die Umsetzung werden sich dann in vielen Fällen zu wenig Gedanken gemacht. Dabei kann die Verwirklichung einer solchen Architektur als sehr anspruchsvoll angesehen werden. Es gilt Herausforderungen zu überwinden wie zum Beispiel die Modellierung der Datenmodelle mit jeweils einer eigenen Datenbank pro Microservice, Kommunikation unter verteilten Systemen, sowie Fehlerbehandlung und Monitoring über mehrere Services hinweg.<sup>4</sup>

---

<sup>1</sup> (Fink, 2012)

<sup>2</sup> (Alzve, 2021)

<sup>3</sup> (Gnatyk, 2018)

<sup>4</sup> (Wolff, 2017)

### 1.2 Ziel

Diese wissenschaftliche Arbeit behandelt Problemstellungen, die bei der technischen Umsetzung einer Microservice-Architektur auftreten. Es soll dargelegt werden, welchen Mehraufwand Microservices gegenüber Monolithen bei der Entwicklung erfordern. Es werden populäre Frameworks, Bibliotheken und Entwurfsmuster vorgestellt, welche in einer Microservice-Architektur häufig zum Einsatz kommen. Dabei wird erläutert welchen Nutzen diese bringen und wie sie implementiert werden.

Mit dem Thema wird sich anhand eines praktischen Beispiels auseinandergesetzt. Dieses wird umgesetzt anhand eines Entwurfes für ein System zur Verwaltung der IT-Kontaktmesse an der Fachhochschule Erfurt. Die Messe findet jährlich auf dem Gelände der Hochschule statt. Unternehmen aus der Region stellen sich gegenüber den Studierenden an Messeständen vor und präsentieren sich anhand eigener Vorträge. Das System soll den Firmen unter anderem die Möglichkeit bieten sich für die Messe zu registrieren, sich zu informieren und einen eigenen Messeauftritt zu organisieren. Im weiteren Verlauf dieser Arbeit wird das System anhand einer Microservice-Architektur entworfen. Es wird die Architekturentscheidung begründet und es wird auf die Nachteile eingegangen, welche bei der Entwicklung der Anwendung aufgrund dieser Entscheidung entstehen. Im Anschluss wird ein Prototyp implementiert. Dieser soll zeigen, mit welchen Mitteln die genannten Problemstellungen in der Praxis gelöst werden können. Am Ende wird der Aufwand der Umsetzung ausgewertet. Es werden Probleme genannt, die bei der Implementierung auftraten. Zusätzlich erfolgt eine Einschätzung, mit welcher Relevanz die einzelnen verwendeten Technologien zu einem erfolgreichen Betrieb der Anwendung beitragen.

## 2 Grundlagen

### 2.1 Microservices

Microservices stellen einen Software-Architekturansatz dar. Dieser entstand in den frühen 1980er Jahren mit den von der Firma Sun Microsystems entwickelten Remote Procedure Calls, welche als eine der ersten Technologien zur Umsetzung von verteilten Systemen entwickelt wurden. Die ersten Einsätze von Microservices wurden von James Lewis und Martin Fowler im Jahr 2014 beschrieben.<sup>5</sup>

Im Gegensatz zum Architekturansatz des Deployment-Monolithen, bei dem das System nur als Ganzes deployt werden kann, gelten Microservices laut Eberhard Wolff als unabhängig funktionsfähige Module. Die Größe der einzelnen Services hängt vom jeweiligen Anwendungsfall ab.<sup>6</sup> Ein Service sollte klein genug gehalten werden, um von einem einzelnen Entwicklerteam entwickelt zu werden. Bei zu kleinen Services steigt die Anzahl der Services im gesamten System. Verteilte Aufrufe anderer Systeme über das Netzwerk sind zeitaufwändiger als Aufrufe im selben Prozess. Um einer Erhöhung der Verzögerungszeit entgegenzuwirken, sollten die Services nach Möglichkeit nicht zu klein gehalten werden.<sup>7</sup>

#### 2.1.1 Vorteile

Eine Microservice-Architektur bildet ein System bestehend aus kleinen einzelnen Programmen. Diese werden möglichst voneinander unabhängig gehalten. Dadurch ist das gesamte System weniger anfällig gegen ungewolltes Einbauen von Abhängigkeiten zwischen einzelnen Komponenten. Aufgrund dessen bleibt das System übersichtlich und wartbar. Durch die Aufteilung von Fachlichkeit bei einer Microservices-Architektur, ist die Logik für Entwickler einfacher zu verstehen. Entwickler müssen nicht die Funktionalitäten der gesamten Anwendung verstehen, sondern nur die, die für sie zugewiesenen Microservices. Die einzelnen Microservices sind unabhängig voneinander skalierbar. Dadurch werden Ressourcen gespart. Bei der hohen Auslastung eines Teilsystems werden zum Beispiel nur weitere Instanzen für die betroffenen Services erstellt. Die Duplizierung des gesamten Systems ist nicht notwendig. Das Gesamtsystem ist robust, weil Ausfälle einzelner Services nicht das gesamte System lahmlegen.<sup>8</sup>

#### 2.1.2 Nachteile

Microservices sind verteilte Systeme. Diese Systeme haben den Nachteil, dass jedes einzelne Teilsystem einen Knotenpunkt in einem Netzwerk darstellt. Weil ein solches Netzwerk entsprechend viele Knoten besitzt, hat es eine relativ hohe Latenzzeit. Darüber hinaus benötigt jeder einzelne Microservice einen eigenen Deploymentprozess. Durch die Verteilung der einzelnen Systeme gestaltet sich auch das Testen schwieriger. Die Services müssen einzeln und zusätzlich als Gesamtsystem getestet werden. Weiterhin besteht in einer Anwendung, welche aus einer Vielzahl einzelner Services besteht, eine relativ hohe Wahrscheinlichkeit, dass einzelne Services ausfallen. Microservices sollten daher mit dem

---

<sup>5</sup> (Mohapatra, et al., 2019)

<sup>6</sup> (Wolff, 2018)

<sup>7</sup> (Wolff, 2018)

<sup>8</sup> (Wolff, 2018)

Ausfallen anderer Services umgehen können, ohne dabei selbst auszufallen. Logging und Monitoring lässt sich nur mit relativ hohem Aufwand umsetzen, weil mehrere Systeme daran involviert sind. Darüber hinaus kann es nicht ohne weiteres von zentraler Stelle aus durchgeführt werden. Weil die Daten im System über verschiedene Services verteilt sind, kann sich die Aufrechterhaltung der Konsistenz als schwierig erweisen.<sup>9, 10</sup>

## 2.2 Frameworks

Laut Stephan Augsten schafft ein Framework einen Ordnungsrahmen durch Basisbausteine für die Entwickler. Diese Basisbausteine sind unterteilt in abstrakte und konkrete Klassen. Diese werden dem Entwickler direkt zur Verfügung gestellt und unterstützen mit einer Vielzahl von Entwurfsmustern. Ein Framework besteht aus Bibliotheken, Laufzeitumgebung und weiteren Komponenten. Frameworks bilden ein Programmiergerüst, mit dem Entwicklungszeit eingespart und damit Entwicklungskosten reduziert werden können.<sup>11</sup>

### Microservice Frameworks

Microservice Frameworks erleichtern die Implementierung von Features, welche ein Microservice anbieten sollte. Beispiele dafür sind Sicherheit, Tracing, Service Discovery und Konfigurierbarkeit.

Bei der Entscheidung über den Einsatz eines Microservice Frameworks sollten Vor- und Nachteile abgewogen werden. Folgende Vor- und Nachteile sollten beachtet werden.

Vorteile:

- Weniger Code pro Service
- Ein Microservice ist schneller für die Cloud bereit
- Kürzere Entwicklungszeit
- Infrastrukturcode muss nicht selbst entwickelt werden

Nachteile:

- Entwickler benötigen Einarbeitungszeit zum Verständnis der Framework-Konzepte
- Entwickler geben Kontrolle ab und wissen unter Umständen nicht welche Funktionen das Framework im Hintergrund ausführt
- Der Einsatz von einer Vielzahl von Framework Bibliotheken kann zu Versionsproblemen mit umständlicher Fehlersuche führen

<sup>12</sup>

Populäre Microservice Frameworks sind Springboot mit Spring Cloud, Eclipse Vert.X, Oracle Helidon, GoMicro, Molekular und Quarkus.<sup>13</sup>

---

<sup>9</sup> (Wolff, 2019)

<sup>10</sup> (Isheim, 2018)

<sup>11</sup> (Augsten, 2020)

<sup>12</sup> (Bayer, 2019)

<sup>13</sup> (Kurmi, 2020)

## 2.3 Kommunikation zwischen Microservices

Die Kommunikation zwischen den Microservices kann synchron oder asynchron erfolgen. Die Entscheidung hängt vom jeweiligen Anwendungsfall ab. In einer Microservice-Architektur können beide Kommunikationsmöglichkeiten zum Einsatz kommen. <sup>14</sup>

### 2.3.1 Synchrone Kommunikation

Unter Verwendung von Synchroner Kommunikation wird eine Antwort gesendet und anschließend auf eine Antwort gewartet. Es handelt sich dabei um eine relativ simple Herangehensweise. Die Ausführung erfolgt in der Regel per REST-Schnittstelle über HTTP. <sup>15</sup>

#### **Hypertext Transfer Protocol (HTTP)**

HTTP ist ein Protokoll welches die Kommunikation in einem IP-Netzwerk (zum Beispiel zwischen Webserver und Webbrowser) realisiert. Im Falle einer Webanwendung fungiert der Webserver als HTTP-Server und der Client als Browser. Der Client sendet einen Request an den Port des Servers (in der Regel Port 80) und erhält von diesem eine Response-Nachricht. Die Adressierung der Ressourcen erfolgt per Uniform Resource Identifier (URI)

. Folgende Aktionen können per HTTP auf Ressourcen umgesetzt werden

- GET (Ruft Ressourcen auf)
- POST (Erstellt eine neue Instanz einer Ressource)
- PUT (ändert eine Ressource)
- DELETE (löscht die Instanz einer Ressource)

<sup>16</sup>, <sup>17</sup>, <sup>18</sup>

#### **Uniform Resource Identifier**

Mit einem URI lassen sich abstrakte oder physische Ressourcen wie zum Beispiel Webseiten oder Sender/Empfänger von E-Mails ansprechen. Der Anwendung wird dadurch eine eindeutige Identifikation für die Abfrage von Ressourcen ermöglicht. Die Syntax einer URI besteht laut IONOS höchstens aus folgenden Komponenten.

- Scheme (gibt Auskunft über das verwendete Protokoll wie zum Beispiel HTTP)
- Authority (kennzeichnet die Domäne)
- Path (zeigt den genauen Pfad zur Ressource)
- Query (bietet die Möglichkeit für Abfragen)
- Fragment (kennzeichnet einen Teilaspekt einer Ressource)

*scheme://authority path ? query # fragment*

---

<sup>14</sup> (Schwab, 2019)

<sup>15</sup> (Schwab, 2019)

<sup>16</sup> (mozilla)

<sup>17</sup> (w3schools)

<sup>18</sup> (Luber, 2018)

Ein Beispiel URI wäre `ldap://[2001:db8::7]/c=GB?objectClass?one`. Eine Teilmenge aller URI's bilden Uniform Resource Locator (URL). Diese dienen zum Lokalisieren von Ressourcen und ermöglichen es mit ihnen zu interagieren. Dazu wird bei einer URL immer ein Scheme angegeben wie zum Beispiel HTTP oder ftp.<sup>19, 20</sup>

### Representational State Transfer (REST)

REST bildet eine Softwarearchitektur, welche den Datenaustausch in einem Client-Server Softwaresystem ermöglicht. Eine Schnittstelle, welche unter dem Standard von REST, die Kommunikation zwischen Server und Client umsetzt, wird als REST Application Programming Interface (REST API) bezeichnet. Für REST gibt es kein festgelegtes Übertragungsprotokoll. In der Praxis kommt in der Regel HTTP zum Einsatz. Es können die Grundaktionen Auslesen (GET), Erstellen (POST), Ändern (PUT), und Löschen (DELETE) ausgeführt werden. Beschreibenden Parameter können unter der Verwendung dieser Aktionen zwischen Client und Server ausgetauscht werden.

Bei der Verwendung von REST besitzen Clientanfragen alle Adressinformationen, die für eine Kommunikation benötigt werden. Der Aufbau einer Sitzung ist bei der Kommunikation nicht notwendig. Dieses Verhalten wird als zustandslos bezeichnet. Es trägt zu einer Verbesserung von Zuverlässigkeit und Skalierbarkeit bei. Es liefert jedoch den Nachteil einer Verschlechterung der Netzwerkperformance. Ein weiteres Feature von REST ist das Caching. Es ermöglicht das Speichern von Antworten, welche bei gleichartigen Anfragen erneut verwendet werden können, wodurch die Effizienz erhöht werden kann.<sup>21, 22</sup>

### 2.3.2 Asynchrone Kommunikation

Asynchrone Kommunikation wird bei einer Microservice-Architektur eingesetzt, wenn ein Aufruf die Änderungen von Datenbankeinträgen über mehrere Microservices hinweg verursacht. Ein solcher Aufruf wird über eine ereignisgesteuerte Kommunikation realisiert. Microservices können dabei einen Ereignisbus abonnieren, um Benachrichtigungen zu empfangen. Die Benachrichtigungen werden versendet, wenn ein Microservice Ereignisse auf einem Ereignisbus veröffentlicht. Ein Protokoll, welches eine zuverlässige Kommunikation gewährleistet ist das Advanced Message Queuing Protocol (AMQP). Dieses arbeitet auf der Anwendungsschicht und orientiert sich an dem Transmission Control Protokoll (TCP).<sup>23</sup>

AMQP realisiert eine schnelle und robuste Datenübertragung, weil es keinen Leerlauf produziert. Dieser Vorteil wird durch den Einsatz einer Nachrichtenwarteschlange erzielt. Sender und Empfänger agieren asynchron. Der Sender muss nicht zwingend auf die Bestätigung der Nachricht des Empfängers warten, ohne weiterzuarbeiten. Hat der Empfänger freie Kapazitäten, wird die Nachricht aus der Warteschlange geholt, verarbeitet und bestätigt. Die Kompatibilität zwischen verschiedenen Systemen wird gewährleistet, weil AMQP als ein binäres Nachrichtensystem mit strikten Verhalten für die Nachrichten gestreamt wird.

---

<sup>19</sup> (Ionos, 2019)

<sup>20</sup> (Bauer, 2020)

<sup>21</sup> (Srocke, et al., 2017)

<sup>22</sup> (RedHat)

<sup>23</sup> (Nish, 2021)

Für den Nachrichtenaustausch können folgende verschiedene Betriebsmodi gewählt werden:

- Exactly-once (Einmalige garantierte Auslieferung)
- At-least-once (Dopplungen beim Nachrichtenaustausch möglich, garantierte Auslieferung)
- At-most-once (Die Nachricht wird einmalig gesendet und kann verloren gehen)

<sup>24, 25</sup>

Abbildung 1 zeigt eine ereignisgesteuerte Kommunikation zwischen 3 Microservices über einen Ereignisbus

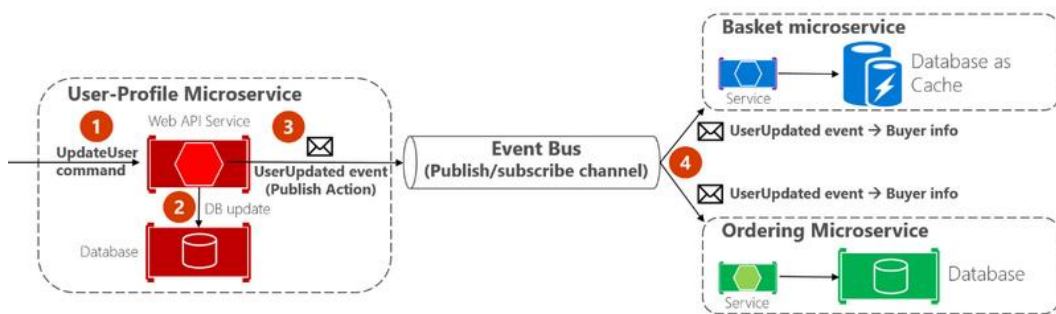


Abbildung 1 ereignisgesteuerte Kommunikation <sup>26</sup>

## 2.4 Load Balancer

Ein Load Balancer welcher als Software- oder Hardware-Load Balancer angeboten wird, setzt die Lastverteilung in einem Netzwerk um. Ziel ist es Arbeitsbelastung auf Rechenressourcen wie zum Beispiel Servern gleichmäßig zu verteilen, um dadurch die Zuverlässigkeit, Effizienz und Kapazität des Netzwerkes zu optimieren. Hardware Load Balancer können als physische Maschine, welche in der Regel mit speziellen Prozessoren ausgestattet ist, erworben werden. Hardware Load Balancer können leistungsfähiger sein als Software Load Balancer. Dem gegenüber laufen Software Load Balancer virtuell auf handelsüblicher Hardware, sie sind preiswerter als Hardware Load Balancer und flexibel einsetzbar.

Ein Load Balancer ermittelt in Echtzeit, welche Rechenressource die entsprechende Clientanforderung erfüllen kann. Dabei soll eine Netzwerküberlastung vermieden werden. Es gibt mehrere Methoden, um Loadbalancing umzusetzen. Eine davon ist der Round Robin Algorithmus, welcher in sequenzieller Reihenfolge eine Liste der verfügbaren Rechenressourcen durchläuft und dadurch die Last gleichmäßig verteilt. Weitere Möglichkeiten bieten unter anderem der Hash basierte Ansatz, der Least-time-Algorithmus und die Least-Connection-Methode. <sup>27, 28</sup> Eine Veranschaulichung des Loadbalancings wird mit Abbildung 2 dargestellt.

<sup>24</sup> (TechTarget, 2019)

<sup>25</sup> (Ionos, 2019)

<sup>26</sup> (Nish, 2021)

<sup>27</sup> (ComputerWeekly, 2020)

<sup>28</sup> (Donner, et al., 2019)

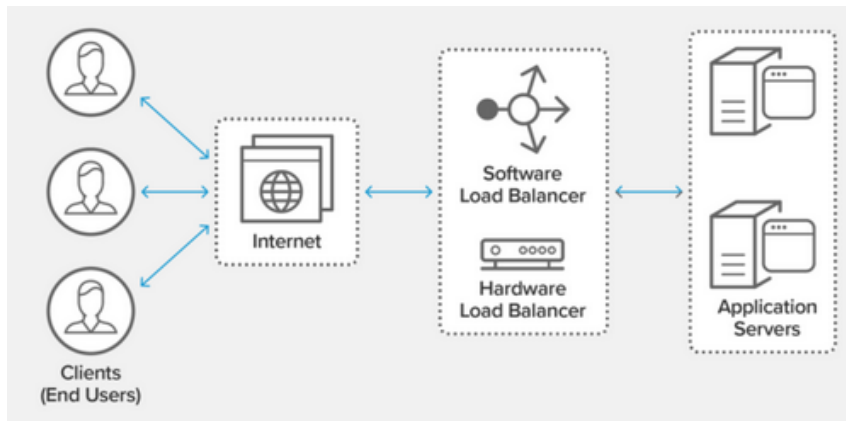


Abbildung 2 Loadbalancing <sup>29</sup>

## 2.5 Service Discovery

Zur Umsetzung von Skalierbarkeit und Wartbarkeit werden in einer Microservice-Architektur instanziierten Diensten dynamisch Netzwerkstandorte zugewiesen. In einem Netzwerk können sich diese Dienste jedoch nicht ohne weiteres gegenseitig finden. Laut Alexander S. Gillis identifiziert die Service Discovery Geräte und Dienste in einem verteilten Netzwerk. Die Umsetzung erfolgt auf Basis eines gemeinsamen Netzwerkprotokolls wie zum Beispiel der Domain Name System Service Discovery oder dem Dynamic Host Configuration Protocol. <sup>30</sup>

Service Discovery wird als Software realisiert, welche die Registrierung von Service-Instanzen in einem System ermöglicht. Bei Anfragen werden alle verfügbaren Ziele aus einer Liste mit den registrierten Instanzen abgerufen. Diese Liste, welche Adressen und Ports der Services beinhaltet, wird als Service Registry bezeichnet. Service Discovery ermöglicht es, Service Adressen über den Namen des aufgerufenen Service für den Client aufzulösen. Service-Adresse und Port werden dazu aus der Service Registry übermittelt. <sup>31</sup> Für die Umsetzung von Service Discovery gibt es unter anderem folgende Möglichkeiten.

### Clientseitig Erkennung

Beim Clientseitigen Erkennungsmuster ist der Client für die Erkennung der Netzwerkstandorte von Dienstinstanzen verantwortlich und kontaktiert diese über einen Lastenausgleichsalgorithmus. Der Client fragt die verfügbaren Dienstinstanzen aus der Service Registry ab. Netzwerk Instanzen werden beim Starten an der Registry angemeldet und beim Herunterfahren wieder abgemeldet. Die Registrierung wird in regelmäßigen Abständen automatisch überprüft. Die Client Side Discovery ist relativ unkompliziert. Sie hat jedoch den Nachteil, dass es den Client mit der Dienstregistrierung koppelt. Dadurch wird für jedes Framework und jede Programmiersprache die Implementierung einer Logik für die Diensterkennung erforderlich. Lösungen für Clientseitige Erkennung liefern zum Beispiel Netflix Eureka und Zookeeper.

<sup>29</sup> (NGINX)

<sup>30</sup> (Gillis, 2021)

<sup>31</sup> (Bayer, 2019)



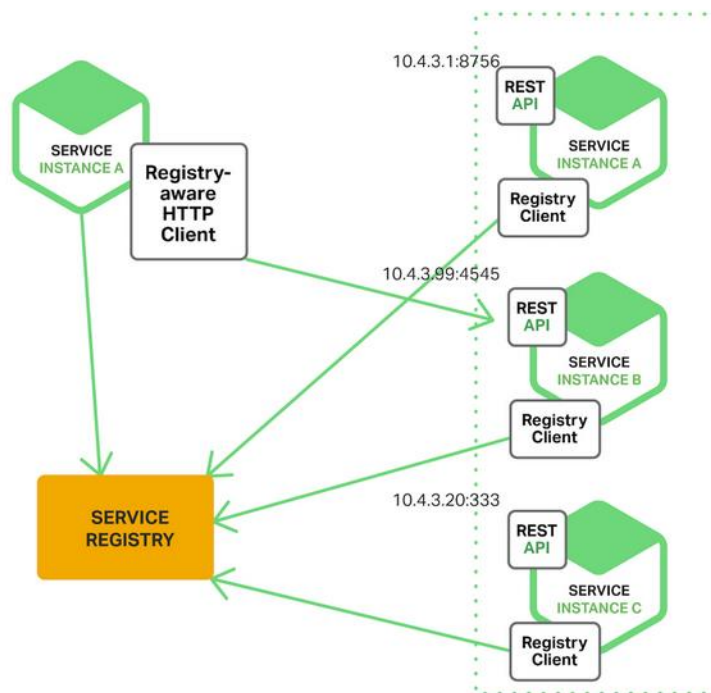


Abbildung 3 Clientseitige Erkennung <sup>32</sup>

### Serverseitige Erkennung

Bei der serverseitigen Erkennung nimmt ein Load Balancer die Anfragen entgegen und leitet diese an den entsprechenden Dienst weiter. Der Load Balancer fragt dazu die Service Registry ab. Server-side Discovery hat den Nachteil, dass der Load Balancer bei einem Ausfall das ganze System lahmlegen kann. Darüber hinaus stellt dieser eine weitere Komponente im Netzwerk dar. Dadurch entsteht bei der Kommunikation ein weiterer Hopp, was eine Verlangsamung zur Folge hat. Serverseitige Erkennung bietet den Vorteil, dass Abfragen für Clients vereinfacht werden. Lösungen für serverseitige Service Discovery liefern NGINX und AWS. Abbildung 4 zeigt den Ablauf einer Serverseitigen Erkennung.

<sup>33</sup>, <sup>34</sup>, <sup>35</sup>

<sup>32</sup> (Richardson, 2015)

<sup>33</sup> (Gillis, 2021)

<sup>34</sup> (Richardson, 2015)

<sup>35</sup> (NS1)

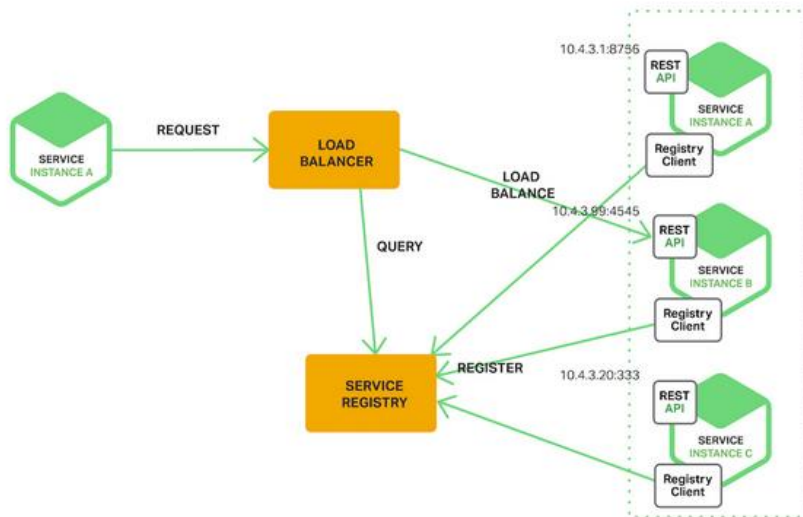


Abbildung 4 Serverseitige Erkennung<sup>36</sup>

## 2.6 API Gateway

Bei einer Microservice-Architektur kann es sich als problematisch erweisen, wenn der Client direkt mit den einzelnen Services kommuniziert. Dabei können unter anderem folgende Komplikationen auftreten

- Sicherheitsprobleme:  
Für eine direkte Kommunikation müssen alle Microservices für den Client offengelegt werden. Dieser muss die Adressen aller Services kennen. Es wird zum Beispiel eine große Angriffsfläche geboten, wenn auch Endpunkte für eine interne Kommunikation zwischen den Microservices (wie auf Abbildung 5 zwischen Microservice 1 und Microservice 2) für den Client bekannt gemacht werden.
- Enge Kopplung:  
direkte Verweise zwischen Client und Microservices führen zu einer engen Kopplung, aufgrund einer Vielzahl von Kommunikationsverbindungen. Dadurch verschlechtert sich die Wartbarkeit des Systems. Wenn zum Beispiel ein Service in zwei kleinere Services zerlegt wird, müssen alle Kommunikationsverbindungen beim Client angepasst werden.

Abbildung 5 veranschaulicht die direkte Kommunikation zwischen Client und mehreren Microservices

<sup>36</sup> (Richardson, 2015)

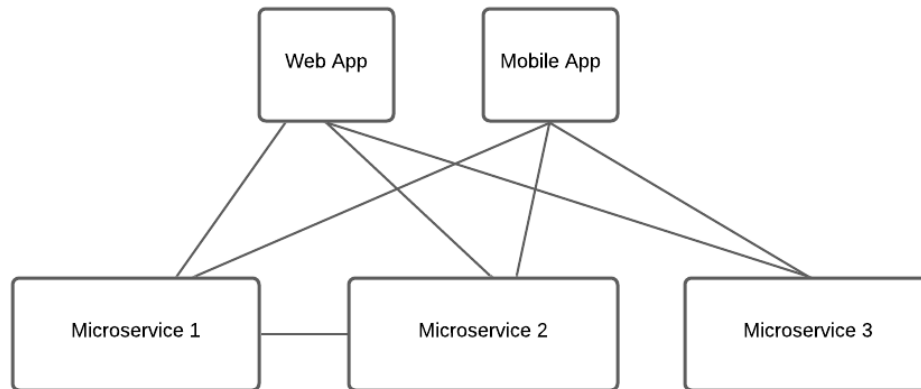


Abbildung 5 Kommunikation ohne Gateway <sup>37</sup>

Ein API Gateway schafft dem gegenüber Lösungsmöglichkeiten. Dieses gleicht bezüglich seiner Funktionalitäten einem Reverse Proxy. Es stellt jeweils einen Kontaktpunkt für ein- und ausgehenden Netzwerkverkehr bereit. Es bietet dazu ein vereinheitlichtes Interface, welches mit dem Client interagiert. Dementsprechend werden Gruppen interner Microservices unter einer einzigen URL bereitgestellt. Eine einzelne Clientanfrage kann mehrere Microservices aggregieren. Dadurch kann der Datenaustausch zwischen Backend und Client reduziert werden.

Zur Gewährleistung der Systemsicherheit bietet ein API Gateway die Möglichkeit der Umsetzung von Autorisierung und Authentifizierung an zentraler Stelle. Zusätzlich bietet es die Möglichkeit ein zentrales Logging für Requests durchzuführen.

---

<sup>37</sup> (Eigene Darstellung)

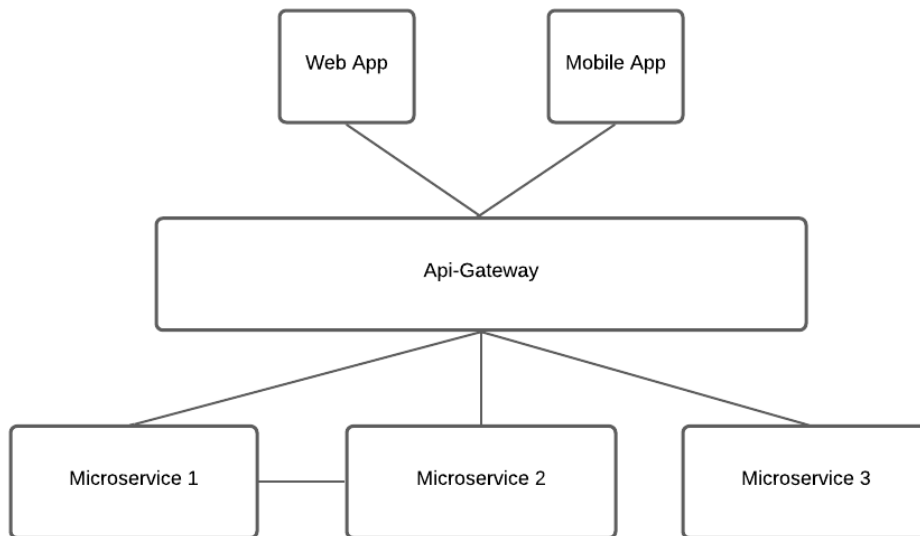


Abbildung 6 Kommunikation mit Gateway<sup>38</sup>

Der größte Nachteil beim Einsatz eines API Gateways ist, dass ein Ausfall das gesamte System lahmgelegt, weil alle Clientanfragen das Gateway passieren müssen. Daher muss es mindestens die gleiche Verfügbarkeit aufweisen wie die Services dahinter.

Lösungen für ein API Gateway bieten zum Beispiel Netflix Zuul, Spring-Cloud API Gateway, Ocelot und KrakenD. Weil sich der Austausch einer Gateway-Technologie als relativ unkompliziert gestaltet, bietet es sich an, vorerst ein möglichst leicht zu implementierendes Gateway zu verwenden. Auf eine komplexere Lösung kann gewechselt werden, wenn alle Anforderungen festgelegt wurden und diese den Einsatz eines Gateways mit größerem Funktionsumfang erfordern.<sup>39, 40</sup>

## 2.7 Autorisierung und Authentifizierung

Damit in einer Anwendung einem Nutzer die Möglichkeit geboten wird, seine Identität nachzuweisen und dem Nutzer nur die Daten zur Verfügung zu stehen, für deren Verwendung er berechtigt ist, bedarf es der Authentifizierung und Autorisierung. Diese Begriffe werden im folgenden Abschnitt erklärt.

### Authentifizierung

Authentifizierung beschreibt die Art und Weise, wie ein Benutzer oder ein System seine Identität nachweist. Die Umsetzung von Authentifizierung kann unter anderem über die Abfrage von biometrischen Daten oder Passwörtern durchgeführt werden.

---

<sup>38</sup> (Eigene Darstellung)

<sup>39</sup> (Anil, 2021)

<sup>40</sup> (sidion, 2019)

### **Autorisierung**

Autorisierung bedeutet zu prüfen, ob eine Entität für den Zugriff auf eine bestimmte Ressource oder Funktion berechtigt ist. Die Autorisierung kann unter anderem rollenbasiert erfolgen. Bei einer rollenbasierten Implementierung werden dem Benutzer Rollen zugewiesen wie zum Beispiel Dozent, Redakteur oder Administrator. Einem Service können ebenfalls Rollen als Voraussetzungen für dessen Ausführung zugewiesen werden. Der Benutzer kann einen Service nur aufrufen, wenn diesem die vom Service geforderten Rollen zugewiesen wurden. Ein Redakteur kann zum Beispiel dafür autorisiert sein, bestimmte Textinhalte einer Anwendung zu ändern.

### **OAuth2 und OpenId Connect**

Zur Umsetzung von Authentifizierung und Autorisierung in einer Microservice-Architektur bedarf es einer anderen Herangehensweise als wie bei einer monolithischen Architektur. Bei Monolithen werden häufig Sitzungen eingesetzt, welche auf dem Server gespeichert werden. Der Client erhält in diesem Fall eine Sitzungs-Id, welche bei Anfragen der Sitzung zugeordnet wird. Unter verteilten Anwendungen können Sitzungen jedoch nicht gemeinsam genutzt werden. Es ist auch nicht sinnvoll für jeden Service eine eigene Sitzung zu erstellen. In diesem Fall müsste sich der Benutzer dann zum Beispiel vor der Benutzung jedes Services Einloggen, um eine Sitzung zu speichern.

Der Einsatz eines Autorisierungsservers bietet unter Verwendung des OAuth2 Protokolls eine Lösung zur Umsetzung von Autorisierung in einem verteilten System. Dadurch kann eine Authentifizierung für die Verwendung mehrerer Microservices mit nur einem Login ermöglicht werden.

Unter Verwendung von OAuth2 können folgende Rollen definiert werden:

- Client – Die Anwendung, welche auf das Benutzerkonto zugreifen möchte
- Ressourcen Besitzer – Der Benutzer dem die Ressource gehört
- Ressourcen Server – verwaltet die geschützten Ressourcen und gewährt mit Hilfe eines Zugriffstokens den Zugriff.
- Autorisierungsserver – stellt nach einer erfolgreichen Identitätsprüfung ein Zugriffstoken an den Client aus

Auf einer hohen Abstraktionsebene lässt sich der Ablauf von OAuth2 wie folgt beschreiben:

1. Der Benutzer erhält von dem Client eine Anfrage auf die Zugriffsberechtigung der entsprechenden Dienstressourcen
2. Die Anfrage wird an den Autorisierungsserver weitergeleitet, welcher nach erfolgreicher Authentifizierung ein Zugriffstoken ausstellt
3. Der Client fordert die Ressource vom Ressourcenserver an. Die Authentifizierung erfolgt über das Zugriffstoken.
4. Bei der Gültigkeit des Tokens erhält die Anwendung vom Ressourcenserver die Ressource.

OpenId Connect ist ein Authentifizierungsprotokoll welches auf OAuth2 aufsetzt. Es bietet unter Verwendung von OAuth2 zusätzlich das Feature, einen Login über ein Fremdsystem zu ermöglichen. Dieses Verfahren wird als Single Sign-On bezeichnet. Single Sign-On ermöglicht es, sich von sozialen Netzwerkdiensten wie zum Beispiel Facebook, Twitter oder Xing aus, auf einer anderen Anwendung anzumelden. <sup>41, 42, 43,</sup>

44

## 2.8 Circuit Breaker Pattern

In einer Microservice-Architektur sind in der Regel unter der synchronen Kommunikation mehrere Services voneinander abhängig. Fällt einer dieser Services aus, dann warten abhängige Microservices nach Anfragen an diesen Service auf dessen Antwort. Während der Wartezeit können weitere Anfragen in das System eintreffen. Dieses Verhalten kann dazu führen, dass sich die Anfragen anstauen. Im schlimmsten Fall kann dadurch das gesamte System lahmgelegt werden. Das Circuit Breaker Pattern dient zur Vermeidung einer solchen Problemstellung. Die Funktionalität kann, mit der einer Sicherung in einem elektrischen Stromkreis verglichen werden. Fließt in einem Stromkreis ein zu hoher elektrischer Strom, dann unterbricht die Sicherung den Stromfluss, bevor weitere Komponenten des Systems Schaden nehmen können.

Ein Circuit Breaker kann zwischen den Zuständen offen, geschlossen und halb offen wechseln. Wird bei dem Aufruf einer Ressource eine bestimmte Fehlerrate aufgrund eines ausgefallenen Services überschritten, dann nimmt der Circuit Breaker den Zustand Offen an. In diesem Zustand werden die entsprechenden Aufrufe blockiert. Der ausgefallene Service soll dadurch die Möglichkeit erhalten, in einen fehlerfreien Zustand zurückzukehren, ohne dabei durch weitere Aufrufe blockiert zu werden. Tritt kein Fehler mehr auf, dann werden alle Anfragen wieder weitergeleitet und der Service ist verfügbar. Der Circuit Breaker ist in diesem Fall geschlossen. Einen Übergang zwischen den Zuständen offen und geschlossen bietet der Zustand halb offen, welcher nach einer gewissen Zeit im Zustand offen aktiviert wird. In diesem Zustand werden Anfragen teilweise weitergeleitet. Nach erfolgreichen Anfragen wird der Circuit Breaker wieder geschlossen. Sollten bei Anfragen weiterhin Fehler auftreten, ändert sich der Zustand wieder auf offen.

Der Einsatz eines Circuit Breakers bietet zusätzlich gute Möglichkeiten zur Überwachung des Systems. Jeder Zustandswechsel kann geloggt werden. Dadurch entsteht eine Möglichkeit tiefgreifende Fehlerquellen aufzuspüren. <sup>45,46</sup>

Die Funktionsweise des Circuit Breaker Patterns wird mit Abbildung 7 dargestellt.

---

<sup>41</sup> (te Wierik, 2020)

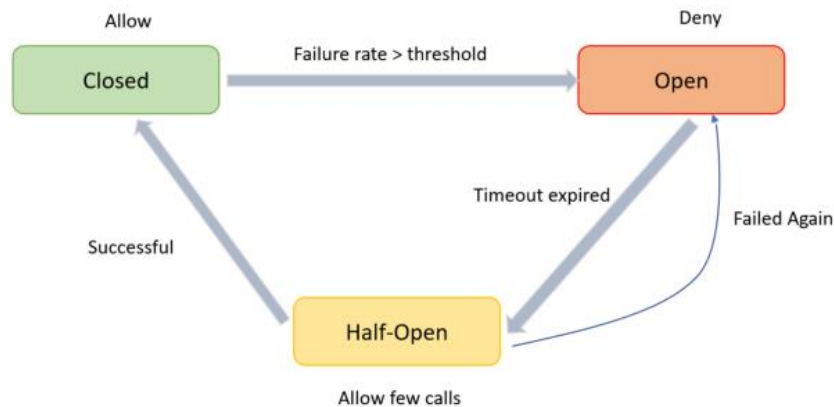
<sup>42</sup> (Anicas, 2014)

<sup>43</sup> (Securai)

<sup>44</sup> (Neal, 2019)

<sup>45</sup> (Richardson, 2021)

<sup>46</sup> (Kothari, 2020)

Abbildung 7 Circuit Breaker Funktionsweise <sup>47</sup>

## 2.9 Distributed Tracing

In einer Microservice-Architektur können jederzeit neue Instanzen eines Service erstellt oder gelöscht werden. Aufgrund dessen lassen sich Logs nicht zuverlässig nachverfolgen. Anfragen eines Clients können über eine Vielzahl von Microservices hinweg ausgeführt werden. Die Dauer für die einzelnen Aufrufe unter den Services lässt sich nicht ohne weiteres feststellen. Engpässe lassen sich dementsprechend kaum nachvollziehen. Bei einer Entwicklung mit einzelnen Entwicklerteams pro Microservice kann es sich als aufwendig erweisen herauszufinden, welches Entwicklerteam für einen Fehlerfall verantwortlich ist.

Distributed Tracing ist ein Verfahren welches zum Profilieren, Überwachen und Debuggen von verteilten Anwendungen eingesetzt wird. Funktionen, welche die App verlangsamen oder einen Ausfall verursachen, können anhand dieses Verfahrens ermittelt werden. Weiterhin kann das Distributed Tracing zum Optimieren von Code eingesetzt werden. Das Verfahren nutzt ein Konzept, bei dem Daten zwischen den Aufrufen unter den Microservices mitgeliefert werden. Diese werden als Span bezeichnet. Ein Span wird mit einer eindeutigen ID versehen. Verschachtelte Aufrufe unter den Microservices erhalten zusätzlich eine Parent Span ID, welche an die betreffenden Microservices übergeben wird. Zu Identifikation der übergeordneten Operationen generiert der erste Aufruf eine Root Trace ID. Der Fluss der Microservice Aufrufe wird also über Trace, Span und Parent Span ID beschrieben.

In einem Span werden folgende Informationen gespeichert:

- Zeitstempel für den Abschluss eines Zwischenschrittes
- Tags zur Klassifizierung per Schlüssel- / Wertpaar
- Textbasierte Anmerkungen zum Beispiel über Fehlermeldungen oder Informationen über die Beeinflussung des Flusses

<sup>47</sup> (medium, 2020)

Die Daten werden von einem Trace Server verarbeitet. In einer Web-UI werden die zeitlichen Abläufe der Aufrufe dargestellt. Diese können dann anhand der Informationen wie zum Beispiel Beginn und Dauer der einzelnen Spans ausgewertet werden. Dadurch entsteht die Möglichkeit, die Anwendung als Gesamtsystem zu überwachen. <sup>48, 49, 50</sup>

## 2.10 User Interface

### Frontend Monolith

Bei einem Frontend Monolithen teilen sich die Microservices einer Anwendung ein einzelnes Frontend. Laut Eberhard Wolff sollte der Einsatz eines Frontend Monolithen bei einer Microservice-Architektur stets hinterfragt werden, weil das Userinterface bei vielen fachlichen Änderungen verschiedener Microservices angepasst werden muss. Dadurch kann ein Frontend Monolith zu einem Änderungsschwerpunkt werden. Es kann jedoch Gründe geben, die für eine Umsetzung eines Monolithen im Frontend sprechen. Unter folgenden Voraussetzungen ist ein Monolithisches Frontend die richtige Wahl:

- **Single Page App**  
Diese Anwendungen sind heutzutage sehr populär und bieten die Möglichkeit für eine Modularisierung. Laut Eberhard Wolff führen sie allerdings mit der Zeit zu einem Monolithischen Frontend.
- **Native mobile Anwendung**  
Diese Anwendungen können nur als Ganzes deployt werden.
- **Frontend Entwicklerteam**  
Beim Einsatz eines Teams, welches speziell für die Frontendentwicklung spezialisiert ist bietet sich die Umsetzung eines Monolithen an. Dadurch kann das Team in gewohnter Arbeitsumgebung agieren.

51

Abbildung 8 visualisiert eine Microservice-Architektur mit einem monolithischen Frontend.

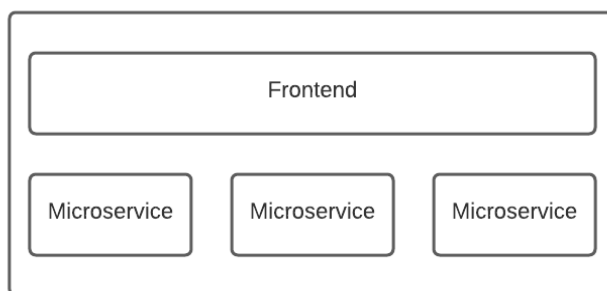


Abbildung 8 Frontend Monolith <sup>52</sup>

---

<sup>48</sup> (NewRelic, 2020)

<sup>49</sup> (Guimaraes, 2019)

<sup>50</sup> (Augsten, 2021)

<sup>51</sup> (Wolff, 2018)

<sup>52</sup> (Eigene Darstellung)



## Modularisiertes Frontend

Bei einem modularisierten Frontend erhält jeder Microservice sein eigenes Frontend. Dadurch können die Microservices fachlich unabhängig gehalten werden. Weiterhin entstehen bei modularisierten Frontends keine Einschränkungen bei der Wahl von Technologie Entscheidungen. Ein modularisiertes Frontend droht nicht zu einem Änderungsschwerpunkt zu werden. Der Wechsel unter den einzelnen Anwendungen wird durch Verlinkungen umgesetzt. Der Benutzer wechselt über Verlinkungen zwischen den einzelnen Systemen. Unter der Verwendung eines einheitlich gehaltenen Designs bemerkt dieser nichts von einem Wechsel.<sup>53</sup>

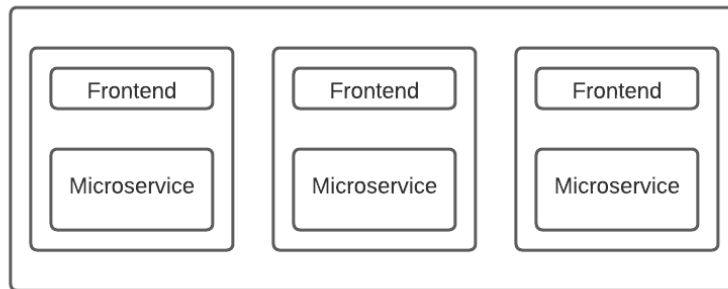


Abbildung 9 Modulares Frontend<sup>54</sup>

## 2.11 Container

Damit die Vorteile von Microservices ausgenutzt werden können, müssen diese laut Eberhard Wolff mindestens getrennte Prozesse sein. Es soll dadurch vermieden werden, dass ein Absturz eines Microservices zum Absturz weiterer Microservices führt. Das System bleibt dabei robust. Für die Gewährleistung der Skalierbarkeit eines Systems ist diese Trennung allerdings nicht ausreichend. Laufen mehrere Prozesse auf einem Server, dann steht nur eine begrenzte Menge an Hardwarekapazität zur Verfügung. Kompatibilitätsprobleme mehrerer Bibliotheken auf nur einem Betriebssystem führen zu weiteren Hindernissen.<sup>55</sup>

Eine Möglichkeit zur Lösung dieser Probleme bieten virtuelle Maschinen. Diese ermöglichen es, auf einem Rechner mehrere Betriebssysteme wie zum Beispiel Windows und Linux zur gleichen Zeit laufen zu lassen. Die Aufteilung der Microservices auf virtuellen Maschinen beanspruchen allerdings viel Speicher, weil dementsprechend jeder Microservice die Instanz eines Betriebssystems besitzt. Eine effizientere Lösung bietet der Einsatz von Containern. Diese bieten einen universellen Paketierungsansatz, bei dem alle Anwendungsabhängigkeiten in einem Container gebündelt werden. Container bringen gegenüber Virtuellen Maschinen mehrere Vorteile mit sich. Beim Einsatz von Containern wird kein ganzes Betriebssystem installiert, sondern nur alle zum Ausführen notwendigen Pakete einer Anwendung. Dadurch bleibt ein Microservice so leichtgewichtig wie ein Prozess. Dementsprechend wird die Erzeugung des Overheads vermieden welcher zum Beispiel beim Ausführen von Softwarekomponenten wie Webserver oder Datenbanken bei

---

<sup>53</sup> (Wolff, 2018)

<sup>54</sup> (Eigene Darstellung)

<sup>55</sup> (Wolff, 2018)

einer virtuellen Maschine entsteht. Darüber hinaus lässt sich ein Container auch deutlich schneller starten als eine virtuelle Maschine. Bei dieser muss erst das komplette Betriebssystem hochgefahren werden. Container lassen sich daher schneller aufsetzen als virtuelle Maschinen und Entwicklern werden neue Möglichkeiten im Deployment geboten. Container können zum Beispiel im System für einen Lastenausgleich sorgen, indem je nach Systemlast, Container und die darüber laufenden Instanzen eines Services, zu- oder abgeschaltet werden. Dieser Vorgang lässt sich automatisieren, wodurch der IT-Betrieb auf lange Sicht entlastet wird. <sup>56, 57, 58</sup>

---

<sup>56</sup> (Wolff, 2018)

<sup>57</sup> (Preissler, et al., 2015)

<sup>58</sup> (Öggl, et al., 2019)

## 3 Anforderungsanalyse

### 3.1 Aufgabenstellung

Für die Fachhochschule Erfurt soll ein System zur Verwaltung der jährlichen IT-Kontaktmesse an der Fachhochschule Erfurt umgesetzt werden. Das System wird unter dem Namen *IT-KoM Verwaltung*, in Form einer Webseite realisiert, welche die folgenden Inhalte abdeckt:

#### **Firmenverwaltung**

Firmen können folgende Features des Systems nutzen:

- Registrierung
- Newsletter Abonnieren (Firmen erhalten dadurch regelmäßig Informationen über die Messe per E-Mail.)
- Anmeldung / Abmeldung für die Teilnahme als Messeaussteller
- Verwaltung von Vorträgen (Themen, Präsentatoren, Zeitslots, benötigtes Equipment)
- Informationsbereitstellung für Besucher (Firmenbeschreibung, Link zur Webseite, Ansprechpartner, Vorträge)

#### **Informationsbereitstellung für Besucher**

Das System soll einen öffentlichen Zugang für Besucher in Form einer Webseite bieten. Diese sollen sich über folgende Themen informieren können:

- Allgemeine Messedaten (Veranstaltungsort, Zeitpunkt und Ansprechpartner)
- Informationen über ausstellende Firmen (Firmenbeschreibung, Lage des Messestandes, Link zur Webseite, Ansprechpartner)
- Information zu Vorträgen (Thema, Firma, Zeitpunkt, Gebäudenummer und Raumnummer)

#### **Administration**

Für Mitarbeiter der Fachhochschule wird eine Oberfläche zur Verwaltung des Systems geboten. Folgende Verwaltungsmöglichkeiten werden gegeben:

- Allgemeine Messedaten (Veranstaltungsort, Zeitpunkt und Ansprechpartner)
- An / Abmeldung der Messeaussteller
- Anlegen von Zeitslots für Vorträge (Zeitslot für zugehörigen Vortragsraum)
- Bearbeitung und Versendung des Newsletters (Sendung per Broadcast an Abonnenten)
- Löschen registrierter Firmenaccounts

### 3.2 Qualitätsziele

Laut Dr. Peter Hruschka und Dr. Gernot Starke beeinflusst die Ernennung der für die Stakeholder wichtigsten Qualitätsziele, die Softwarearchitektur maßgebend.<sup>59</sup> Daher werden im folgenden Abschnitt die wichtigsten Qualitätsziele aufgelistet.

#### **Wartbarkeit (Modularität)**

Die Fachhochschule Erfurt wird voraussichtlich noch über viele Jahre (Jahrzehnte) die Fachrichtung Angewandte Informatik und die dazugehörige IT-Kontaktmesse anbieten. Das System wird im Laufe der Zeit stark anwachsen. Das System müsste im schlimmsten Fall bei weitreichenden Änderungen komplett neu geschrieben werden, falls zwischen den einzelnen Modulen zu starken Abhängigkeiten bestehen. Um die Wartbarkeit auf Dauer sicherzustellen, wird dieses Qualitätsziel mit hoher Priorität eingestuft.

#### **Zuverlässigkeit (Verfügbarkeit)**

Das System soll im Besonderen, während des Zeitraumes vor der Messe eine hohe Verfügbarkeit sicherstellen. Lange Ausfallzeiten würden während dieses Zeitraumes den Informationsaustausch zwischen Fachhochschule, Firmen und Studierenden blockieren. Die Akteure würden unter Umständen keine Informationen über Änderungen des Ablaufes der Messe erhalten oder diese Informationen zu spät erhalten. Deshalb wird ein hoher Wert auf die Verfügbarkeit des Systems gelegt.

#### **Integrität**

Sollten zwischen der Fachhochschule Erfurt, den teilnehmenden Firmen und Studierenden Missverständnisse entstehen könnte im schlimmsten Fall die gesamte Messe scheitern. Beispielsweise würden die Messeaussteller an einer Messe ohne Besucher teilnehmen, wenn Besucher nicht über eine Änderung des Messetermins informiert werden.

#### **Bedienbarkeit (Einfache Benutzung / Erlernbarkeit, Zugänglichkeit)**

Eine schlechte Bedienbarkeit könnte unter anderem dazu führen, dass weniger Studierende an der Messe teilnehmen. Zum Beispiel könnte die Organisation der Messe aufgrund einer falschen Bedienung aus dem Ruder laufen. Darüber hinaus könnten Studierende aufgrund einer schlechten Bedienbarkeit nicht die benötigten Informationen für eine Teilnahme an der Messe erhalten. Damit die genannten Beispiele vermieden werden, wird die Bedienbarkeit als sehr wichtig betrachtet. Zusätzlich wird Wert daraufgelegt, dass auch Benutzer mit körperlichen Einschränkungen das System verwenden können.

---

<sup>59</sup> (Hruschka, et al., 2017)

### 3.3 Stakeholder

Im folgenden Abschnitt werden die wichtigsten Akteure ermittelt, welche mit dem Projekt in Verbindung stehen. Es werden Erwartungen und Einflüsse der Akteure erfasst, um Probleme rechtzeitig zu erkennen und zu lösen. In folgender Tabelle werden alle Stakeholder abgebildet, die mit dem Projekt in Verbindung stehen.

Rolle	Beschreibung	Erwartungshaltung
FH-Sekretariat	<ul style="list-style-type: none"> <li>Anlaufstelle für Studierende und StudiumsbewerberInnen</li> <li>Verwalten die Studierenden und beantworten deren Fragen</li> </ul>	<ul style="list-style-type: none"> <li>Möchte eine leicht zu bedienende Administrationsoberfläche zur Verwaltung der IT-Kontaktmesse</li> </ul>
Dozenten	<ul style="list-style-type: none"> <li>Halten Lehrveranstaltungen an der Hochschule</li> </ul>	<ul style="list-style-type: none"> <li>Möchte eine leicht zu bedienende Administrationsoberfläche zur Verwaltung der IT-Kontaktmesse</li> </ul>
Studenten	<ul style="list-style-type: none"> <li>Studieren an der Hochschule</li> </ul>	<ul style="list-style-type: none"> <li>Möchten sich möglichst unkompliziert mit wenigen Klicks über Ort und Zeitpunkt der Messe, teilnehmende Firmen und deren Jobangebote informieren</li> <li>Möchten anhand der Messe Jobangebote von den ausstellenden Firmen erhalten.</li> </ul>
Studiumsinteressenten	<ul style="list-style-type: none"> <li>Sind an einem Studium der Angewandten Informatik an der Fachhochschule Erfurt interessiert</li> </ul>	<ul style="list-style-type: none"> <li>Möchten sich möglichst unkompliziert mit wenigen Klicks über Ort und Zeitpunkt der Messe, teilnehmende Firmen und deren Jobangebote informieren</li> <li>Erwarten eine gut organisierte Messe, um sich zu vergewissern, dass ein Studium an der Fachhochschule Erfurt die richtige Entscheidung ist.</li> </ul>
Firmenvertreter	<ul style="list-style-type: none"> <li>Nehmen als Ansprechpartner Ihres Unternehmens an den Firmeneigenen Messeständen Teil.</li> </ul>	<ul style="list-style-type: none"> <li>Möchten möglichst unkompliziert mit wenigen Klicks ihre Messeteilnahme verwalten und Studenten / Studiumsinteressenten</li> </ul>

		<p>Informationen bereitstellen</p> <ul style="list-style-type: none"> <li>• Möchten frühestmöglich über den organisatorischen Ablauf der Messe informiert werden. Dabei sollen keine Unklarheiten entstehen.</li> </ul>
Präsentatoren	<ul style="list-style-type: none"> <li>• Halten Vorträge für die Besucher der Messe</li> </ul>	<ul style="list-style-type: none"> <li>• Möchten möglichst unkompliziert mit wenigen Klicks ihre Präsentationen verwalten und Studenten / Studiumsinteressenten Informationen über die Vorträge bereitstellen</li> <li>• Möchten frühestmöglich über den organisatorischen Ablauf der Messe informiert werden. Dabei sollen keine Unklarheiten entstehen.</li> </ul>
Entwickler	<ul style="list-style-type: none"> <li>• Entwickeln und Warten die Anwendung</li> </ul>	<ul style="list-style-type: none"> <li>• Wollen möglichst klar definiert Arbeitsaufgaben</li> <li>• Wollen Möglichkeiten zur Überwachung des Systems</li> <li>• Wollen zur Entwicklung keine veralteten Technologien verwenden</li> </ul>
Administratoren	<ul style="list-style-type: none"> <li>• Pflegen die Server auf denen die Anwendung bereitgestellt wird</li> </ul>	<ul style="list-style-type: none"> <li>• Möchten die Anwendung auf möglichst zuverlässigen Servern betreiben</li> <li>• Wollen Möglichkeiten zur Überwachung des Systems</li> <li>• Möchten ein Skalierbares System</li> </ul>

## 4 Architekturentwurf

### 4.1 Lösungsstrategie

#### **Allgemeine Architektur**

Die Anwendung IT-KoM Verwaltung wird für eine Messe entwickelt, welche auf unbestimmte Zeit jährlich an der Fachhochschule Erfurt stattfindet. Das System wird über Jahre hinweg ausgebaut. Die einzelnen Komponenten müssen deshalb aus langer Sicht wartbar bleiben. Dieser Zustand wird erreicht, wenn Abhängigkeiten unter den Komponenten minimal gehalten werden. Bei einem monolithischen System wäre die Wartung im Laufe der Zeit immer schwieriger zu handhaben. Daher wird eine Microservice-Architektur, welche aus einzelnen möglichst unabhängigen Systemen besteht, umgesetzt. Neue Features, wie zum Beispiel ein Chat zwischen Hochschule und einzelnen Firmen könnte relativ schnell als neuer Microservice hinzugefügt werden. Diesem kann ein eigenes Entwicklerteam zugewiesen werden, wodurch sich die Erweiterung unabhängig vom Rest der Anwendung entwickeln lässt.

Die einzelnen Microservices werden möglichst nach den Richtlinien von self-contained-systems entworfen. Diese stellen einen Architekturansatz dar, welcher sich auf eine Trennung der Funktionalitäten in viele unabhängige Teilsysteme konzentriert (unter anderem mit jeweils eigenem User Interface). Dadurch soll es möglich sein bei Ausfällen eines Services, den Rest des Systems in Betrieb zu halten ohne dass der Nutzer beim verwenden Anderer Services beeinträchtigt wird. Stehen zum Beispiel gerade keine Vorträge aufgrund eines Ausfalls zur Verfügung, dann soll der Nutzer den Rest des Systems wie zum Beispiel Informationen über teilnehmende Firmen abrufen können. Weiterhin könnte zum Beispiel die Möglichkeit zum Versenden des Newsletters gerade nicht zur Verfügung stehen, weil der entsprechende Microservice aufgrund von Wartungsarbeiten im Frontend neu gestartet wird. In diesem Fall wäre der Rest des Systems noch immer voll funktionsfähig.<sup>60</sup>

Zum deployen der Microservices kommt eine Containerengine zum Einsatz. Dadurch können die Microservices relativ schnell und einfach bereitgestellt werden. Die Container sollen auf jeder Umgebung mit der entsprechenden Containerengine funktionieren. Diese können dadurch auf den verschiedensten Maschinen ausgeführt werden. Weiterhin wird die Möglichkeit geboten einen automatisierten Lastenausgleich umzusetzen, was zur Verbesserung der Verfügbarkeit beiträgt und die Administration entlastet.

#### **Frontend**

Das Frontend steht als reine Webapp zur Verfügung. Die Entwicklung einer nativen App (welche speziell für mobile Geräte umgesetzt wird) ist für einen Prototyp vorerst nicht erforderlich. In einer späteren Erweiterung könnte eine native App entwickelt werden. Damit die Bedienbarkeit der Benutzeroberfläche für die zugehörigen Services optimiert werden kann, wird das Frontend modular entwickelt. Das bedeutet jeder Microservice wird zusammen mit einem eigenen Frontend deployt. Dadurch wird auch die Wartbarkeit auf Dauer verbessert, weil es kaum Abhängigkeiten unter den einzelnen Frontends gibt.

---

<sup>60</sup> (scs-architecture)

Dadurch können die Microservices nach den Standards von self contained systems entwickelt werden.

Für das Frontend wird eine serverseitige Template-Engine eingesetzt. Es wird unterschieden zwischen serverseitigen- und clientseitigen Rendern. Serverseitiges Rendern bietet den Vorteil dass die Seite beim ersten Laden schneller zur Verfügung steht. Bei clientseitigen Rendern entstehen Nachteile bezüglich der Suchmaschinen-optimierung. Suchmaschinen können ohne weiteres keine Inhalte erkennen, welche über JavaScript dynamisch generiert werden. Daher wird ohne entsprechenden Konfigurationsaufwand eine Clientseitig gerenderte Webseite von einer Suchmaschine schlechter gerankt als eine serverseitig gerenderte. Damit möglichst viele Interessenten der Messe die Besucherwebseite über eine Suchmaschine finden können, ist ein serverseitiges Rendering zu bevorzugen. Serverseitiges Rendern ist ein ausgefeiltes Konzept und daher relativ leicht umzusetzen. Der hohe Konfigurationsaufwand beim Einsatz einer serverseitigen Single-Page-Application wird vermieden. Serverseitiges Rendern hat den Nachteil, dass es bei weiteren Seitenaufrufen jedes Mal die komplette Seite lädt, während beim Clientseitigen Rendern nur einzelne Komponenten neu geladen werden. Die Anwendung erfordert jedoch keine komplexe Benutzeroberfläche. Viele Seitenaufrufe wie zum Beispiel bei einem Bestellprozess eines Onlineshops sind nicht notwendig. Daher wird das Rendering des Frontends serverseitig umgesetzt.<sup>61, 62</sup>

Für die Benutzer lässt sich die Oberfläche grob in drei Bereiche einteilen. Diese sind eine öffentliche Webseite für Besucher, eine Verwaltungsoberfläche für die teilnehmenden Firmen und ein Administrationsbereich für die Hochschule. Die Bereiche dürfen von Aussehen und Handhabung voneinander abweichen. Nach Möglichkeit sollten sie aber einheitlich gehalten werden. Innerhalb dieser Bereiche muss das Design einheitlich bleiben. Die einzelnen Entwicklerteams müssen daher in enger Absprache stehen und festgelegte Stylekonventionen einhalten. Stylesheets könnten zentral platziert und entwickelt werden. Die Anwendung würden sich in diesem Fall die eigenen Styles herunterladen. Das ermöglicht den Einsatz eines Entwicklerteams für Styles. Es wurde sich gegen diese Herangehensweise entschieden, weil auch dadurch im System eine Art Flaschenhals entsteht, was der Idee von Microservices widerspricht.

### Backend

Innerhalb des Systems kommunizieren Backend Komponenten untereinander synchron per REST über HTTP. Die einzelnen Microservices werden als Container deployt. Dadurch bleiben diese skalierbar. Weiterhin können sie schnell und Ressourcenschonend bereitgestellt werden. Außer dem Besucherservice und dem Administrationsservice enthält jeder Microservice eine eigene Datenbank. Abhängigkeiten unter der Logik zwischen den Microservices werden vermieden. Die Datenbank wird unter anderem anhand der Use Cases und technischen Gegebenheiten für jeden Microservice individuell gewählt. Es können im System zum Beispiel relationale und nicht-relationale Datenbanken für verschiedene Microservices zum Einsatz kommen. Die Entwicklerteams stehen bei der Wahl der Datenbanken im engen Kontakt, damit der Einsatz von zu vielen verschiedenen Datenbanken vermieden wird. Durch den Einsatz von zu vielen Datenbanktechnologien

---

<sup>61</sup> (Koller, 2018)

<sup>62</sup> (Shah, 2020)



würde das System für Entwickler unnötig komplex gehalten werden, weil diese sich in zu viele Technologien einarbeiten müssten.

Als Programmiersprache unter den Microservices ist Go auf dem Vormarsch. Es bietet eine besonders hohe Performance für nicht allzu komplexe Use Cases. Es bietet das hochkommunikative Kommunikationsframework gRPC und eignet sich deshalb zur Umsetzung von massiver Skalierbarkeit.

Eine weitere sehr verbreitete Sprache ist Java. Diese bietet teilweise eine schlechtere Datenverarbeitung als Go. Dafür werden Entwicklern mehr Möglichkeiten in Form von vielfältigen Tools geboten. Deshalb wird als Programmiersprache Java gewählt. Auf die Vorteile von Go wird verzichtet, weil die Anwendung keine extrem hohen Zeitanforderungen wie zum Beispiel bei einem Echtzeitsystem voraussetzt.<sup>63, 64</sup>

## 4.2 Systemkontext (Ebene 0)

Im folgenden Abschnitt werden das Umfeld des Systems, Nutzer des Systems, sowie die Fremdsysteme welche mit dem System interagieren beschrieben.

Das System wird für folgende Nutzer entworfen

- **Firma (Aussteller)**  
Vertreter von Firmen nutzen das System, um sich für die Teilnahme an der Messe zu registrieren, den eigenen Messeauftritt zu verwalten und den Newsletter zu abonnieren.
- **FHE-Mitarbeiter**  
Zu den FHE-Mitarbeitern zählen Sekretariat-Mitarbeiter und Dozenten. Diese verwalten die Messedaten, teilnehmende Firmen und Vorträge über eine Administrationsoberfläche. Zusätzlich erstellen und versenden sie den Newsletter.
- **Besucher**  
Zu den Besuchern gehören Personen, die sich auf dem öffentlichen Bereich der Webseite über die Messe informieren wollen. Dazu gehören zum Beispiel Studenten und Studieninteressierte.

Folgende Fremdsysteme interagieren mit dem System:

- **FHE-Webseite**  
Die Webseite der Fachhochschule Erfurt kann Daten des Systems aufrufen und bereitstellen.
- **AI-Webseite**  
Die Webseite der Angewandten Informatik kann Daten des Systems aufrufen und bereitstellen.
- **FHE-Mail**  
Der Newsletter für die Messe wird über den Mailserver der Fachhochschule Erfurt versendet. Das System sendet die Newsletter per Broadcast an alle Firmen, die den Newsletter abonniert haben.

---

<sup>63</sup> (RubyGarage, 2019)

<sup>64</sup> (Weigend, 2019)

Der Systemkontext wird auf Abbildung 10 dargestellt.

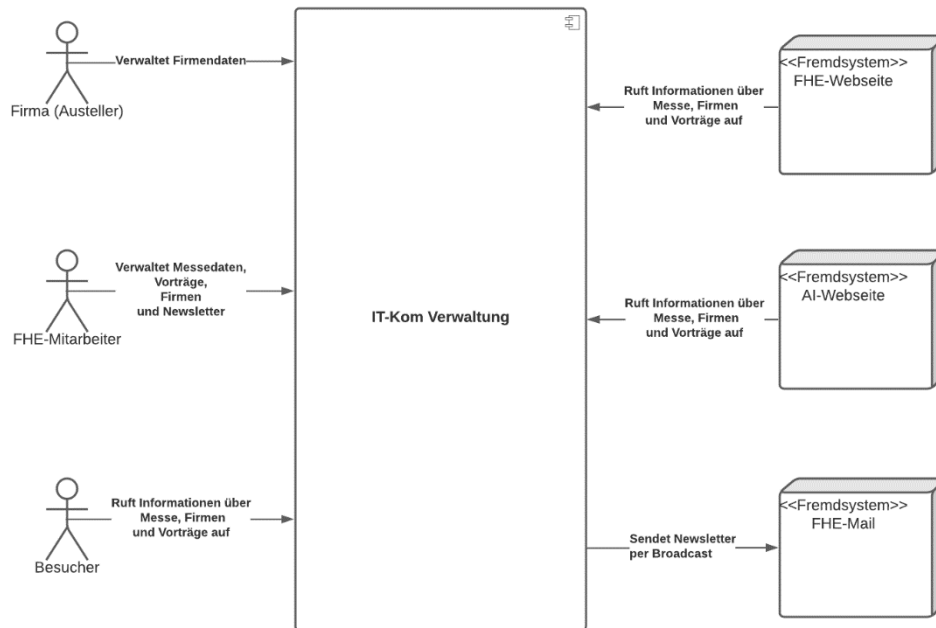


Abbildung 10 Systemkontext <sup>65</sup>

### 4.3 Domain Driven Design

Domain Driven Design ist eine Sammlung von zusammenhängenden Entwurfsmustern, welche im gleichnamigen Buch Domain Driven Design von Eric Evans beschrieben werden. Es hilft dabei, Microservices zu verstehen, weil es dabei um die Strukturierung größerer Systeme nach Fachlichkeit geht. Es wird anhand von Strategic Design beschrieben, wie komplexe Systeme aufgebaut werden können und Domänenmodelle miteinander interagieren. Bounded Context stellt dabei einen zentralen Punkt des Strategic Designs dar. Bounded Context beschreibt den gültigen Einsatzbereich für ein Domänenmodell und stellt einen in sich geschlossenen Fachbereich dar. Zum Beispiel steht ein Artikel für die Versandabteilung eines Onlineshops in einem anderen Kontext als für die Buchhaltung des Shops. Die Versandabteilung betrachtet unter anderem die Maße des Artikels. Für die Buchhaltung sind zum Beispiel Preise und Steuersätze von Bedeutung. <sup>66</sup> Laut Arne Limburg und Lars Röwekamp sollte ein Microservice einen Bounded Context abbilden. Dieser führt bei einer Einteilung nach Domänenobjekt selten zum Ziel. Eine bessere Lösung bietet die Einteilung nach Anwendungsfällen. <sup>67</sup> Für die Erstellung des Verwaltungsprogramms der IT-Kontaktmesse an der Fachhochschule Erfurt, lassen sich unter anderem Bounded Contexts wie auf Abbildung 11 darstellen:

<sup>65</sup> (Eigene Darstellung)

<sup>66</sup> (Wolff, 2018)

<sup>67</sup> (Röwekamp, et al., 2016)

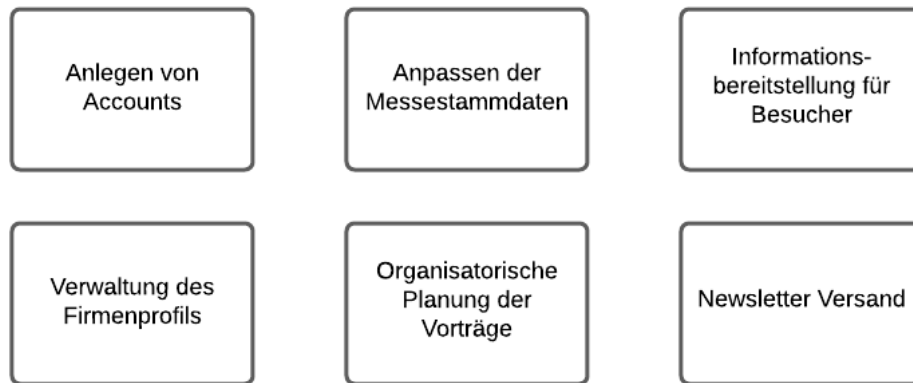


Abbildung 11 Bounded Contexts <sup>68</sup>

Daraus können folgende Microservices abgeleitet werden:

- Accountservice
- Messestammdatenservice
- Besucherservice
- Firmenservice
- Vortragsservice
- Newsletterservice

Für die Interaktion zwischen den Bounded Contexts sorgt im besten Fall ein Eventsystem.

<sup>69</sup> Im Idealfall sollte ein Microservice nur aus einem Bounded Context bestehen. Dadurch wird das Ziel erreicht, dass ein Team an einem Microservice unabhängig arbeiten kann. <sup>70</sup>

#### 4.4 Bausteinsicht (Ebene 1)

Im folgenden Abschnitt werden die Subsysteme der Anwendung dargestellt. Dazu gehören die einzelnen Microservices und weitere Subsysteme, welche für den reibungslosen Betrieb beitragen. Die Microservices wurden auf Basis der Bounded Contexts bestimmt. Diese werden als Container Deployt. Die Kommunikation unter den Microservices erfolgt per HTTP über Restschnittstellen.

Folgende Subsysteme gehören zur Anwendung:

- **API Gateway**  
Leitet Anfragen, welche in das System eintreffen von zentraler Stelle aus an entsprechende Services weiter. Ein integrierter Load Balancer erzeugt beim Aufruf eines Microservice mit mehreren Instanzen einen Lastenausgleich.
- **Service Discovery**  
Ermöglicht eine Adressauflösung über die Namen der einzelnen Services.

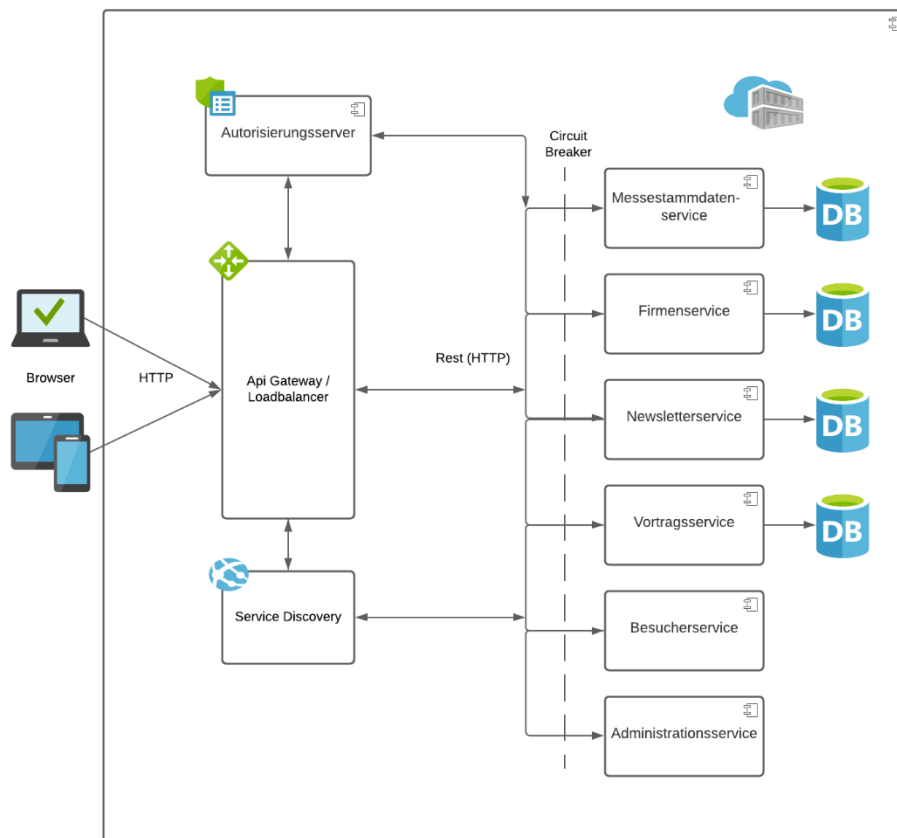
---

<sup>68</sup> (Eigene Darstellung)

<sup>69</sup> (Plöd, 2016)

<sup>70</sup> (Wolff, 2018)

- **Autorisierungsserver**  
Setzt das OAuth2 Protocol um und ermöglicht die Umsetzung von Autorisierung und Authentifizierung im System.
- **Circuit Breaker**  
Wird von Microservices implementiert, die mit anderen Microservices synchron kommunizieren. Der Circuit Breaker blockiert fehlerhafte Anfragen, um eine Überlastung des Systems zu verhindern.
- **Messestammdatenservice**  
Enthält Methoden, Datenstrukturen und ein User Interface zur Verwaltung der Messestammdaten.
- **Firmenservice**  
Enthält Methoden, Datenstrukturen und ein User Interface zur Verwaltung der Firmendaten.
- **Newsletterservice**  
Enthält Methoden, Datenstrukturen und ein User Interface zur Verwaltung des Newsletters.
- **Vortragsservice**  
Enthält Methoden, Datenstrukturen und ein User Interface zur Verwaltung der Vorträge.
- **Besucherservice**  
Enthält Methoden zum Aufrufen der Daten von den Microservices Messestammdatenservice, Firmenservice und Vortragsservice. Der Service wird verwendet, um die Daten mehrerer Microservices wie zum Beispiel Firmen- und Vortragsdaten auf einer View auszugeben. Er dient dazu um die synchrone Kommunikation zwischen den Microservices zu demonstrieren.
- **Administrationsservice**  
Enthält Methoden zum Aufrufen und Ändern der Daten von den Microservices Messestammdatenservice, Firmenservice, Vortragsservice und Newsletterservice. Der Service wird verwendet, wenn die Daten mehrerer Microservices im Administrations- oder Firmenverwaltungsbereich über eine View ausgegeben werden.

Abbildung 12 Bausteinsicht Ebene 1 <sup>71</sup>

## 4.5 Bausteinsicht (Ebene2)

Im folgenden Abschnitt werden die Module der Microservices Messestammdatenservice, Firmenservice, Newsletterservice, Vortragsservice, Administrationsservice und Besucherservice vorgestellt. Diese enthalten folgende Module:

- Controller**  
 Ein Controller stellt eine REST API bereit. Dabei werden Anfragen von der Benutzeroberfläche entgegengenommen und verarbeitet. Die Microservices kommunizieren untereinander über die API.
- Resources**  
 Hier lagern die Frontendelemente wie zum Beispiel HTML-, CSS- und Javascriptdateien. Zusätzlich beinhaltet das Modul die Konfigurationsdateien, in denen unter anderem der Applikationsname und Server-Port definiert werden. Diese dienen zum Beispiel für die Umsetzung der Service Discovery.

<sup>71</sup> (Eigene Darstellung)

- **Model**  
Ein Model enthält die jeweiligen Datenmodelle eines Microservices. Es wird zum Erstellen, Lesen und Bearbeiten von Daten verwendet.
- **Repository**  
Setzt die Datenbankzugriffe um
- **Datenbank (nicht im Besucherservice und Administrationsservice)**  
Speichert die Daten des Services. Die Wahl der Datenbank hängt von den Anforderungen an den jeweiligen Service ab. Je nach Anwendungsfall könnte zum Beispiel eine Relationale Datenbank oder eine NoSQL Datenbank (welche einen nicht-Relationalen Ansatz verfolgt) verwendet werden. Für den Prototyp wird jeweils eine H2 Datenbank verwendet. Es handelt sich dabei um eine in Memory Datenbank, welche sich besonders leicht einrichten lässt und daher für die Implementierung eines Prototyps geeignet ist. Besucherservice und Administrationsservice enthalten keine eigene Datenbank, weil diese nur Daten anderer Services verwenden. Der Besucherservice dient lediglich dazu, die Daten anderer Microservices den Besuchern auf der von ihm bereitgestellten Besucherwebseite gebündelt anzuzeigen. Der Administrationsservice bietet Mitarbeitern der Hochschule die Möglichkeit, die Daten anderer Microservices über eine Administrationsoberfläche gebündelt zu lesen oder anzupassen.

Der Aufbau der einzelnen Microservices wird auf Abbildung 13 dargestellt.

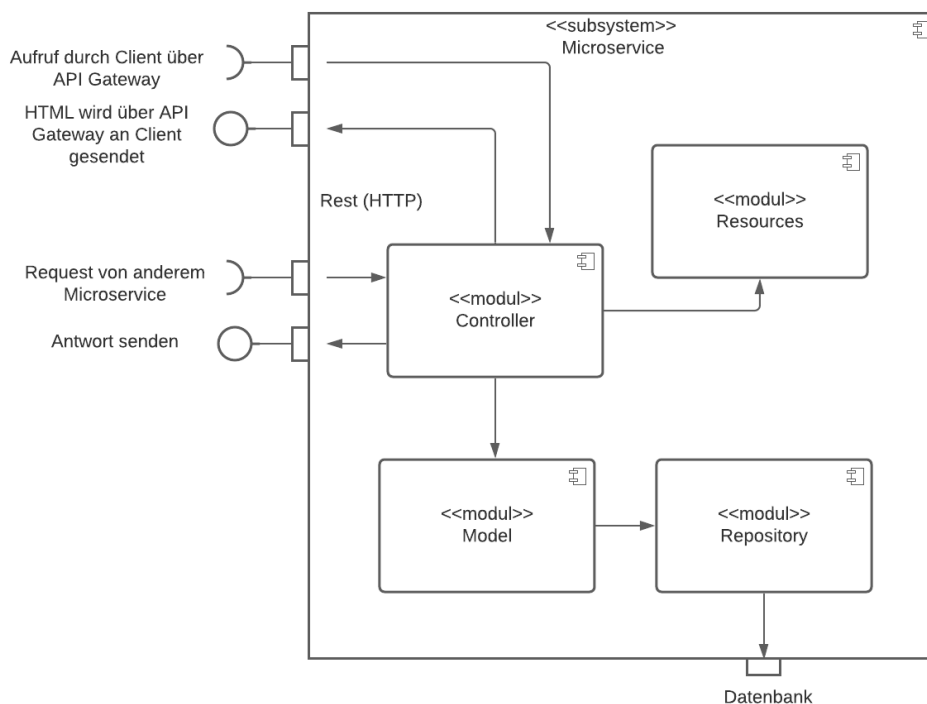


Abbildung 13 Bausteinsicht Ebene 2 <sup>72</sup>

<sup>72</sup> (Eigene Darstellung)

## 4.6 Domainmodell

Das folgende Kapitel soll einen Überblick über die Entitäten und deren Attribute für die jeweiligen Microservices geben.

### 4.6.1 Messestammdatenservice

Für die Messe können Stammdaten wie zum Beispiel Veranstaltungsort- und Zeitpunkt festgelegt werden. Für eine Bildergalerie können Bilder bereitgestellt werden. Zusätzlich können Daten von Ansprechpartnern bereitgestellt werden.

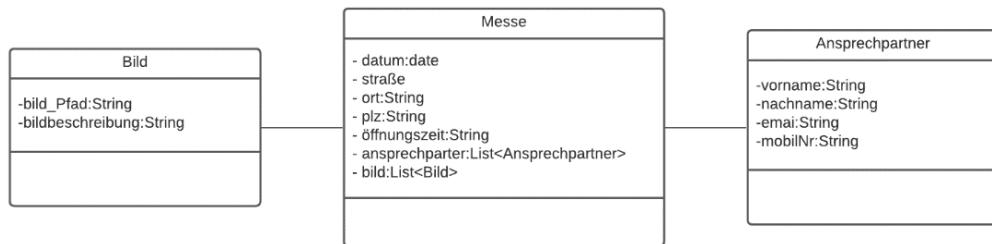


Abbildung 14 Domainmodel Messestammdatenservice <sup>73</sup>

### 4.6.2 Firmenservice

Es können allgemeine Firmendaten wie zum Beispiel Firmenname oder URL zur Webseite der Firma erfasst werden. Es kann festgelegt werden, ob die Firma an der aktuellen Messe teilnimmt. Eine Firma kann über mehrere Standorte und Firmenvertreter verfügen.

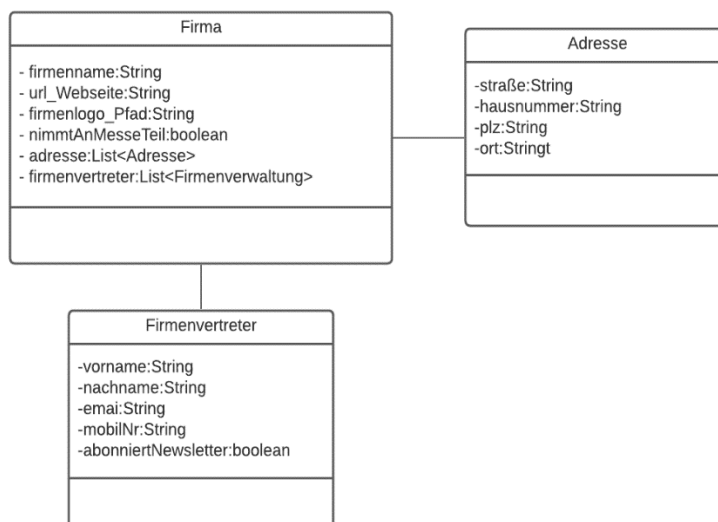


Abbildung 15 Domainmodel Firmenservice <sup>74</sup>

<sup>73</sup> (Eigene Darstellung)

<sup>74</sup> (Eigene Darstellung)

### 4.6.3 Newsletterservice

Ein Newsletter besteht aus allgemeinen Daten wie zum Beispiel Erstellungsdatum. Er kann mehrere Artikel beinhalten. Ein Artikel kann mehrere Bilder enthalten. Ein Newsletter kann per Broadcast an alle erfassten Mailadressen versendet werden. Eine Mailadresse wird im jeweiligen Empfänger gespeichert.

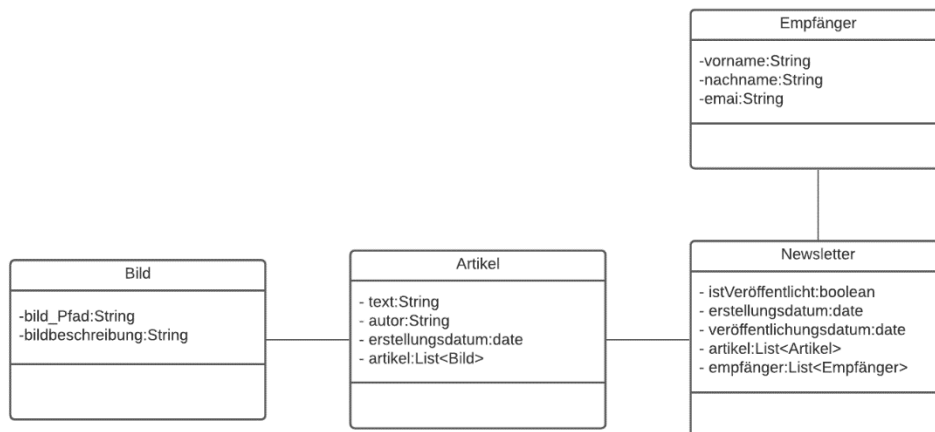


Abbildung 16 Domainmodel Newsletterservice <sup>75</sup>

### 4.6.4 Vortragsservice

Während der Messe werden verschiedene Vorträge gehalten. Für jeden Vortrag wird der Name der vortragenden Person gespeichert. Zusätzlich kann benötigtes Equipment erfasst werden. Für einen Vortrag muss ein Zeitslot und ein Raum gewählt werden. Es dürfen nicht mehrere Vorträge zur gleichen Zeit am gleichen Ort stattfinden.

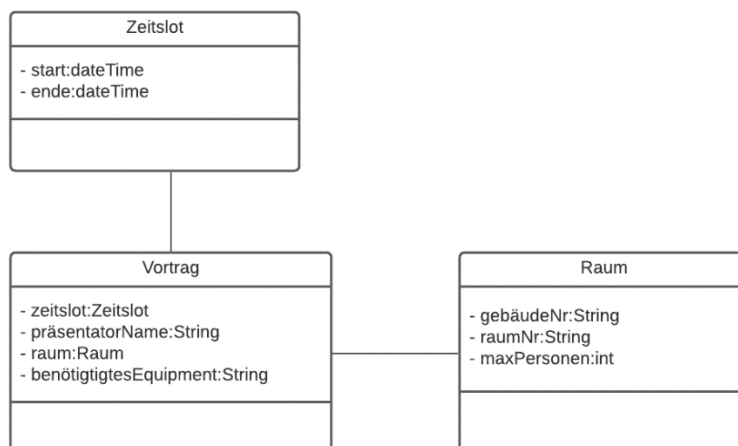


Abbildung 17 Domainmodel Vortragsservice <sup>76</sup>

<sup>75</sup> (Eigene Darstellung)

<sup>76</sup> (Eigene Darstellung)



## 5 Implementierung

### 5.1 Spring Framework

Beim Springframework handelt es sich um ein Open Source Java Framework, welches über Aspektorientierte Programmierung und Dependency Injection einen gut wartbaren und leichten Programmcode ermöglichen soll. Zusätzlich soll damit die Komplexität der Java-Plattform reduziert werden. Am häufigsten kommt das Framework zur Programmierung von Webanwendungen zum Einsatz.<sup>77</sup>

Aspektorientierte Programmierung ermöglicht eine modulare Gestaltung des Codes. Wichtige Funktionen wie zum Beispiel Logging Fehlerbehandlung und Caching werden dabei zentral verwaltet.<sup>78</sup>

Bei der Dependency Injection liefert die Methode einer externen Instanz die Abhängigkeit für ein Objekt. Die Abhängigkeit kann zum Beispiel als Constructor-Parameter der entsprechenden Klasse übergeben werden. Die Übergabe wird als Injection bezeichnet. Dependency Injection soll dazu beitragen, die Abhängigkeiten von Klassen auf ein Minimum zu reduzieren.<sup>79</sup>

#### Spring Boot

Die Microservices der Anwendung werden über das Framework Springboot realisiert, welches auf dem Spring Framework aufsetzt. Es bietet anhand von Autokonfigurations-Mechanismen sehr einfach zu entwickelnde Spring Anwendungen. Unter Spring-Cloud bietet es verschiedene Erweiterungsprojekte, welche zur Realisierung von Microservices genutzt werden können. Weil es Open Source ist, eignet es sich zur Erstellung eines Prototyps. Bei der Entwicklung entstehen aufgrund des Frameworks keine Kosten. Als Programmiersprache für die einzelnen Services wurde Java gewählt. Spring wurde seit der Veröffentlichung im Jahr 2003 (beziehungsweise Spring Boot im Jahr 2012) ständig weiterentwickelt. Es handelt sich dabei um ein ausgereiftes weit verbreitetes Java-Framework. Ein Ende des Spring-Projekts in naher Zukunft ist dahingehend relativ unwahrscheinlich. Das Framework kann deshalb als zukunftssicher angesehen werden und eignet sich für eine Anwendung, welche über Jahre hinweg weiterentwickelt wird. Daher werden die einzelnen Microservices der IT-KoM Verwaltung jeweils als Springboot Projekt verwirklicht.

Dazu wurde der Spring initializr verwendet. Ein Springboot-Projekt lässt sich relativ einfach über den spring initializr auf der Webseite <https://start.spring.io/> erstellen und im Anschluss herunterladen. Die Seite bietet eine Oberfläche mit Eingabe- / Auswahlmöglichkeiten von Buildtool, Programmiersprache, Springbootversion, Paketierung, Java-Version und Abhängigkeiten.<sup>80</sup>

---

<sup>77</sup> (Augsten, 2019)

<sup>78</sup> (Biswanger, 2016)

<sup>79</sup> (Augsten, 2019)

<sup>80</sup> (Rauch, et al., 2021)

Abbildung 18 zeigt die Oberfläche von Spring-initializr.

Abbildung 18 spring initializr <sup>81</sup>

## 5.2 Build-Automatisierung mit Maven

Buildtools automatisieren die Erstellung von Anwendungen aus Quellcode. Dazu gehört Kompilierung, Verlinkung und Verpackung des Codes. Dieser wird dadurch in eine ausführbare Form gebracht. Für die IT-KoM Verwaltung wurde Maven als Buildtool gewählt. Ein Mavenprojekt wird über XML in einer Datei mit dem Namen pom.xml definiert. Unter anderem werden Abhängigkeiten externer Bibliotheken, Name des Projekts und Spring-Version in dieser Datei angegeben. Die Datei wird von Maven für die Umsetzung des Build-Prozesses vorausgesetzt. Beim Buildprozess werden die einzelnen Quelldateien eines Programms in ein lauffähiges Konstrukt konvertiert. <sup>82</sup>

Neben Maven bietet sich für Spring Projekte das Buildtool Gradle an. Dieses verfügt über einen noch größeren Funktionsumfang und ist leistungsfähiger. Für die einzelnen Springbootprojekte wurde eine Konfiguration über XML (welche von Gradle nicht unterstützt wird) bevorzugt. Daher fiel die Entscheidung auf Maven. <sup>83</sup>

Die Abhängigkeiten von externen Bibliotheken werden jeweils pro Microservice über ein zugehöriges Mavenprojekt verwaltet. Jede Springboot-Version verfügt über eine Liste von Abhängigkeiten, welche speziell für die Version getestet wurden. Dadurch wird die Kompatibilität gewährleistet. Die Versionen der Abhängigkeiten müssen nicht angegeben werden. Die Verwaltung der Versionen wird von Maven automatisch umgesetzt. <sup>84</sup>

Abbildung 19 zeigt die Datei pom.xml von dem Microservice DiscoveryService.

<sup>81</sup> (Eigene Darstellung)

<sup>82</sup> (Augsten, 2018)

<sup>83</sup> (Stringfellow, 2017)

<sup>84</sup> (baeldung, 2021)

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.3</version>
    <relativePath/> <!--lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>DiscoveryService</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>serviceDiscover</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>2020.0.3</spring-cloud.version>
  </properties>
  <dependencies>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

Abbildung 19 Discovery Service pom.xml<sup>85</sup>

### 5.3 Frontend mit Thymeleaf

Das Frontend wird in den einzelnen Microservices mit der serverseitigen Java Template-Engine Thymeleaf realisiert. Dieses lässt sich leicht in eine Spring-Anwendung integrieren und ermöglicht die schnelle Entwicklung eines Frontends für eine relativ kleine Anwendung wie einen Microservice.

<sup>85</sup> (Eigene Darstellung)

Zur Integration von Thymeleaf in die einzelnen Microservices wurde jeweils in der Datei `pom.xml` der Services die Abhängigkeit `spring-boot-starter-thymeleaf` hinzugefügt. Thymeleaf beinhaltet eine Komponente, welche View-Namen auf Thymeleaf Templates mappt, welche von Spring Boot im Projektordner unter dem Pfad `src/main/resources/templates` gesucht werden. Ein solches Template wurde zum Beispiel im Besucherservice unter dem genannten Pfad mit dem Namen `start.html` angelegt. Die Datei stellt die Startseite für die öffentliche Besucherwebseite bereit. Abbildung 20 zeigt die Datei `start.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Firmenverwaltung</title>
</head>
<body>

<h1>
  Herzlich Willkommen zur Webseite der IT-Kontaktmesse der Fachhochschule Erfurt
</h1>
```

Abbildung 20 Besucherservice `start.html` <sup>86</sup>

Bei einem Aufruf der Seite `start.html` wird die angeforderte HTML-Datei von einer Controller-Methode des Servers zurückgegeben. Der entsprechende Controller wird dazu mit der Annotation `@Controller` versehen. Die Methoden des Controllers können vom Client über HTTP aufgerufen werden. Diese Methoden erhalten eine Annotation (wie zum Beispiel `@GetMapping("/{Pfad}")`), mit der jeweils die HTTP-Methode und die Aufrufadresse definiert wird. Abbildung 21 zeigt die Implementierung eines Controllers des Besucherservices, welcher über die Methode `home`, die Startseite der Besucherwebseite an den Client sendet.

```
@Controller
public class questController {

    @GetMapping("/") //Pfad zur Startseite (über die HTTP-Methode GET)
    public String home() {

        return „start“; //Name des Templates
    }
}
```

Abbildung 21 Besucherservice `questController.java` <sup>87</sup>

---

<sup>86</sup> (Eigene Darstellung)

<sup>87</sup> (Eigene Darstellung)

Zum dynamischen Hinzufügen von HTML-Elementen stellt Thymeleaf eine Vielzahl von Anweisungen zur Verfügung. Diese sind zum Beispiel `th:if` welche HTML-Elemente anzeigt, wenn eine Bedingung erfüllt ist oder `th:each` welche es ermöglicht die Elemente eines Daten-Containers wie zum Beispiel einer `ArrayList` dynamisch im HTML-Dokument einzufügen. Die Daten werden über ein Model im HTML-Dokument eingebunden. Dieses Model wird von Spring an die View weitergereicht. Der Server ersetzt Thymeleaf-Anweisungen in eine vom Browser lesbare Form bevor er das HTML-Dokument an diesen übermittelt. Auf Abbildung 22 wird der Einsatz der Thymeleaf-Anweisungen anhand eines Beispiels verdeutlicht.

```
//Ist die Liste „companies“ leer dann wird die Bedingung ausgeführt
<table class="table" th:if="${companies.empty}">
  <td colspan="2"> Service ist nicht verfügbar </td>
</table>

// Die Liste companies wird in einer foreach-Schleife durchlaufen und ausgegeben
<table class="table" th:each="company : ${companies}">
  <tbody>
    <tr >
      <td >Firmenname:</td><td th:text="${company.companyName}" ></td>
    </tr>
    <tr>
      <td >Webseite:</td><td th:text="${company.url}"></td>
    </tr>
  </tbody>
</table>
```

Abbildung 22 Thymeleaf Anweisungen, *Besucherservice firmen.html* <sup>88</sup>

Die Abbildung zeigt ein Template zur Ausgabe der Firmendaten aller teilnehmenden Firmen auf der Besucherseite. Dabei werden die Elemente aus dem Model `companies` welches im Controller des Besucherservices dem Model hinzugefügt wurde in einer `foreach`-Schleife durchlaufen und anhand einer Tabelle wiedergegeben. Die Tabelle wird dynamisch erzeugt. Ausgegeben werden Firmenname und Name der Webseite (URL) der einzelnen Firmen. Falls der Microservice Firmenservice (welcher die Firmendaten bereitstellt) nicht verfügbar ist, enthält `companies` keine Elemente. Aufgrund der Angabe `th:if="${companies.empty}"` wird in diesem Fall „Service ist nicht verfügbar ausgegeben“.

89

Abbildung 23 zeigt die Ausgabe im Browser mit 3 teilnehmenden Firmen.

---

<sup>88</sup> (Eigene Darstellung)

<sup>89</sup> (Koller, et al., 2020)



Abbildung 23 Ausgabe im Browser, Besucherservice firmen.html<sup>90</sup>

### Navigation

Die Navigation erfolgt über ein Navigationsmenü, welches von allen Microservices implementiert wird, die diese Funktionalität bereitstellen. Das Navigationsmenü wird nach Möglichkeit einheitlich gehalten. Das Navigationsmenü für die Besucherwebseite wird auf Abbildung 23 dargestellt. Auf dem abgebildeten Menü kann über die Menüpunkte welche mit entsprechenden Links versehen wurden, zwischen den einzelnen Microservices gewechselt werden.

Im Beispiel werden die Firmendaten zu demonstrationszwecken der synchronen Kommunikation vom Frontend des Besucherservices ausgegeben. Es wäre möglich, die Daten direkt über den Firmenservice auszugeben. Ein Klick auf den Link „Messestammdaten“ ruft das Frontend des Messestammdatenservices auf. Das Frontend würde in diesem Fall die Daten des entsprechenden Services ausgeben. Ein Klick auf Login ruft eine Loginmaske auf (deren Implementierung in Kapitel 5.6 behandelt wird). Je nach eingegebenen Accountnamen (und der zugehörigen Rolle) würde der Benutzer entweder zu einem Menü zum Verwalten des Messeauftritts einer entsprechenden Firma oder zum Administrationsmenü für FH-Mitarbeiter weitergeleitet werden. In diesen Verwaltungsbereichen können die Frontends der einzelnen Microservices ebenfalls über ein jeweils einheitlich gehaltenes Menü aufgerufen werden.

<sup>90</sup> (Eigene Darstellung)

## 5.4 Spring Cloud API Gateway

Das API Gateway wurde mit dem Spring Cloud API Gateway umgesetzt. Dieses läuft unter dem asynchronen event-driven Framework Netty. Laut Michael Wellner bietet es folgende Features:

- Discovery Client mit Eureka
- Load Balancer mit Ribbon
- Sicherheitskonfigurationsmöglichkeiten mit Spring Security
- Robustheit mit Hystrix
- Limitsetzung für eine bestimmte Anzahl an Requests pro Zeiteinheit (zum Beispiel mit Redis)
- Pfadänderungen
- zeitabhängiges Routing
- Einbindung eigener Filter

91

Das Gateway wurde gewählt, weil es sich sehr einfach in eine Spring-Anwendung integrieren lässt und weil es problemlos mit Eureka Discovery Service und Feign interagiert. Die Verwendung von Eureka Discovery Service und Feign in der Anwendung wird in den folgenden Kapiteln beschrieben.<sup>92</sup>

Für die IT-KoM Verwaltung wurde das Gateway als Spring Boot Projekt ApiGateway umgesetzt. Diesem wurde die Abhängigkeit spring-cloud-starter-gateway in der Datei pom.xml hinzugefügt. In der Datei application.properties werden zentral die Eigenschaften der jeweiligen Anwendung gespeichert. In dieser Datei wurde der Port 8081 festgelegt, über den Anfragen des Clients zum jeweiligen benötigten Service weitergeleitet werden. Zum Beispiel leitet ein Aufruf der URL localhost://8081/firmenservice/ einen Request zum Firmenservice. Somit muss dem Aufrufer nur noch die Adresse des API Gateways und der Name des aufzurufenden Services bekannt sein. Die Adressauflösung über den Namen erfolgt über die Service Discovery. Die Implementierung der Service Discovery über Eureka wird im Anschluss an dieses Kapitel erläutert.

Mit dem integrierten Load Balancer Ribbon ist es möglich für Requests eine Lastverteilung auf mehrere Serverinstanzen umzusetzen. Die Verwendung des Load Balancers wird ermöglicht, indem in der Datei application.properties des API Gateways folgende Notation angegeben wird:

`spring.cloud.gateway.routes[index der Route].uri=lb//servicename`

Ein Microservice kann mehrfach instanziiert werden. Der Load Balancer verteilt Aufrufe unter den Instanzen des entsprechenden Services, um einen Lastenausgleich im System umzusetzen.<sup>93</sup>

Abbildung 24 zeigt die Konfiguration des Gateways in der Datei application.properties

---

<sup>91</sup> (Wellner, 2019)

<sup>92</sup> (Bayer, 2019)

<sup>93</sup> (laverma, 2020)

```

server.port=8081
spring.application.name=ApiGateway

spring.cloud.gateway.routes[2].id=lbFirmenservice    //id der Route

spring.cloud.gateway.routes[2].uri=lb://firmenservice /* Name des aufzurufenden -
                                                    Services
                                                    (für Loadbalancing) */

spring.cloud.gateway.routes[2].predicates[0]=Path=/** /* Pfad unter dem
                                                    Loadbalancing
                                                    Erfolgen soll */

```

Abbildung 24 API Gateway application.properties

## 5.5 Eureka Discovery Service

Die Service Discovery wird von dem Netflix Tool Eureka umgesetzt. Es handelt sich um eine Clientseitige Service Discovery, welche sich relativ einfach über Spring implementieren lässt und daher gut für einen Prototypen geeignet ist. Eureka ist unter Spring Projekten weit verbreitet und gut dokumentiert. Im Gegensatz dazu steht zum Beispiel eine serverseitige Erkennung über NGINX, welche eine bessere Performance ermöglicht, aber einen entsprechend hohen Konfigurationsaufwand erfordert.

Zur Einrichtung des Eureka Servers wurde das Spring Boot Projekt ServiceDiscovery erstellt. In der Datei pom.xml wurde die Abhängigkeit spring-cloud-starter-netflix-eureka-server hinzugefügt.<sup>94</sup>

Die Main-Klasse wurde mit der Annotation @EnableEurekaServer versehen. Dadurch dient die Springboot-Anwendung als Eureka-Server. Für die Registrierung der Services mit der Serviceregistry muss jeder Microservice als Eureka-Client fungieren. Deshalb wurde jede Main-Klasse der entsprechenden Services mit der Annotation @EnableEurekaClient versehen. Zusätzlich wurde die Abhängigkeit spring-cloud-starter-netflix-eureka-client jeweils in der Datei pom.xml der einzelnen Microservices hinzugefügt.

Für den Eureka Server wurden Server Port und Applikationsname in der Datei application.properties wie in Abbildung 25 definiert. Zusätzlich wurde dabei festgelegt, dass sich der Server nicht mit sich selbst registrieren kann.

```

server.port=8010

spring.application.name=discoveryservice

eureka.client.register-with-eureka=false // verhindert die Registrierung des -
eureka.client.fetch-registry=false      // Servers mit sich selbst

eureka.client.serviceUrl.defaultZone = http://localhost:8010/eureka /*URL Eureka-
                                                    Server */

```

Abbildung 25 DiscoveryService (Eureka Server) application.properties<sup>95</sup>

<sup>94</sup> (Richardson, 2015)

<sup>95</sup> (Eigene Darstellung)



Für die einzelnen Microservices (ausgenommen dem API Gateway) wird eine automatische Erstellung des Ports festgelegt.

Jeder Microservice erhält einen Applikationsnamen. Eureka ermöglicht eine Adressauflösung über diesen Namen. Zusätzlich erhält jede Instanz eines Microservices eine automatisch erzeugte Id. Mit dieser Id kann zwischen den einzelnen Instanzen eines Microservice unterschieden werden. Abbildung 26 zeigt die Konfiguration von Port, Name, und Instanz-Id eines Microservice und Adresse des Eureka Servers.

```
Server.port = ${PORT:0} //Port wird automatisch festgelegt
spring.application.name = firmenservice
eureka.instance.instance-id=${spring.application.name}:${random.uuid} //Instanz-Id
eureka.client.serviceUrl.defaultZone = http://localhost:8010/eureka
```

Abbildung 26 Microservices (Eureka Clients) application.properties <sup>96</sup>

Über den festgelegten Port des Eureka-Servers (unter der URL <http://localhost:8010/>) erhält man Zugriff zum Eureka Dashboard im Browser. Man erhält von dort aus unter anderem Informationen über alle registrierten Eureka-Clients. Abbildung 27 zeigt das Eureka Dashboard mit den Registrierten Microservices API Gateway, Besucherservice, Firmenservice und Vortragservice.

System Status			
Environment	N/A	Current time	2021-10-1
Data center	N/A	Uptime	00:03
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	3
DS Replicas			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
APIGATEWAY	n/a (1)	(1)	UP (1) - localhost:ApiGateway:8081
BESUCHERSERVICE	n/a (1)	(1)	UP (1) - Besucherservice:f92df4b8-9f4b-4db6-9838-1a127e80e00c
FIRMENSERVICE	n/a (2)	(2)	UP (2) - Firmenservice:d61b55f3-e84a-4c95-9830-70511445b956 , Firmenservice:1f9c59d8-13
VORTRAGSERVICE	n/a (1)	(1)	UP (1) - Vortragsservice:dc4735dfe7856aaa8515408dcb370cdc

Abbildung 27 Eureka Dashboard <sup>97</sup>

Es können jederzeit Eureka-Clients zugeschaltet oder abgeschaltet werden. Es könnten zum Beispiel noch die Microservices Newsletterservice und Administrationsservice registriert werden, ohne dabei die anderen Services zu beeinflussen. Das API Gateway wird mit dem Port 8081 auf dem Eureka Dashboard angezeigt. Die restlichen Microservices enthalten einen automatisch erzeugten Port und eine automatisch generierte Id. Im Beispiel sind zwei Instanzen des Microservice Firmenservice mit verschiedenen Id's

<sup>96</sup> (Eigene Darstellung)

<sup>97</sup> (Eigene Darstellung)

registriert. Der Load Balancer des API Gateways verteilt in diesem Fall bei Aufrufen des Firmenservice die Last über die beiden Instanzen. Ein Beispiel zum Testen des Load Balancers in der Anwendung wird im Kapitel 5.10 Unter Verwendung von Distributed Tracing beschrieben.<sup>98</sup>

### 5.6 Keycloak und Spring Security

Keycloak ist ein Opensource Projekt für Identity und Access Management von der Firma RedHat. Ziel des Projektes ist es eine sichere Authentifizierung und Autorisierung im System, mit möglichst wenig Code zur Verfügung zu stellen. Keycloak stellt einen Autorisierungsserver bereit, welcher das OpenId Connect Protokoll verwendet. Es bietet die Möglichkeit auf Accounts von Drittanbietern wie zum Beispiel Facebook oder Youtube zu referenzieren. Dadurch kann dem Benutzer unter anderem das Anlegen von Benutzeraccounts für die Anwendung erspart bleiben. Darüber hinaus stellt es eine Loginmaske bereit, welche zum Beispiel Funktionen wie remember me oder Verlinkungen zu einem Registrierungsformular bietet.

Für die IT-KoM Verwaltung könnte mit dem Einsatz von Keycloak ebenfalls ein Login über Drittanbieter wie zum Beispiel unter der Verwendung des Hochschulaccounts, über den Server der Hochschule ermöglicht werden. Die Entwicklungszeit für Login- und Registrierungsmaske entfällt durch den Einsatz von Keycloak. Die Formulare können über die Keycloak-Administrationsoberfläche bereitgestellt werden. Zusätzlich entfällt die Entwicklungszeit einer Accountverwaltung für FH-Mitarbeiter. Diese Funktionalität wird ebenfalls von Keycloak bereitgestellt.

Damit angemeldete Benutzer nur Ressourcen verwenden können, für deren Verwendung sie autorisiert sind, wurde der Einsatz von Rollen wie zum Beispiel Firma und FH-Mitarbeiter festgelegt. Diese lassen sich ebenfalls mit wenig Aufwand über die Benutzeroberfläche konfigurieren. Die Implementierung der Rollenverwaltung in den einzelnen Microservices konnte jedoch unter dem Rahmen dieser Wissenschaftlichen Arbeit nicht umgesetzt werden.

Neben Keycloak gibt es zum Beispiel noch Lösungen wie Okta oder Auth0 welche über einen höheren Funktionsumfang und besseren Support verfügen aber kostenpflichtig sind. Keycloak wurde für die Anwendung gewählt, weil es sich dabei um ein beliebtes Open-Source-Projekt handelt.

Abbildung 28 zeigt die Konfigurationsmöglichkeiten der Anmeldeformulare.

---

<sup>98</sup> (spring, 2015)

it-kom

General Login Keys Email Themes I

User registration ☒ ON

Email as username ☐ OFF

Edit username ☐ OFF

Forgot password ☐ OFF

Remember Me ☐ OFF

Verify email ☐ OFF

Login with email ☒ ON

Require SSL external requests ▼

Save Cancel

Abbildung 28 Keycloak Konfiguration Anmeldeformular <sup>99</sup>

Auf der offiziellen Webseite von Keycloak steht eine Standalon-Server-Distribution frei zum Download zur Verfügung. Diese lässt sich per Konsole über den Befehl `bin\standalone.bat` (für Windows) und über `bin\standalone.sh` (für Linux) starten.

Nach dem Start des Keycloak Servers lässt sich im Browser über die URL `http://localhost:8080/auth/admin` die Administrationsoberfläche aufrufen. Für die IT-KoM Verwaltung wurde über diese Oberfläche der Realm `it-kom` eingerichtet. Unter dem Realm lassen sich Benutzer, Anmeldeinformationen, Rollen und Gruppen anlegen und verwalten. Zusätzlich lassen sich Clients Managen. Bei Clients handelt es sich laut `keycloak.org` um Entitäten, welche von Keycloak angefordert werden um den Benutzer zu authentifizieren. Dabei handelt es sich um Dienste oder Anwendungen. Für die IT-KoM Verwaltung wurde der Client `it-kom-client` eingerichtet. Die Abbildung 29 zeigt die Administrationsoberfläche von Keycloak. <sup>100</sup>

---

<sup>99</sup> (Eigene Darstellung)

<sup>100</sup> (keycloak, 2021)

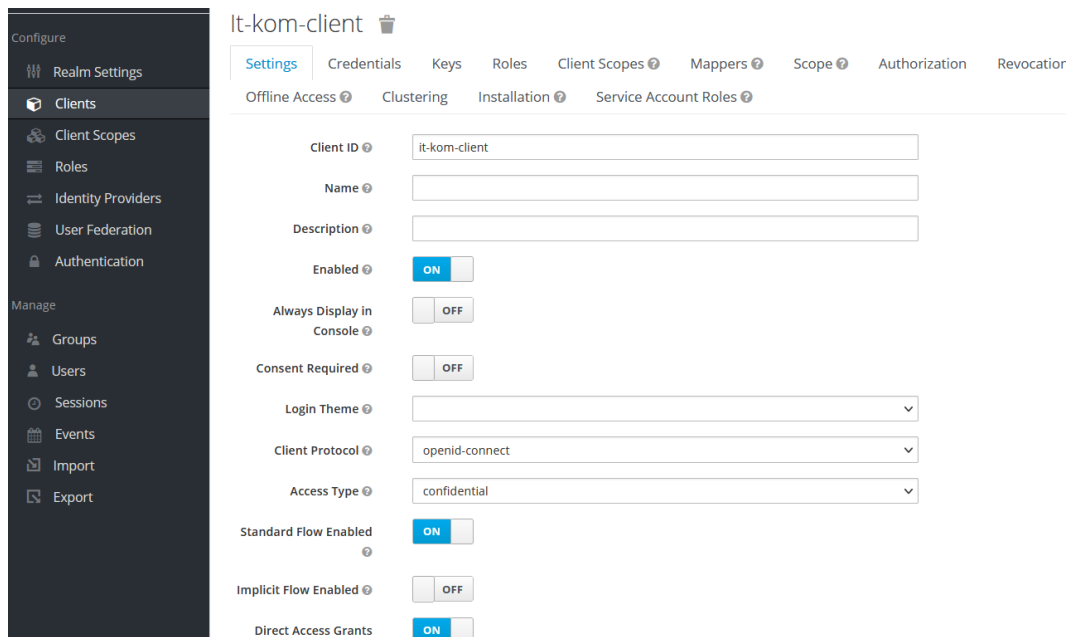


Abbildung 29 Keycloak Administrationsoberfläche <sup>101</sup>

Damit eine Authentifizierung über Keycloak unter Verwendung von OAuth2 und OpenID Connect in der IT-Kom Verwaltung umgesetzt werden kann, wurden in der Datei `application.properties` des API Gateways die Befehle wie auf Abbildung 30 hinzugefügt.

<sup>101</sup> (Eigene Darstellung)

```
Spring.security.oauth2.client.provider.keycloak.issuer-  
uri=http://localhost:8080/auth/realms/it-kom //realm  
  
spring.security.oauth2.client.registration.spring-cloud-gateway-client.client-  
id=it-kom-client //client  
  
spring.security.oauth2.client.registration.spring-cloud-gateway-client.client-  
secret=c6f5012f-4151-4b59-b63c-164f5a51e428 //secret  
  
spring.security.oauth2.client.registration.spring-cloud-gateway-  
client.provider=keycloak //provider  
  
spring.security.oauth2.client.registration.spring-cloud-gateway-  
client.authorization-grant-type=authorization_code //grant-type  
  
spring.security.oauth2.client.registration.spring-cloud-gateway-client.redirect-  
uri=http://localhost:8081/login/oauth2/code/it-kom-client //URL zur Loginmaske  
  
spring.security.oauth2.resourceserver.jwt.jwk-set-  
uri=http://localhost:8080/auth/realms/it-kom/protocol/openid-connect/certs  
// URL zum Austausch eines Verifizierungsschlüssels
```

Abbildung 30 Keycloak-Konfiguration, ApiGateway application.properties <sup>102</sup>

Mit diesen Befehlen wird unter anderem Keycloak-realm und Keycloak-client festgelegt. Weiterhin wurde die Adresse des Ressourcenservers und der Loginmaske festgelegt. Zusätzlich wurden in der Datei pom.xml die Abhängigkeiten spring-boot-starter-oauth2-client und spring-boot-starter-oauth2-resource-server hinzugefügt.

In den restlichen Microservices (ausgenommen DiscoveryService) wurden in der Datei pom.xml die Abhängigkeiten spring-boot-starter-oauth2-resource-server und spring-boot-starter-oauth2-jose hinterlegt. Zusätzlich wurde jeweils folgender Befehl zur Festlegung des Ressourcenservers in der Datei application.properties wie auf Abbildung 31 eingetragen:

```
spring.security.oauth2.resourceserver.jwt.issuer-  
uri=http://localhost:8080/auth/realms/it-kom
```

Abbildung 31 URL des Autorisierungsservers in den einzelnen Microservices, application.properties <sup>103</sup>

## Spring Security

Spring Security ist ein Authentifizierungs- und Zugriffskontroll-Framework für Java Anwendungen. Laut spring.io bietet es unter anderem folgende Features:

- Erweiterbare Unterstützung für Autorisierung und Authentifizierung
- Schutz vor Angriffen wie zum Beispiel Clickjacking, Cross-Side-Request-Forgery und fixation
- Integration mit Spring Web MVC

Zur Integration von Spring Security wurde in der Datei pom.xml der Services und im API Gateway die Abhängigkeit spring-boot-starter-security hinzugefügt. Weiterhin wurde jeweils die Klasse SecurityConfig erstellt und mit @EnableWebSecurity gekennzeichnet. Dadurch werden bei jedem Request die Sicherheitskonfigurationen umgesetzt. Zusätzlich wurden Filterketten wie auf Abbildung 32 definiert.

---

<sup>102</sup> (Eigene Darstellung)

<sup>103</sup> (Eigene Darstellung)

```
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http.authorizeExchange()
        //keine Authentifizierung erforderlich
        .pathMatchers("/besucherservice/**").permitAll()
        .pathMatchers("/firmenservice/allCompanies").permitAll()

        //Authentifizierung erforderlich
        .pathMatchers("/newsletter-service/").authenticated()
        .pathMatchers("/firmenservice/create").authenticated()

        .and()
        .oauth2Login(withDefaults()) //Autorisierung nur über OAuth2 zugelassen
        .oauth2Client();
    http.csrf().disable(); // verhindert csrf-Attacken
    return http.build();
}
```

Abbildung 32 Spring Security Filterkette, ApiGateway SecurityConfig.java <sup>104</sup>

Es wurde festgelegt, für welche URL ein Login erforderlich ist. Weiterhin können Filter anhand von Rollen gesetzt werden. Die Methoden `pathMatchers` und `permitAll()` ermöglichen einen Aufruf der gewählten URL, ohne Authentifizierung. Die Methode `authenticated` setzt eine Authentifizierung des Benutzers voraus.

In der Anwendung sind alle URL's des Besucherservice (unter `/besucherservice/..`) für Benutzer ohne Authentifizierung verfügbar.

Weiterhin kann der Besucherservice, Firmendaten vom Firmenverwaltungsservice holen (über die URL `/firmenverwaltung/allCompanies`) und ausgeben.

Die URL's `/newsletter-service` und `/firmenservice/create` sind nur von authentifizierten Benutzern aufrufbar.

Ist für einen Benutzer bei einem Aufruf eine Authentifizierung erforderlich, dann wird er dazu zur Keycloak-Loginmaske weitergeleitet deren URL in der Datei `application.properties` im API Gateway angegeben wurde. Nach einem erfolgreichen Login erfolgt eine Automatische Weiterleitung zur Ziel-URL.

Abbildung 33 zeigt die Loginmaske der Anwendung.

---

<sup>104</sup> (Eigene Darstellung)

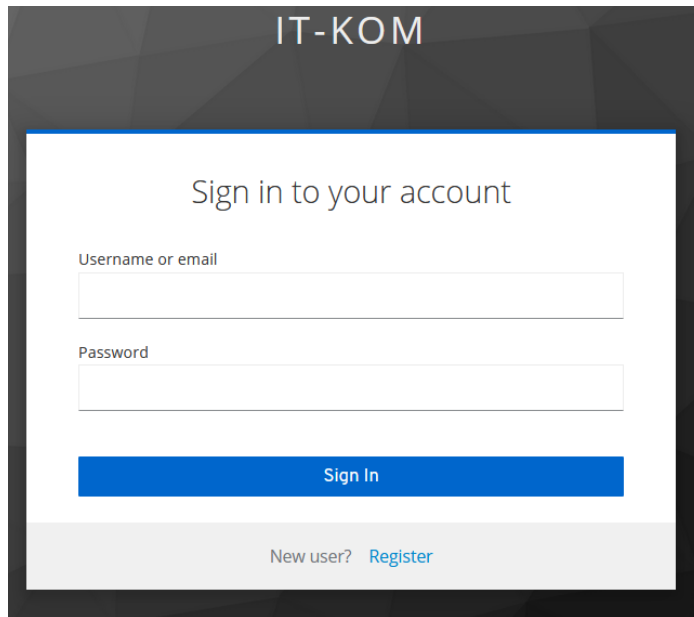


Abbildung 33 Loginmaske IT-KoM Verwaltung <sup>105</sup>

## 5.7 Synchrone Kommunikation mit Feign Client

Eine Möglichkeit für den Datenaustausch zwischen den Microservices bietet das Spring RestTemplate. Es bietet einen Client zum synchronen Datenaustausch per HTTP. Es stellt dabei Vorlagen für eine Vielzahl möglicher HTTP Anfrageszenarien zur Verfügung. Darüber hinaus bietet es die Möglichkeit clientseitiges Loadbalancing einzusetzen. Die Verwendung von RestTemplate ist allerdings umständlich, weil dabei relativ viel Code benötigt wird, welcher nicht rein intuitiv zu verstehen ist. Eine einfachere Lösung bietet Feign. Es handelt sich dabei um einen HTTP-Client welcher von Netflix zur Vereinfachung von HTTP-Clients entwickelt wurde. Der Einsatz von Feign bringt folgende Features mit sich:

- Frei konfigurierbarer Decoder / Encoder zum Beispiel für XML oder JSON
- Logger
- Errorhandler
- Load Balancer (mittels Ribbon und Eureka)

Feign-Clients bringen jedoch den Nachteil mit sich, dass diese nicht mit Binärdateien wie zum Beispiel Datei-Download / -Upload umgehen können. Nur der Einsatz von textbasierten Schnittstellen wird unterstützt. <sup>106</sup>

Das folgende Beispiel zeigt den Einsatz von Feign in der IT-KoM Anwendung. Die Ausgabe der Besucherseite wird über den Microservice Besucherservice ermöglicht. Dieser benötigt vom Microservice Firmenverwaltung die für die Messe angemeldeten Firmendaten. Dazu wird über den Controller guestController ein Request per Feign-Client an den Microservice Firmenverwaltung gesendet. Dieser ruft die Methode allCompanies auf, welche alle Firmen aus der Datenbank des Firmenservice holt und als Liste von CompanieData-Objekten zurückgibt.

---

<sup>105</sup> (Eigene Darstellung)

<sup>106</sup> (jambit, 2019)

Zur Verwendung von Feign wird die Abhängigkeit spring-boot-starter-openfeign benötigt, welche in der IT-Kom Anwendung in den entsprechenden Microservices hinzugefügt wurde. Weiterhin wurde in der Main-Klasse die Annotation @EnableFeignClients hinzugefügt.

Für den Zugriff auf die API des Firmenservice wurde das Interface FirmenServiceClient erstellt. Dieses stellt eine Methode zum Aufruf der API bereit. Abbildung 34 zeigt die Deklaration vom Interface.

```
@FeignClient(name="firmenservice") //Name des Aufzurufenden Services
public interface FirmenServiceClient {

    @GetMapping("/allCompanies") // Aufzurufenden Methode
    public List<CompanyData> allCompanies();

}
```

Abbildung 34 Besucherservice FirmenServiceClient <sup>107</sup>

Das Interface wurde mit der Annotation @FeignClient versehen, wodurch dieses von Spring als Komponente erkannt wird. Die Annotation enthält weiterhin den Namen des aufzurufenden Services, wodurch eine Adressauflösung per Discovery Service erfolgt. Der Name muss dem Namen entsprechen, welcher beim aufzurufenden Service in der Datei applications.properties festgelegt wurde. Die Methode allCompanies im Interface ruft die gleichnamige Methode des Firmenservice auf, welche die Firmendaten aus der Datenbank des Services holt und diese Daten als Liste zurückgibt. Der Pfad der aufzurufenden Methode wird per annotation @GetMapping(„Pfad“) bestimmt.

Zum Aufrufen der Methode des Interfaces wurde dieses im guestController deklariert und mit der Annotation @Autowired versehen. Das interface wird dadurch von Spring per Dependency-Injection automatisch verdrahtet, wodurch die Funktion allCompanies des Interfaces, automatisch implementiert wird.

Abbildung 35 zeigt die Implementierung des Interfaces im guestController und den Aufruf der Funktion allCompanies.

```
@Autowired
FirmenServiceClient firmenServiceClient;

...

List<CompanyData> companyDataList = firmenServiceClient.allCompanies();
```

Abbildung 35 Implementierung FirmenServiceClient und Funktionsaufruf, Besucherservice guestController.java <sup>108</sup>

Die Methode allCompanies des Firmenservice wird auf Abbildung 36 dargestellt.

---

<sup>107</sup> (Eigene Darstellung)

<sup>108</sup> (Eigene Darstellung)



```
@GetMapping("/allCompanies")
@ResponseBody
public ResponseEntity <List<CompanyData>> allCompanies() throws
InterruptedException {

    //Holt die Firmendaten aus der Datenbank des Firmenservices
    List<CompanyData> companyDataList = companyDataRepository.findAll();

    //gibt eine ResponseEntity zurück, welche im Body die Firmendaten beinhaltet
    return ResponseEntity.status(HttpStatus.OK).body(companyDataList);
}
```

Abbildung 36 allCompanies Aufruf, Firmenservice Controller.java <sup>109</sup>

## 5.8 Resilience4J Circuit Breaker

Resilience4J ist eine Fehlertoleranzbibliothek, welche von der populären Bibliothek Netflix Hystrix inspiriert wurde, welche sich mittlerweile im Wartungsmodus befindet. Folgende Funktionalitäten werden von Resilience4J zur Verfügung gestellt:

- Circuit-breaking
- Rate-Limiting (Festlegung der Anzahl gleichzeitiger Anfragen, die ein Client stellen darf)
- Retry (Konfigurierbare Anzahl von Wiederholungen bei fehlgeschlagenen Anfragen, bevor ein Fehler geworfen wird)
- Bulkhead (fehlerhafte Elemente werden in Pools isoliert, so dass funktionsfähige Komponenten weiterhin funktionieren)

110

Eine Alternative gegenüber Resilience4J bietet Sentinel. Es bringt ein eigenes Dashboard Modul mit sich und eignet sich für komplexe verteilte Anwendungen. Resilience4J bringt gegenüber Sentinel den Vorteil, dass Module wie zum Beispiel Circuit Breaker oder Rate Limiter einzeln als Abhängigkeit hinzugefügt werden können. Sentinel lässt sich dem gegenüber nur als komplettes Paket hinzufügen, unabhängig davon, ob nur einzelne Funktionalitäten benötigt werden. Bei der Implementierung des Prototypen wurde vorerst nur das Circuit Breaker Pattern verwendet. Daher wurde in der IT-KoM Verwaltung Resilience4J eingesetzt. <sup>111</sup>

Der Circuit Breaker wurde zum Beispiel im Microservice Besucherservice angewendet. Zur Integrierung wurde in der Datei pom.xml die Abhängigkeit spring-cloud-starter-circuitbreaker-resilience4j hinzugefügt. Das Circuit-Breaker Pattern speichert und aggregiert Aufrufe in einem sliding window. Dieses kann Zählbasiert oder Zeitbasiert implementiert werden. Ein Zählbasiertes sliding window setzt den Zustand des Circuit Breaker Pattern auf geöffnet, wenn eine bestimmte Anzahl an Aufrufen fehlschlägt. Wird die Konfiguration auf Zeitbasierend gesetzt, dann wird der Circuit Breaker nach einer bestimmten Anzahl fehlerhafter Aufrufe in einem bestimmten Zeitraum auf geöffnet gesetzt. Für Testzwecke wurde das Circuit Breaker Pattern im Controller guestController im Besucherservice wie auf Abbildung 37 implementiert.

---

<sup>109</sup> (Eigene darstellung)

<sup>110</sup> (Vitz, 2021)

<sup>111</sup> (Bhuyan, 2020)

```
CircuitBreakerConfig config = CircuitBreakerConfig.custom()

    // setzt das sliding window auf Zählbasiert
    .slidingWindowType(CircuitBreakerConfig.SlidingWindowType.COUNT_BASED)

    .slidingWindowSize(3) // Fenstergröße 3

    .slowCallRateThreshold(50.0f) // maximale Fehlerrate 50%

    .slowCallDurationThreshold(Duration.ofSeconds(1)) //maximale Aufrufzeit 1s

    .build();

/* Instanziert den Circuit Breaker mit der entsprechenden Konfiguration
   und dem Namen „test“ */
CircuitBreakerRegistry registry = CircuitBreakerRegistry.of(config);
CircuitBreaker circuitBreaker = registry.circuitBreaker("test");
```

Abbildung 37 Circuit Breaker Konfiguration, Besucherservice guestController.java <sup>112</sup>

Der Typ des sliding window wurde auf Zählbasiert gesetzt. Für die Größe des sliding window wurde der Wert 3 festgelegt. Die Methode SlowCallRateThreshold setzt einen Schwellenwert in Abhängigkeit zur konfigurierten Fenstergröße in Prozent. Überschreitet die Anzahl an Aufrufen mit Zeitüberschreitung den Schwellenwert, dann wird der Zustand des Circuit Breaker Pattern nach dem Befüllen des sliding window auf Offen gesetzt. Im Beispiel wurde dieser Wert auf 50 gesetzt. SlowCallDurationThreshold setzt für jeden Aufruf ein Zeitfenster. Wird dieses Zeitfenster überschritten dann wird dieser Aufruf mit einer Zeitüberschreitung vermerkt. Das Zeitfenster wurde im Beispiel auf maximal eine Sekunde gesetzt. Der Circuit Breaker wurde unter dem Namen *test* instanziiert. <sup>113</sup>

Über einen Feign Client der Request an den Microservice Firmenverwaltung gesendet. Dieser erhält als Antwort eine Liste mit allen teilnehmenden Firmen. Welche den Gästen an entsprechender Stelle im User Interface angezeigt werden. Der Aufruf des Feign Client erfolgt über einen Supplier, welcher wie unter Abbildung 38 mit dem Circuit Breaker dekoriert wurde.

```
Supplier<List<CompanyData>> companyDataSupplier =
    () -> firmenServiceClient.allCompanies();

Supplier<List<CompanyData>> decoratedCompanyDataSupplier =
    circuitBreaker.decorateSupplier(companyDataSupplier);
```

Abbildung 38 Supplier mit Circuit Breaker dekoriert <sup>114</sup>

Ein Test des Circuit Breakers wird auf Abbildung 39 gezeigt.

<sup>112</sup> (Eigene Darstellung)

<sup>113</sup> (resilience4j, 2021)

<sup>114</sup> (Eigene Darstellung)

```
List<CompanyData> companies = new ArrayList<>();

For (int i = 0; i < 10; i++){ // Die Schleife wird 10 mal durchlaufen

    try { //Der Circuit Breaker ist geschlossen

        //Holt die Firmendaten vom Firmenservice
        // Der Aufruf erfolgt über den dekorierten Supplier
        companies = decoratedCompanyDataSupplier.get();

        //Gibt den Firmennamen des ersten Datensatzes aus
        System.out.println(companies.get(0).getCompanyName());

        //Der Circuit Breaker ist offen / halb-offen
    } catch (CallNotPermittedException e) {

        //Ausgabe der Fehlermeldung über die Konsole
        System.out.println(e.getMessage());

    }

}
/*Die Daten werden einem Model übergeben welches von Thymeleaf zur Ausgabe der
Daten verwendet wird. */
model.addAttribute("companies", companies);

return "firmen"; //das Template firmen.html wird gerendert
```

Abbildung 39 Test des Circuit Breakers über eine for Schleife <sup>115</sup>

Dabei wird eine for-Schleife zehnmal durchlaufen. Bei jedem Durchlauf wird im try-Block versucht über die Funktion `decoratedCompanySupplier.get()` die Firmendaten vom Firmenverwaltungsservice zu holen. Der Name der ersten Firma wird zum Prüfen der erfolgreichen Datenübertragung bei jedem Durchlauf in der Konsole ausgegeben. Werden die in der Konfiguration festgelegten Grenzwerte überschritten, dann wird eine `CallNotPermittedException` geworfen und der Code wird im catch-Block weiter ausgeführt. Im Falle eines fehlerfreien Ablaufes werden alle Firmendaten an ein Model übergeben. Am Ende wird „firmen“ zurückgegeben, wodurch die Seite `firmen.html` gerendert wird. Die Firmendaten werden über das Model ausgegeben. Im Fehlerfall wird die Seite `firmen.html` ohne Ausgabe der Firmendaten gerendert. Die Fehlermeldung wird in der Konsole ausgegeben.

Die Anfrage des Services Besucher ruft im Firmenverwaltungsservice die Methode `allCompanies()` auf. Diese gibt eine Liste mit allen verfügbaren Firmendaten zurück. Im Test soll geprüft werden, ob der Circuit Breaker entsprechend der Konfiguration auf den Zustand offen gesetzt wird und eine `CallNotPermittedException` geworfen wird. Dieses Verhalten wird entsprechend der Konfiguration nach drei Aufrufen mit höchstens einer Zeitüberschreitung ausgelöst. Zur Umsetzung wurde in der Methode `allCompanies()` im Firmenservice, die Methode `Thread.sleep(1100)` eingefügt. Dadurch erfolgt nach jedem Aufruf eine Zeitüberschreitung. Abbildung 40 zeigt die Methode `allCompanies` im Firmenservice.

---

<sup>115</sup> (Eigene Darstellung)

```
@GetMapping („/allCompanies“)
@ResponseBody
public ResponseEntity <List<CompanyData>> allCompanies() throws
InterruptedException {

    Thread.sleep(1100);

    List<CompanyData> companyDataList = companyDataRepository.findAll();
    return ResponseEntity.status(HttpStatus.CREATED).body(companyDataList);
}
```

Abbildung 40 Firmenservice Controller allCompanies (mit Thread.sleep) <sup>116</sup>

Die Konsolenausgabe nach der Ausführung des Tests wird auf Abbildung 41 dargestellt.

```
Firma 1
Firma 1
Firma 1
CircuitBreaker 'test' is OPEN and does not permit further calls
CircuitBreaker 'test' is OPEN and does not permit further calls
CircuitBreaker 'test' is OPEN and does not permit further calls
CircuitBreaker 'test' is OPEN and does not permit further calls
CircuitBreaker 'test' is OPEN and does not permit further calls
CircuitBreaker 'test' is OPEN and does not permit further calls
CircuitBreaker 'test' is OPEN and does not permit further calls
```

Abbildung 41 Test des Circuit Breakers Konsolenausgabe <sup>117</sup>

Es wird dreimal der Firmenname des ersten Datensatzes aus der Firmendatenliste ausgegeben. Weil alle drei Aufrufe die Zeit überschreiten, wird nach dem dritten Aufruf eine CallNotPermittedException geworfen und der Circuit Breaker geöffnet. Die Fehlermeldung wird in der Konsole ausgegeben. Über einen bestimmten Zeitraum ist keine Anfrage an den Firmenverwaltungsservice mehr möglich. Der Service erhält dadurch die Möglichkeit, ohne Erzeugung weiterer Fehler in einen fehlerfreien Zustand zurückzukehren. Nach Ablauf einer konfigurierbaren Zeit (welche über den Befehl `.waitDurationInOpenState(Duration.ofSeconds(seconds))` definiert werden kann) geht der Circuit Breaker in den Zustand halb offen und nach weiteren erfolgreichen Aufrufen in den Zustand geschlossen. <sup>118, 119</sup>

## 5.9 Docker Container

Docker ist die Containersoftware der Firma Docker Inc. welche laut Bernd Öggl und Michael Kofler den Container-Markt als solchen geschaffen haben und aufgrund der schnellen Entwicklung in der Branche das Tempo vorgeben. <sup>120</sup> Die Container werden von der Docker Engine verwaltet. <sup>121</sup> Docker läuft auf Linux- (CentOS, Debian, Fedora, Oracle Linux, RHEL, Suse und Ubuntu) und Windows Server – Betriebssystemen. <sup>122</sup> Die Docker Engine lässt sich über die offizielle Webseite <https://docs.docker.com/get-docker/> herunterladen und wird über die Kommandozeile ausgeführt. Die Docker Desktop Version

---

<sup>116</sup> (Eigene Darstellung)

<sup>117</sup> (Eigene Darstellung)

<sup>118</sup> (resilience4j, 2021)

<sup>119</sup> (Nagendra, 2021)

<sup>120</sup> (Öggl, et al., 2019)

<sup>121</sup> (Docker, 2021)

<sup>122</sup> (Docker, 2021)

bringt zusätzlich noch eine Benutzeroberfläche mit sich. Jeder Container wird aus einem Image mit eigenem Dateisystem heraus gestartet, welches eine Blaupause für einen Container darstellt. Zum Image gibt es zusätzlich noch eine Beschreibungsdatei welche Konfigurationsanweisungen bereitstellen. Diese Datei wird Dockerfile genannt. Docker-Images können weltweit über Dockerhub ausgetauscht werden, Dockerhub ist ein Repository-Registrierungsdienst welcher es ermöglicht Docker-Images öffentlich oder privat in der Cloud bereitzustellen.<sup>123</sup>

Die Verwendung von Docker-Containern für die IT-KoM Verwaltung war unter der Entwicklung an einer lokalen Maschine nicht möglich, weil keine IP-Konfiguration für das Netzwerk vorgenommen wurde. Der lokale Rechner auf dem die Anwendung Entwickelt wurde, erhielt die Adresse *localhost*, welche innerhalb eines Docker-Container nicht ohne weiteres aufgerufen werden kann. Beim Prototypen würde zum Beispiel die Adresse `http://localhost:8081/besucher/` zur Öffentlichen Webseite für Besucher führen. Im Livebetrieb würde localhost mit der IP-Adresse der Domain oder dem Domainnamen wie zum Beispiel: `http://123.123.123.123:8081/besucher/` oder `http://www.ai-it-kom/besucher/` ersetzt werden. Diese IP-Adressen können von Docker-Containern aufgerufen werden. Im folgenden Abschnitt wird die Erstellung eines Docker-Containers für den Besucherservice erläutert, welcher zum Beispiel über Hochschulrechenzentrum zum Bereitstellen der IT-KoM Verwaltung verwendet werden könnte.

Als Erstes wurde eine jar-Datei (*Besucher-0.0.1-SNAPSHOT.jar*) über Maven mit dem Befehl `mvn clean package` erstellt. Für die Erstellung des Docker-Images wurde zunächst eine Datei mit dem Namen *dockerfile* im Besucherservice erstellt. Der Inhalt der Datei wird auf Abbildung 42 dargestellt.

```
FROM openjdk:8-jdk-alpine
COPY target/Besucher-0.0.1-SNAPSHOT.jar Besucherservice.jar
ENTRYPOINT ["java","-jar","/Besucherservice.jar"]
```

Abbildung 42 Besucherservice dockerfile <sup>124</sup>

FROM gibt das Basis-Image an. Es handelt sich hierbei um ein Javafähiges alpine-Linux. COPY kopiert die Jar-Datei in das Image. ENTRYPOINT legt die ausführbare Datei zum Starten des Containers fest. Das Image wird mit dem Befehl `docker build --tag=besucherservice:latest` erstellt. Der Container kann mit dem Befehl `docker run -d -p lokaler-Port:Container-Port Besucherservice` gestartet werden. Alternativ kann der Container nach der Erstellung des Images über die Docker-Desktop Benutzeroberfläche gestartet werden. Unter der Verwendung eines Dockerhub-Repositories könnte der Container schnell und einfach in der Cloud zum Beispiel für Testzwecke geteilt werden.

<sup>125</sup>

Abbildung 43 zeigt die Benutzeroberfläche von Docker Desktop

---

<sup>123</sup> (Rieger, et al., 2020)

<sup>124</sup> (Eigene Darstellung)

<sup>125</sup> (spring, 2021)

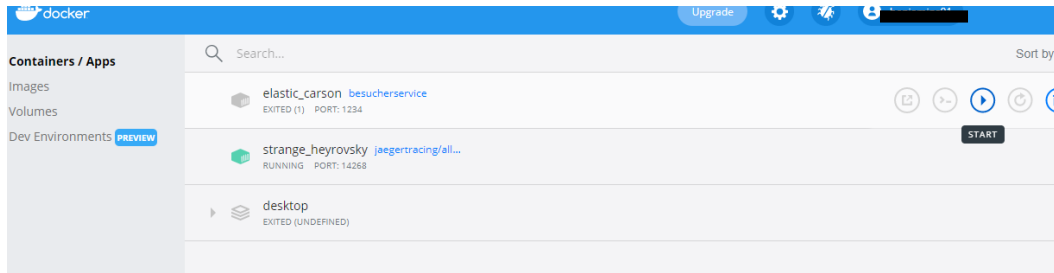


Abbildung 43 docker Desktop Benutzeroberfläche <sup>126</sup>

### 5.10 Distributed Tracing mit Jaeger

Jaeger ist ein open source Distributed Tracing System von der Firma Uber. Es dient zur Fehlerbehebung und Überwachung in einer Microservice-Anwendung. Es visualisiert den gesamten Prozessfluss einer Anfrage durch verschiedene Microservices. Folgende Inhalte werden dabei geboten:

- Transaktionsüberwachung
- Ursachenanalyse
- Analyse von Dienstabhängigkeiten
- Latenz- und Performanceoptimierung

127

Eine Alternative zu Jaeger stellt das Distributed Tracing System Zipkin zur Verfügung. Jaeger und Zipkin verfügen in etwa über den gleichen Funktionsumfang. Zipkin ist älter und ausgereifter, Jaeger ist dagegen schneller und flexibler einsetzbar. Weil Jaeger einen neueren und moderneren Architekturansatz darstellt, wurde es als Distributed Tracing System für die IT-KoM Verwaltung gewählt. <sup>128</sup>

Jaeger wurde in der IT-KoM Verwaltung implementiert, um die Wartbarkeit zu optimieren. Es ermöglicht die Auswertung bestimmter Anwendungsfälle. Es kann zum Beispiel geprüft werden, ob der verwendete Circuit Breaker von Resilience4J oder der Load Balancer Ribbon richtig funktionieren. Zur Implementierung von Jaeger wurden im API Gateway und in den jeweiligen Microservices in der Datei pom.xml die Abhängigkeit opentracing-spring-jaeger-cloud-starter hinzugefügt. Im API Gateway wurde zusätzlich noch die Abhängigkeit opentracing-spring-gateway-cloud-starter hinzugefügt, damit die Spans eines Traces weitergeleitet werden. Ohne diese Abhängigkeit könnten unter der Verwendung des Spring-Cloud-Gateways Aufrufe über mehrere Services hinweg nicht zusammen ausgewertet werden. Die Spans würden in diesem Fall nicht über das Gateway weitergeleitet werden.

Zur Konfiguration von Jaeger wurde im Gateway und in den einzelnen Services die Klasse JaegerConfig erstellt. Die Implementierung dieser Klasse wird auf Abbildung 44 dargestellt.

<sup>126</sup> (Eigene Darstellung)

<sup>127</sup> (Jaeger, 2021)

<sup>128</sup> (Özal, 2020)

```
@Configuration
public class JaegerConfig {

    @Bean
    public WebClient webClient() {
        return WebClient.create();
    }

    @Bean
    public JaegerTracer jaegerTracer() {
        return new io.jaegertracing.Configuration("besucherservice")
            .withSampler(new
io.jaegertracing.Configuration.SamplerConfiguration().withType(ConstSampler.TYPE)
            .withParam(1))
            .withReporter(new
io.jaegertracing.Configuration.ReporterConfiguration().withLogSpans(true))
            .getTracer();
    }
}
```

Abbildung 44 JaegerConfig.java <sup>129</sup>

In der IT-KoM Verwaltung wurde die All-in-One Lösung von Jaeger verwendet. Diese bietet eine einfache Anwendung für Testzwecke. Sie enthält eine Benutzeroberfläche und eine integrierte in Memory Speicherkomponente. Starten lässt sich die Anwendung über ein vorgefertigtes Docker Image welches von DockerHub heruntergeladen wurde. Dazu wurde der Befehl `docker pull jaegertracing/all-in-one` in der Konsole eingegeben. Wird der Container gestartet dann lässt sich im Anschluss die Jaeger-Benutzeroberfläche im Browser über die URL `http://localhost:16686/search` aufrufen. Auf Abbildung 45 wird die Benutzeroberfläche dargestellt. <sup>130</sup>

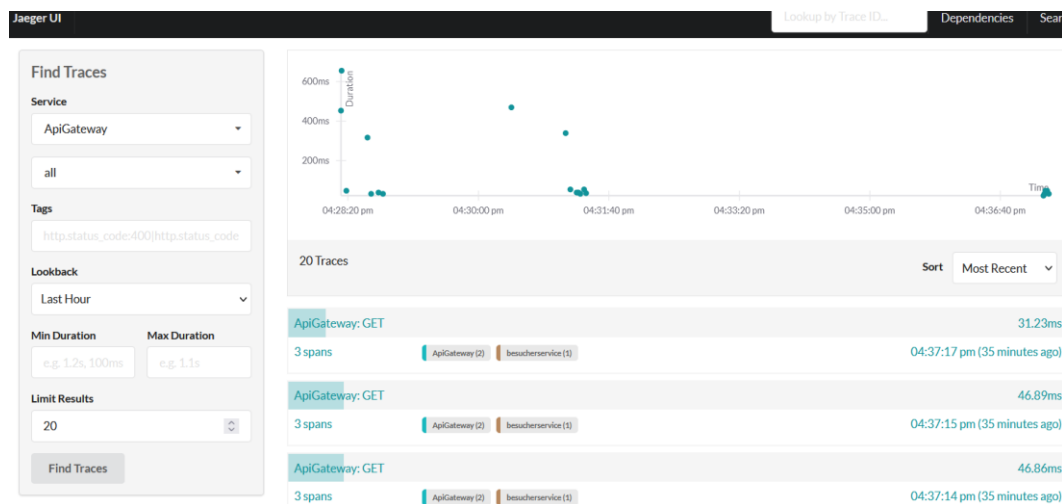


Abbildung 45 Jaeger Benutzeroberfläche <sup>131</sup>

Über die Jaeger-UI kann zum Beispiel ein Aufruf des Firmenservice über den Besucherservice ausgewertet werden.

<sup>129</sup> (Eigene Darstellung)

<sup>130</sup> (Jaeger, 2021)

<sup>131</sup> (Eigene Darstellung)

Dazu wird die URL `http://localhost:8081/firmenservice/allCompanies` im Browser aufgerufen. Der Aufruf wird über das API Gateway an den Besucherservice geleitet. Dieser sendet über Feign einen Request an den Firmenservice, welcher als Antwort die Firmendaten an den Besucherservice sendet. Die Aufrufe bilden zusammen einen Trace. Die einzelnen Aufrufe bilden die Spans. Der Trace wird zusammen mit den Spans auf der Benutzeroberfläche aufgelistet. Dauer für Trace und Spans kann per Jaeger UI ausgewertet werden.

Abbildung 46 zeigt Trace und Spans der Funktion `allCompanies` des Besucherservice.

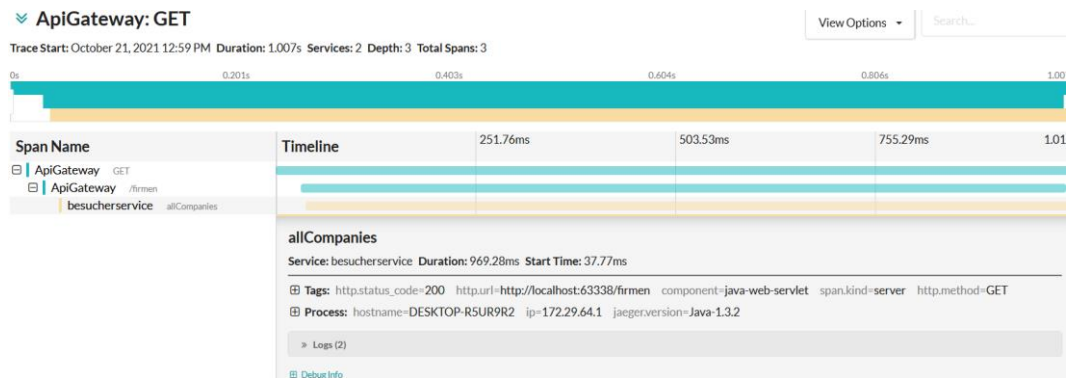


Abbildung 46 Trace eines Funktionsaufrufes der Funktion `allCompanies` vom Besucherservice<sup>132</sup>

Es lässt sich erkennen, dass die Spans des Firmenservices nicht aufgelistet wird. Aufgrund einer Inkompatibilität zwischen den aktuellen Versionen von Feign und Jaeger wird der Trace durch die Verwendung des Feign Clients unterbrochen. Daher wird der Span dem Trace nicht zugeordnet. Der Trace lässt sich allerdings einzeln unter den Traces des Firmenservices auswerten. Dabei zeigt sich ein Nachteil von Microservices. Ein Microservice allein kann fehlerfrei funktionieren. Rufen sich Microservices gegenseitig auf, können jedoch unvorhersehbare Fehlfunktionen auftreten. Die Wahl für das Ausgereiftere Tool Zipkin welches aktuell mit Feign kompatibel ist, wäre die bessere Entscheidung gewesen.

Ein weiteres Beispiel für die Verwendung von Jaeger zeigt ein Test des Load Balancers. Dazu wurden zwei Instanzen des Firmenservices gestartet. Ein Firmenservice wurde mit dem Port 61603 gestartet. Der andere wurde mit dem Port 54471 gestartet. Für den Funktionstest des Load Balancers wurde zehnmal hintereinander die URL `http://localhost:8081/besucherservice/firmen` aufgerufen. Für alle zehn Aufrufe wurde jeweils die Adresse des aufgerufenen Firmenservices in der Jaeger UI geprüft. Der erste, dritte, fünfte, siebte und neunte Aufruf nutzte den Firmenservice unter der Adresse `http://localhost:61603/allCompanies`. Der zweite, vierte, sechste, achte und zehnte Aufruf wurde von der Instanz des Firmenservices unter der Adresse `http://localhost:54471/allCompanies` bearbeitet. Die Aufrufe des Firmenservices wurden unter den beiden Instanzen des Services gleichmäßig verteilt. Der Funktionstest des Load Balancers war für diesen Anwendungsfall erfolgreich.

<sup>132</sup> (Eigene Darstellung)



Abbildung 47 zeigt die Adressangabe eines Aufrufs (im Feld http.url) in der Jaeger UI.

### ✔ firmenservice: allCompanies

Trace Start: October 16, 2021 4:31 PM Duration: 24.381ms Services: 1 Depth: 2 Total Spans: 2

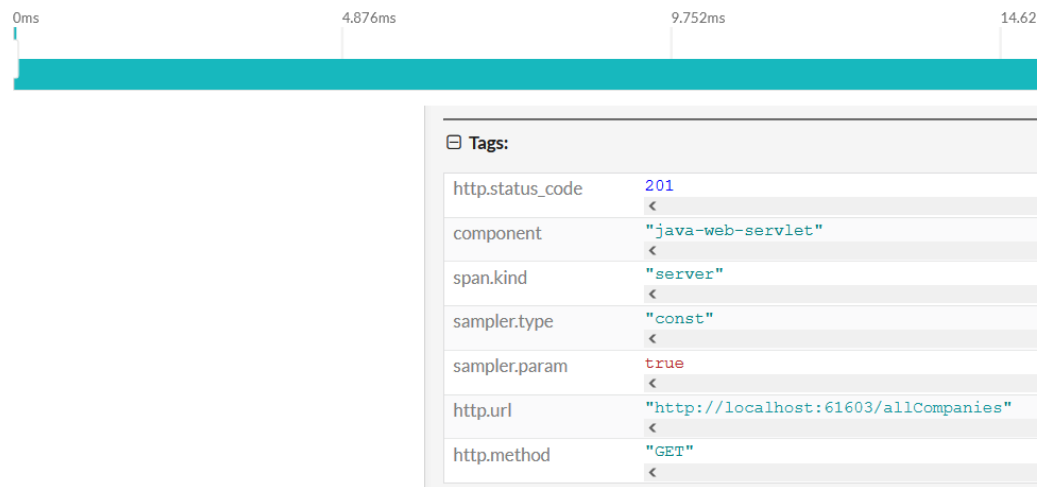


Abbildung 47 Firmenservice Adressangabe <sup>133</sup>

---

<sup>133</sup> (Eigene Darstellung)

## 6 Auswertung

Die Auflistung der einzelnen Technologien und Entwurfsmuster, welche für die Anwendung eingesetzt wurden, hat dargestellt, wie komplex sich die Umsetzung von Microservices gestaltet. Es hat sich gezeigt, dass ein einzelner Microservice relativ leicht zu verstehen und umzusetzen ist. Aus den in der Arbeit genannten Implementierungsansätzen ein in sich geschlossenes funktionsfähiges System zu erstellen, kann Entwickler jedoch vor eine Herausforderung stellen. Es wurde demonstriert, wie mit der Wahl eines Frameworks wie Spring und der Verwendung einer Vielzahl von Entwurfsmustern, Bibliotheken, Tools und Protokollen, eine Microservice-Architektur umgesetzt werden kann.

Es konnten für alle beschriebenen Problemstellungen Lösungsmöglichkeiten anhand des Prototyps implementiert werden. Im Rahmen dieser Arbeit konnte dabei allerdings nicht auf Details eingegangen werden. Der Prototyp könnte jedoch weiter ausgebaut und von der Fachhochschule Erfurt als Grundlage für die Entwicklung eines realen Systems zur Verwaltung der IT-Kontaktmesse verwendet werden.

Für den Funktionsumfang, welcher implementiert wurde, waren die genannten Verfahren ausreichend. Es wurde zum Beispiel die Funktionsfähigkeit für Service Discovery mit Eureka oder die Möglichkeit der synchronen Kommunikation unter den Microservices über Feign belegt. Die Implementierung einer kompletten Anwendung wäre jedoch noch deutlich komplexer. Der Funktionsumfang des Prototyps im Rahmen dieser Arbeit deckt nur wenige Anwendungsfälle ab. Dazu gehört zum Beispiel das Anzeigen aller Firmendaten auf der Besucherwebseite. Es wurde unter anderem keine Möglichkeit für asynchrone Kommunikation implementiert, welche eingesetzt werden könnte, um Änderungen von Daten in Datenbanken mehrerer Microservices über einen einzigen Aufruf durchzuführen. Es wurde auch nicht getestet, wie sich ein solches System unter einer Vielzahl gleichzeitiger Benutzeranfragen verhält.

Die Ermöglichung von Authentifizierung und Autorisierung über Keycloak war bereits für den geringen Funktionsumfang des Prototyps sehr komplex. Funktionalitäten wie zum Beispiel die Vergabe mehrerer Benutzerrollen zur Umsetzung von Autorisierung konnten unter dem Rahmen des Projekts nicht realisiert werden. Daraus lässt sich schlussfolgern, dass die Entwicklung einer vollständigen Anwendung noch deutlich schwieriger und umfangreicher gewesen wäre.

Ein Großteil der genannten Verfahren wie zum Beispiel die Verwendung des Circuit Breaker Patterns mit Hystrix wären für die Umsetzung einer monolithischen Architektur überflüssig gewesen. Die Zeit für die Integration dieser Verfahren könnte gespart werden. In der Praxis müssten sich neue Entwickler mit den Verfahren vertraut machen, wodurch ein zusätzlicher Zeitaufwand entsteht.

Der Einsatz aller Technologien und Entwurfsmuster wurde begründet. Es besteht die Möglichkeit teilweise darauf zu verzichten. Inwieweit die einzelnen Technologien für eine erfolgreiche Umsetzung der IT-KoM Verwaltung beitragen, wird in folgenden Abschnitt beschrieben:

- **Framework**

Unter der Verwendung von Spring Boot konnten mit wenig Code in kurzer Zeit die einzelnen Microservices erstellt werden. Der Code der einzelnen Microservices enthält durch die Vorgaben des Frameworks eine einheitliche Struktur. Die Bereitstellung der Anwendung und die Verwaltung der Abhängigkeit externer

Bibliotheken konnte mit Maven relativ einfach umgesetzt werden. Eine Umsetzung ohne Framework wäre möglich gewesen. Die Entwicklung wäre jedoch dabei deutlich komplexer. Für eine schnelle Entwicklung und zur Erfüllung des Qualitätsziels einer guten Wartbarkeit sollte auf ein Framework nicht verzichtet werden.

- **Service Discovery und API Gateway**

Ohne Service Discovery müssten stets alle aufrufbaren Adressen fest kodiert werden. Adressänderungen hätten immer Anpassungen des Codes zur Folge. Ohne Service Discovery wäre der Einsatz von Loadbalancing nicht möglich gewesen. Aufgrund des Einsatzes eines API Gateways muss bei der Kommunikation des Clients mit den einzelnen Microservices nur noch die Adresse des Gateways für den Client bekannt sein. Für die Umsetzung des Prototyps mit geringem Funktionsumfang wäre all das nicht zwingend notwendig gewesen. Für eine komplette Anwendung, bei der sich die Anzahl der einzelnen Microservices ständig ändern kann und welche dabei wartbar und zuverlässig bleiben soll, wäre der Einsatz von Service Discovery und API Gateway jedoch nicht verzichtbar.

- **Autorisierung und Authentifizierung**

Die Accountverwaltung über Keycloak wäre auch bei der Entwicklung einer kompletten Anwendung nicht nötig gewesen. Es wäre möglich gewesen einen eigenen Autorisierungsserver und Accountverwaltungsservice zu implementieren. Mithilfe von Keycloak konnte die Entwicklungszeit dafür eingespart werden. OAuth2 ist der Standard zur Umsetzung von Autorisierung in einer Microservice-Architektur. Sicherheit wurde nicht als hochpriorisiertes Qualitätsziel eingestuft, dennoch muss jede Anwendung, welche die Verwaltung von Nutzerdaten ermöglicht, gewisse Sicherheitsrichtlinien erfüllen. Daher sollte auf den Einsatz von OAuth2 nicht verzichtet werden. Eine Möglichkeit zur Benutzeranmeldung über Fremdsysteme wie zum Beispiel Facebook ist für die Anwendung nicht notwendig. Daher ist die Verwendung von OpenID Connect nicht zwingend erforderlich. Bei der Verwendung des Spring Frameworks sollte unter anderem beim Einsatz von Sicherheitsfiltern, nicht auf die Features von Spring Security verzichtet werden. Diese konnten schnell und einfach darüber implementiert werden.

- **Load Balancer**

Die Verwendung eines Load Balancers trägt zum Einhalten des Qualitätsziels Zuverlässigkeit bei hoher Last bei. Die voraussichtliche Systemlast wurde als Anforderung für die Anwendung im Rahmen dieser Arbeit nicht bestimmt. Es wurde nicht geprüft, ab welcher Last mehrere Instanzen eines Microservices zum Ausgleich der Last gestartet werden sollten. Solange das System mit jeweils nur einer Instanz von jedem Microservice stabil läuft, ist der Einsatz eines Load Balancers nicht erforderlich.

- **Synchrone Kommunikation**

In einer Microservice-Architektur sollen die einzelnen Microservices möglichst voneinander unabhängig gehalten werden. Damit aus den einzelnen Services ein in sich geschlossenes System werden kann, lässt sich eine Kommunikation unter

den einzelnen Services nicht immer vermeiden. Die Implementierung der synchronen Kommunikation wurde beispielhaft anhand der Kommunikation zwischen dem Besucherservice und dem Firmenservice dargestellt. Der Besucherservice wurde nur benötigt, um gebündelt Daten mehrerer Microservices im Frontend auf einer Seite auszugeben. Es wäre möglich gewesen, im Frontend auf eine Gesamtübersicht mit den Daten mehrerer Microservices zu verzichten. Um die Daten der Microservices Firmenservice, Vortragservice und Newsletterservice konsistent zu halten, wäre eine Kommunikation unter den Microservices nicht verzichtbar. Diese könnte unter Umständen auch asynchron erfolgen.

- **Circuit Breaker**

Ein Circuit Breaker Pattern kommt bei synchroner Kommunikation zum Einsatz. Es wird eingesetzt, um ein verteiltes System bei Ausfällen unter hoher Last zu stabilisieren. Zur erfolgreichen Implementierung des Prototyps war der Circuit Breaker nicht erforderlich. Weil Ausfälle von Microservices bei einer Vielzahl von Microservices nicht vollständig ausgeschlossen werden können, sollte bei einem Ausbau der Anwendung unter Verwendung von synchroner Kommunikation das Circuit Breaker Pattern verwendet werden.

- **Distributed Tracing**

Distributed Tracing trägt nicht dazu bei, Funktionalitäten der Anwendung dem Benutzer zur Verfügung zu stellen. Es dient zum Überwachen der Anwendung und verbessert dadurch die Wartbarkeit. Für die erfolgreiche Umsetzung des Prototyps war die Verwendung von Distributed Tracing nicht erforderlich. Jedoch wird der Einsatz mit zunehmender Komplexität zunehmend lohnenswerter. Damit bei einer komplexen verteilten Anwendung verschachtelte Aufrufe für Entwickler nachvollziehbar und auswertbar bleiben, sollte bei zunehmender Komplexität über den Einsatz von Distributed Tracing nachgedacht werden.

- **Container**

Zum Bereitstellen des Prototyps war es nicht notwendig auf Containersoftware zurückzugreifen, weil dieser nur lokal ausgeführt und getestet wurde. Damit die einzelnen Microservices bei einer Verwirklichung der IT-KoM Verwaltung möglichst schnell unter Verwendung weniger Ressourcen vom Hochschulrechenzentrum bereitgestellt werden können, sollten dabei eine Containersoftware wie Docker verwendet werden.

## 7 Zusammenfassung und Ausblicke

Im Verlauf dieser Arbeit wurden wichtige Frameworks, Entwurfsmuster, Bibliotheken, Protokolle und Tools zur Verwirklichung einer Microservice-Architektur vorgestellt. Es wurde eine Architektur zur Umsetzung der IT-KoM Verwaltung erstellt. Dabei wurde Priorität auf Einhaltung der Qualitätsziele Wartbarkeit, Bedienbarkeit, Zuverlässigkeit und Integrität als wichtigste Qualitätsziele festgelegt. Es wurde ein Prototyp implementiert, welcher die Eignung der genannten Verfahren zur erfolgreichen Realisierung der IT-KoM Verwaltung für Anwendungsfälle wie zum Beispiel der Kommunikation zwischen zwei Microservices bestätigte. Der Prototyp zeigte allerdings auch Problemstellungen auf. Der Einsatz von Jaeger führte zu Kompatibilitätsproblemen mit Feign. Daher sollte über den Einsatz einer Alternative wie zum Beispiel Zipkin nachgedacht werden.

Um einen optimalen Betrieb der Anwendung zu gewährleisten, muss der Prototyp weiter ausgebaut werden. Die Verwendung asynchroner Kommunikation könnte zum Beispiel über RabbitMQ realisiert werden. Zusätzlich könnte unter Verwendung der asynchronen Kommunikation ein Konfigurationsserver realisiert werden, welcher es ermöglichen würde, Eigenschaften welche in der Datei `applications.properties` in den einzelnen Microservices definiert wurden, von zentraler Stelle aus festzulegen. Dadurch würde die Wartbarkeit verbessert werden. Die Architektur ermöglicht es, neue Microservices wie zum Beispiel einen Chat zwischen Hochschule und einzelnen Firmen unabhängig zu entwickeln. Bestehende Microservices können mit geringem Aufwand ausgetauscht werden. Das System kann daher im Laufe der Zeit stark wachsen. Auch die Verwendung anderer Programmiersprachen oder Frameworks wäre bei der Erstellung weiterer Microservices theoretisch möglich. Dabei müsste jedoch geprüft werden, wie weit der Konfigurationsaufwand zum Beispiel bei der Verwendung von Eureka mit einer anderen Programmiersprache steigen würde. Über die Verwendung von OpenID Connect könnte ein Login für FH-Mitarbeiter über einen allgemeinen Hochschulaccount ermöglicht werden.

Beim weiteren Ausbau sollte eine Einschätzung erfolgen, welcher Last das System in Spitzenzeiten standhalten muss. Spannend wären die Fragen ab wann der Einsatz eines Load Balancers für den Erhalt der Zuverlässigkeit des Systems relevant wird oder wie die eingesetzten Circuit Breaker am besten konfiguriert werden müssten, um das System bei Ausfällen möglichst stabil zu halten.



### III. Literaturverzeichnis

**Alzve, João. 2021.** golem. [Online] 19. Juli 2021. [Zitat vom: 07. August 2021.] <https://www.golem.de/news/verteilte-systeme-die-haeufigsten-probleme-mit-microservices-2107-157885.html>.

**Anicas, Mitchell. 2014.** DigitalOcean. [Online] 21. July 2014. [Zitat vom: 19. August 2021.] <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>.

**Anil, Nish. 2021.** docs.microsoft. [Online] 28. September 2021. [Zitat vom: 24. August 2021.] <https://docs.microsoft.com/de-de/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>.

**Augsten, Stephan. 2019.** Dev Insider. [Online] 7. Juni 2019. [Zitat vom: 22. August 2021.] <https://www.dev-insider.de/was-ist-das-spring-framework-a-829846/>.

—. **2018.** Dev Insider. [Online] 27. April 2018. [Zitat vom: 09. Oktober 2021.]

—. **2021.** Dev Insider. [Online] 30. April 2021. [Zitat vom: 01. Oktober 2021.] <https://www.dev-insider.de/was-ist-distributed-tracing-a-1010389/>.

—. **2019.** Dev Insider. [Online] 02. April 2019. [Zitat vom: 16. September 2021.] <https://www.dev-insider.de/was-ist-dependency-injection-a-814452/>.

—. **2020.** dev-insider. [Online] 03. Juli 2020. [Zitat vom: 19. August 2021.] <https://www.dev-insider.de/was-ist-ein-framework-a-938758/>.

**baeldung. 2021.** Baeldung. [Online] 07. April 2021. [Zitat vom: 10. Oktober 2021.] <https://www.baeldung.com/maven>.

**Bauer, Isabelle. 2020.** heise. [Online] 07. Februar 2020. [Zitat vom: 20. August 2021.] <https://www.heise.de/tipps-tricks/URI-und-URL-was-ist-der-Unterschied-4655338.html>.

**Bayer, Thomas. 2019.** predic8. [Online] 18. November 2019. [Zitat vom: 04. September 2021.] <https://www.predic8.de/microservices-spring-boot-spring-cloud.htm>.

**Bayer, Tomas. 2019.** predic8. [Online] 10. Oktober 2019. [Zitat vom: 21. August 2021.] <https://www.predic8.de/microservices-frameworks.htm>.

**Bhuyan, Ranadeep. 2020.** Slacker. [Online] 03. März 2020. [Zitat vom: 12. September 2021.] <https://slacker.ro/2020/03/03/resiliency-two-alternatives-for-fault-tolerance-to-deprecated-hystrix/>.

**Biswanger, Gregor. 2016.** heise. [Online] 3. Mai 2016. [Zitat vom: 22. August 2021.]

**ComputerWeekly, Redaktion. 2020.** ComputerWeekly. [Online] Juli 2020. [Zitat vom: 16. August 2021.] <https://www.computerweekly.com/de/definition/Load-Balancing>.

**Docker. 2021.** docker.com. [Online] Docker, Inc., 4. August 2021. [Zitat vom: 4. August 2021.] <https://www.docker.com/products/container-runtime>.

**Donner, Andreas Dipl. -Ing und Kunkel, Richard. 2019.** Ip Insider. [Online] 06. März 2019. [Zitat vom: 21. August 2021.] <https://www.ip-insider.de/die-vorteile-des-software-load-balancings-a-801344/>.

**Fink, Andreas. 2012.** Enzyklopädie der wirtschaftsinformatik Online Lexikon. [Online] 31. 10 2012. [Zitat vom: 12. August 2021.] <https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Monolithisches-IT-System>.

**Gillis, Alexander S. 2021.** TechTarget. [Online] April 2021. [Zitat vom: 14. August 2021.] <https://whatistechtarget.com/de/definition/Service-Discovery-Diensterkennung>.

**Gnatyk , Romana . 2018.** N-iX. [Online] 03. Oktober 2018. [Zitat vom: 07. August 2021.] <https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/>.

**Guimaraes, Adelbero Fernandes. 2019.** fme. [Online] 18. Oktober 2019. [Zitat vom: 01. Oktober 2021.] <https://content.fme.de/blog/einfuehrung-in-distributed-tracing>.

**Hruschka, Peter und Starke, Gernot. 2017.** *arc42 template*. Januar 2017.

**Ionos. 2019.** Ionos. [Online] 28. Februar 2019. [Zitat vom: 01. Oktober 2021.] <https://www.ionos.de/digitalguide/websites/web-entwicklung/advanced-message-queuing-protocol-amqp/>.

—. 2019. Ionos. [Online] 25. Oktober 2019. [Zitat vom: 02. September 2021.] <https://www.ionos.de/digitalguide/websites/web-entwicklung/uniform-resource-identifier/>.

**Isheim, Roman. 2018.** it-wegweiser. [Online] 29. Juni 2018. [Zitat vom: 14. August 2021.] <https://it-wegweiser.de/microservices/>.

**Jaeger. 2021.** Jaeger. [Online] 2021. [Zitat vom: 28. September 2021.] <https://www.jaegertracing.io/docs/1.27/getting-started/>.

—. 2021. Jaeger. [Online] 2021. [Zitat vom: 20. September 2021.] <https://www.jaegertracing.io/docs/1.27/>.

**jambit. 2019.** jambit. [Online] 26. Juli 2019. [Zitat vom: 01. September 2021.] <https://www.jambit.com/aktuelles/toilet-papers/leichtgewichtige-rest-clients-mit-feign-feign-is-fine/>.

**keycloak. 2021.** keycloak. [Online] 20. August 2021. [Zitat vom: 17. September 2021.] [https://www.keycloak.org/docs/latest/server\\_admin/](https://www.keycloak.org/docs/latest/server_admin/).

**Koller, Carsten. 2018.** [Online] 20. November 2018. [Zitat vom: 07. September 2021.] <https://www.more-fire.com/blog/java-script-seo-ein-einstieg/>.

**Koller, Dirk und Augsten , Stephan. 2020.** Dev Insider. [Online] 30. November 2020. [Zitat vom: 19. Oktober 2021.] <https://www.dev-insider.de/views-mit-thymeleaf-erstellen-a-976811/>.

**Kothari, Kewal. 2020.** codeburst.io. [Online] 10. Dezember 2020. [Zitat vom: 29. August 2021.] <https://codeburst.io/circuit-breakers-the-saviour-of-your-microservices-69bce05c9e76>.

**Kurmi, Anil. 2020.** medium. [Online] 23. Januar 2020. [Zitat vom: 17. August 2021.] <https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2>.



**laverma. 2020.** La Verma. [Online] 16. September 2020. [Zitat vom: 29. August 2021.] <https://lalverma.medium.com/spring-boot-microservices-api-gateway-e9dbcd4bb754>.

**Luber, Stefan Dipl.-Ing. 2018.** Ip-Insider. [Online] 01. August 2018. [Zitat vom: 17. August 2021.] <https://www.ip-insider.de/was-ist-http-hypertext-transfer-protocol-a-691181/>.

**medium. 2020.** medium. [Online] 11. Oktober 2020. [Zitat vom: 29. August 2021.] <https://medium.com/techno101/importance-of-circuit-breaker-in-microservices-742e0550804b>.

**Mohapatra, Biswa Pujarini, Banerjee, Baishakhi und Aroraa, Gaurav. 2019.** Microservices by Example Using .Net Core. Neu-Delhi : BPB Publications, 2019, S. 2.

**mozilla.** developer.mozilla. [Online] [Zitat vom: 27. August 2021.] <https://developer.mozilla.org/en-US/docs/Web/HTTP>.

**Nagendra, Saajan. 2021.** Reflectoring. [Online] 2021. [Zitat vom: 16. September 2021.] <https://reflectoring.io/circuitbreaker-with-resilience4j/>.

**Neal, David. 2019.** okta Developer. [Online] 21. oktober 2019. [Zitat vom: 30. August 2021.] <https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc>.

**NewRelic. 2020.** New Relic. [Online] Juni 2020. [Zitat vom: 01. Oktober 2021.] <https://newrelic.com/sites/default/files/2021-08/quick-introduction-to-distributed-tracing.pdf>.

**NGINX.** NGINX. [Online] [Zitat vom: 07. September 2021.] <https://www.nginx.com/resources/glossary/load-balancing/>.

**Nish, Anil. 2021.** docs.microsoft. [Online] 28. September 2021. [Zitat vom: 02. Oktober 2021.] <https://docs.microsoft.com/de-de/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication>.

**NS1.** NS1. [Online] [Zitat vom: 24. August 2021.] <https://ns1.com/dns-service-discovery>.

**Öggl, Bernd und Kofler, Michael. 2019.** Docker, Das Praxisbuch für Entwickler und DevOps-Teams. Bonn : Rheinwerk Verlag, 2019, Bd. 1. korrigierter Nachdruck, S. 9.

**Özal, Serkan. 2020.** The New Stack. [Online] 15. Oktober 2020. [Zitat vom: 30. September 2021.] <https://thenewstack.io/jaeger-vs-zipkin-battle-of-the-open-source-tracing-tools/>.

**Plöd, Michael. 2016.** innoq. [Online] 08. Dezember 2016. [Zitat vom: 18. August 2021.] <https://www.innoq.com/de/articles/2016/12/ddd-microservices/>.

**Preissler, Jerry und Tigges, Oliver. 2015.** innoq. [Online] 09. November 2015. <https://www.innoq.com/de/articles/2015/11/docker-perfekte-verpackung-fuer-micro-services/>.

**Rauch, Gedeon und Augsten, Stephan. 2021.** Dev Insider. [Online] 16. April 2021. [Zitat vom: 03. Oktober 2021.] <https://www.dev-insider.de/was-ist-spring-boot-a-1009135/>.

**RedHat.** Red Hat. [Online] [Zitat vom: 20. August 2021.] <https://www.redhat.com/de/topics/api/what-is-a-rest-api>.

**resilience4j. 2021.** resilience4j. [Online] 2021. [Zitat vom: 03. September 2021.] <https://resilience4j.readme.io/docs/circuitbreaker>.

**Richardson, Chris. 2021.** microservices.io. [Online] 2021. [Zitat vom: 29. August 2021.] <https://microservices.io/patterns/reliability/circuit-breaker.html>.

—. **2015.** NGINX. [Online] 12. Oktober 2015. [Zitat vom: 20. August 2021.] <https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>.

**Rieger, Julia und Ego, Christian. 2020.** claranet. [Online] 14. Mai 2020. [Zitat vom: 03. September 2021.] <https://aracom.de/docker/>.

**Röwekamp, Lars und Limburg, Arne. 2016.** heise. [Online] 09. Februar 2016. [Zitat vom: 18. August 2021.] <https://m.heise.de/developer/artikel/Der-perfekte-Microservice-3091905.html?seite=all&hg=1&hgi=2&hgf=false>.

**RubyGarage. 2019.** RubyGarage. [Online] 11. April 2019. [Zitat vom: 12. September 2021.] <https://rubygarage.org/blog/top-languages-for-microservices>.

**Schwab, Michael. 2019.** Host Europe. [Online] 19. Dezember 2019. [Zitat vom: 15. August 2021.] <https://www.hosteurope.de/blog/microservices-grundlagen-und-technologien-von-verteilter-architektur/>.

**scs-architecture.** scs-architecture. [Online] [Zitat vom: 05. September 2021.] <https://scs-architecture.org/index.html>.

**Securai.** Securai. [Online] [Zitat vom: 30. August 2021.] <https://www.securai.de/veroeffentlichungen/blog/authentifizierung-microservice-verteilte-systeme/>.

**Shah, Karan. 2020.** DZone. [Online] 10. Januar 2020. [Zitat vom: 06. September 2021.] <https://dzone.com/articles/client-side-vs-server-side-rendering-what-to-choose>.

**sidion. 2019.** sidion. [Online] 28. Juni 2019. [Zitat vom: 25. August 2021.] <https://www.sidion.de/lernen/sidion-labor/blog/spring-cloud-gateway.html>.

**spring. 2015.** spring. [Online] 20. Januar 2015. [Zitat vom: 23. August 2021.] <https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka>.

—. **2021.** spring. [Online] 2021. [Zitat vom: 10. Oktober 2021.]

**Srocke, Dirk und Karlstetter, Florian. 2017.** Cloudcomputing Insider. [Online] 09. Juni 2017. [Zitat vom: 20. August 2021.] <https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/>.

**Stringfellow, Angela. 2017.** DZone. [Online] 30. Juni 2017. [Zitat vom: 10. Oktober 2021.] <https://dzone.com/articles/gradle-vs-maven>.

**te Wierik, Mattias. 2020.** medium. [Online] 11. November 2020. [Zitat vom: 19. August 2021.] <https://medium.com/swlh/authentication-and-authorization-in-microservices-how-to-implement-it-5d01ed683d6f>.

**TechTarget. 2019.** TechTarget. [Online] Februar 2019. [Zitat vom: 01. Oktober 2021.] <https://whatis.techtarget.com/de/definition/Advanced-Message-Queuing-Protocol-AMQP>.

**Vitz, Michael. 2021.** innoq. [Online] 02. September 2021. [Zitat vom: 14. September 2021.] <https://www.innoq.com/de/articles/2021/09/java-circuit-breaker-resilience4j/>.

**w3schools.** w3schools. [Online] [Zitat vom: 20. August 2021.] [https://www.w3schools.com/tags/ref\\_httpmethods.asp](https://www.w3schools.com/tags/ref_httpmethods.asp).

**Weigend, Johannes. 2019.** Informatik. *Aktuell*. [Online] 11. Juni 2019. [Zitat vom: 09. September 2021.] <https://www.informatik-aktuell.de/entwicklung/methoden/microservices-mit-go.html>.

**Wellner, Michael. 2019.** sidion. [Online] 28. Juni 2019. [Zitat vom: 29. August 2021.] <https://www.sidion.de/lernen/sidion-labor/blog/spring-cloud-gateway.html>.

**Wolff, Eberhard. 2018.** *Microservices*. Heidelberg : dpunkt.verlag GmbH, 2018, S. 32-33.

— **2018.** *Microservices*. Heidelberg : dpunkt.verlag GmbH, 2018, S. 60.

— **2018.** Das Microservices Praxisbuch. Heidelberg : dpunkt.verlag GmbH, 2018.

— **2018.** Das Microservices Praxisbuch. Heidelberg : dpunkt.verlag, 2018.

— **2018.** Das Microservices Praxisbuch. Heidelberg : dpunkt.verlag GmbH, 2018.

— **2018.** Das Microservices-Praxisbuch. Heidelberg : dpunkt.verlag GmbH, 2018, S. 62-63.

— **2017.** innoq. [Online] 04. August 2017. [Zitat vom: 08. August 2021.] <https://www.innoq.com/de/articles/2017/08/microservices-der-aktuelle-stand/>.


— **2019.** innoq. [Online] 28. Oktober 2019. [Zitat vom: 13. August 2021.] <https://www.innoq.com/de/articles/2019/10/warum-microservices-scheitern/>.

— **2018.** *Microservices*. Heidelberg : dpunkt.verlag GmbH, 2018, S. 44-45.

# Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Erfurt den 01.11.2021

A handwritten signature in blue ink, appearing to read 'B. Swarovsky', with a stylized flourish at the end.

Benjamin Swarovsky