Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Architektur einer Microservice-Anwendung und Deployment in OpenShift am Beispiel einer Partnerdatenbank

Inhaltsverzeichnis

zur

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Christoph Ruhsam, BSc

Betreuer:	DI (FH) Thomas Reidinger, 3 Banken IT GmbH
Begutachter:	FH-Prof. DI Dr. Herwig Mayr

November, 2019

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird die grundsätzliche Motivation zu Microservices und Cloudtechnologien gezeigt. Auch das Ziel der Partnerdatenbank, ein Leitfaden, sowie die Gliederung der Schrift werden in diesem Abschnitt beschrieben.

1.1 Motivation zur Architektur von Microservices

Enterprise Software Architektur entwickelt sich durch neue Architekturdesigns immer weiter. Paradigmenwechsel in Technologieumfeld und der Wunsch nach besseren und schnelleren Applikationen tragen zur Entwicklung der Architektur bei [IndrasiriSiriwardena, 2018].

Microservices wurden als Architekturdesign sehr gut aufgenommen und sind nun weit verbreitet. Sie tragen zur schnelleren und sicheren Entwicklung bei der Erstellung von Software bei. Die Microservice-Architektur stärkt den Bau von Softwaresystemen als Sammlung von unabhängigen und autonomen Services. Microservices werden durch lose Kopplung unabhängig voneinander entwickelt, deployt und skaliert. Durch den Zusammenschluss aller Microservices entsteht eine gemeinsame Applikation [IndrasiriSiriwardena, 2018].

Ein Microservice ist dabei eine Implementierung einer wohldefinierten Businessfunktionalität und ist über ein Netzwerk erreichbar. Microservices weisen ein wohldefiniertes Interface auf. Die Aufrufer des Microservice verlassen sich auf das Interface und kümmern sich nicht um die dahinter liegende Implementierung [IndrasiriSiriwardena, 2018].

Eines der Kernkonzepte von Microservices ist, dass die Architektur der Services auf Businesszwecke abgestimmt und definiert sein muss. Ein Microservice fokussiert sich lediglich auf einen Businessaspekt. Ist dies nicht der Fall, muss das Microservice in weitere Microservices zerlegt werden [IndrasiriSiriwardena, 2018].

Durch das unabhängige Deployment der Microservices kann eine unabhängige Skalierung ermöglicht werden. Da sich Businessfunktionalitäten bezüglich der Menge an Netzwerkverkehr unterscheiden, können einzelne Microservices einfach hoch skaliert werden, ohne unnötig weitere Microservices hoch skalieren zu müssen, die ohnehin wenig Netzwerkverkehr aufweisen. Dies spart Ressourcen und ist gerade beim Einsatz von Cloudtechnologien ein entscheidender Faktor [IndrasiriSiriwardena, 2018].

1.2 Motivation zum Einsatz von Cloudtechnologien

Cloudtechnologien präsentieren einen neuen Weg Applikationen zu teilen und zu deployen. Sie bieten Usern die Möglichkeit, Information im Internet gemeinsam und gleichzeitig zu bearbeiten, anzusehen und zu teilen. Die Cloud selbst ist ein Netzwerk von Datencenter, wobei jedes Datencenter eine Fülle von Rechnern aufweist, die gemeinsam interagieren. Die Cloud ist auch ein Set von Services, das eine skalierbare und anpassbare billige Infrastruktur für Entwickler und Enduser bereitstellt. Auf diese Infrastruktur kann jederzeit sehr einfach zugegriffen werden [Wang, 2012].

Cloudanbieter stellen den Usern einen großen Pool an Ressourcen zur Verfügung, mit der Möglichkeit, diese Ressourcen sehr einfach zu managen. Die wichtigsten Kernpunkte und Vorteile von Cloud Computing umfassen [Wang, 2012]:

- **Agilität**: Agilität hilft beim schnellen und billigen Provisionieren von Ressourcen.
- **Standortunabhängigkeit**: Auf Ressourcen kann von überall zugegriffen werden.
- **Teilbarkeit**: Ressourcen werden von einem großen Pool an Usern geteilt.
- **Zuverlässigkeit**: Ressourcen sind hoch verfügbar.
- **Skalierbarkeit**: Dynamische Provisionierung von Daten hilft bei der Vermeidung von Bottleneck-Szenarien.
- **Wartung**: User haben weniger Aufgaben bei Upgrades und beim Management von Ressourcen. Diese Aufgaben übernimmt der Cloud Provider.

Cloud Computing setzt dabei auf zwei wichtige Techniken [Wang, 2012]:

1. **Service-orientierte Architekturen**: Service-orientierte Architekturen beinhalten ein Set von Designprinzipien, die während der Software Entwicklung und der Software Integration genutzt werden. Das Deployment von service-orientierten Architekturen bietet ein lose gekoppeltes Set an Services, die über mehrere Businessgrenzen hinweg kommunizieren können.
2. **Virtualisierung**: Das Konzept der Virtualisierung befreit den User von Ressourcenkäufen und aufwändigen Installationen. Die Cloud bringt die Ressourcen zum User. Virtualisierung umfasst Hardware, Speicher, Speicherzugriff, Software, Daten und Netzwerke. Virtualisierung ist im Cloud Umfeld zu einem unverzichtbaren Bestandteil geworden. Durch Virtualisierung können mehrere Applikationen am selben Server laufen und gemeinsam Ressourcen teilen. Durch verschiedene Host Operating Systeme ist die Konfiguration einiger Applikationen sehr umständlich. Auch dieses Problem löst Virtualisierung durch einfache Konfiguration und Aggregation von Ressourcen. Auch das schnelle Recovery-Management ist ein großer Pluspunkt bei Virtualisierung und bei Cloud Technologien.

Durch diese Vorteile sind Cloud-Technologien in den letzten Jahren immer wichtiger und populärer geworden. Ein Anbieter einer Software kann sich heutzutage Downtimes seines Produkts nicht mehr leisten. Viele Kunden wechseln durch die Fülle an Angeboten schnell zum Konkurrenten. Cloud Provider bieten durch Service Level Agreements meist eine Verfügbarkeit von über 99%. Auch deshalb hosten viele Softwarefirmen nicht mehr selbst, sondern deployen diese in Cloud Systemen [Wang, 2012].

1.3 Zielsetzung der Implementierung der Partnerdatenbank

Bei der 3 Banken IT GmbH gibt es zusätzlich zu den Kunden, wie die BKS Bank, Oberbank und die BTV Vierländer Bank, auch Partner, wie z.B. die FH Oberösterreich Campus Hagenberg oder die JKU Linz. Diese Partner werden aktuell in verschiedenen Datenquellen geführt. Die aktuellen Datenquellen der 3 Banken IT GmbH bestehen aus:

- **FiPe/Doxis** Sobald Dokumente gescannt werden, wird im FiPe der Partner mit Kürzel, Name und laufender Nummer erfasst. Dadurch kann im Doxis auf das Kürzel und die laufende Nummer aus der FiPe zugegriffen werden.
- **SAP**: Sobald Rechnungen eintreffen, wird der Partner im SAP als Kreditor mit Name, Adresse und laufender Nummer aus dem FiPe und Bankverbindungen erfasst.
- **Art28_Verträge_Fernwartungszugänge_VPE**: Diese Daten werden ab der Angebotsanfrage/Ausschreibung benötigt.
- Ansprechpersonen mit **Kontaktdaten** von den Anwendungsverantwortlichen bei den einzelnen Mitarbeitern.

Ziel der Partnerdatenbank ist die Ablöse der Art28_Verträge_Fernwartungszugänge_VPE und der FiPe-Liste. Weitere funktionale Anforderungen werden wie folgt definiert:

- Es können Unternehmen als Partner angelegt werden.
- Zu diesen Unternehmen können Kontaktpersonen hinzugefügt werden.
- Zu Kontaktpersonen gehören verschiedene Links im Doxis. Auch diese werden in der Applikation angezeigt.
- Die Ersterfassung eines Partners ist durch jeden Mitarbeiter möglich.
- Es besteht die Möglichkeit einen Kontakt als privat oder öffentlich zu markieren. Dies ist wichtig, wenn jemand seine Visitenkarten darin verwaltet.
- Der Basiseintrag einer Firma ist generell öffentlich. Nur für Kontaktpersonen kann ein private Status angegeben werden.

Zu den nicht-funktionalen Anforderungen zählen:

- Eine Microsoft SQL Server Datenbank.
- Spring-Boot als Backendtechnologie.
- Angular als Frontendtechnologie.
- Ein rollenbasierter Login mit Username und Passwort.
- Das Backend als Microservice-Architektur.
- Eine Infrastruktur zum Deployment in OpenShift.

Weiters sollen zusätzliche Tools eingebunden werden, die bei der Entwicklung von Microservices und dem Deployment in OpenShift helfen. Zu diesen Tools zählen:

- **Jenkins**: Dient für automatisierte Test-, Build- und Deployment-Pipelines.
- **Spring-Retry**: Dient zur Fehlerbehandlung speziell bei Microservices.
- **Swagger**: Dient zur REST-Schnittstellenbeschreibung.
- **Jaeger**: Dient zum Tracing von Serviceaufrufen.
- **Docker**: Dient zur Containerisierung der einzelnen Services.
- **Fabric8**: Dient zum Deployment der Services in OpenShift.

Durch die Anwendung dieser Tools, der Microservice-Architektur und dem Deployment in OpenShift soll der 3 Banken IT GmbH ein Prototyp einerseits für die Weiterentwicklung der Partnerdatenbank, andererseits eine Schablone für weitere Anwendungen geboten werden.

1.4 Ziel des Deployments der Partnerdatenbank in OpenShift

Cloud Services, wie OpenShift weisen grundsätzlich folgende drei Charaktereigenschaften auf:

1. Nutzer können eine große Anzahl an Ressourcen jederzeit nutzen.
2. Cloud Services sind elastisch. Ein Nutzer kann jederzeit so viele Ressourcen wie möglich und so viele wie nötig nutzen.
3. Das Service wird gänzlich vom Provider gemanagt. Die Nutzer benötigen lediglich Internetzugang und Rechner, um die Services aufzurufen.

Durch die Nutzung von OpenShift und das Deployment der Anwendung in OpenShift muss die 3 Banken IT GmbH keine eigenen Ressourcen zum Hosten der Anwendung bereitstellen. Viele Cloud Provider bieten ein Service Level Agreement von über 99% an. Das bedeutet, dass der Cloud Provider sicherstellt, dass die Infrastruktur zu 99% der Zeit läuft.

OpenShift ist eine Open-Source Platform-as-a-Service Platform. OpenShift bietet automatische Installation, Upgrades und Lifecycle-Management des gesamten Container Stacks auf jeder Cloud Plattform. Zum Container Stack gehören das Operating System, Kubernetes, Cluster Services und Applikationen [RedHatInc, 2019].

OpenShift hilft Teams auch beim schnellen und agilen Erstellen von Anwendungen. Dadurch, dass Befehle in der Dev-Stage dieselben wie in der Production-Stage sind, können Entwickler noch sicherer und einfacher Applikationen entwickeln. Gerade bei Software für Banken ist die Sicherheit wesentlich.

Auch dadurch, dass die 3 Banken IT GmbH bereits Applikationen in OpenShift deployt, ist die Wahl der Cloud Infrastruktur der Partnerdatenbank auf OpenShift gefallen.

1.5 Leitfaden und Gliederung der Schrift

Die Schrift gliedert sich in folgende neun Kapitel:

1. **Einleitung**: In diesem Kapitel wird die Motivation von Microservices und Cloudtechnologien beschrieben. Auch die Zielsetzung der Implementierung der Partnerdatenbank, sowie das Ziel des Deployments in OpenShift wird erklärt.
2. **Service-orientierte Architektur und Microservices**: In diesem Kapitel wird die Abgrenzung service-orientierter Architekturen zu Microservices beschrieben. Auch ein Vergleich zu monolithischen Architekturen und die Charakteristiken, sowie die Vor- und Nachteile von Microservices werden gezeigt.
3. **Containerisierung mit Docker**: In diesem Kapitel wird die Notwendigkeit der Containerisierung von Cloud Applikationen am Beispiel von Docker gezeigt.
4. **OpenShift**: In diesem Kapitel wird OpenShift, sowie die Container-as-a-Service Plattform Kubernetes, auf welcher OpenShift aufsetzt, gezeigt.
5. **Partnerdatenbank**: In diesem Kapitel wird der Grundaufbau und Zweck der Partnerdatenbank, sowie die Beschreibung des Backends und Frontends gezeigt.

6. **Design der Partnerdatenbank**: In diesem Kapitel wird das Design und die Architektur der Partnerdatenbank gezeigt.
7. **Implementierung der Partnerdatenbank**: In diesem Kapitel wird die Implementierung der Partnerdatenbank und auch die Integration der Tools, wie Jenkins, Swagger oder Jaeger, gezeigt.
8. **Evaluierung der Anwendung**: In diesem Kapitel wird die Architekturevaluierung, Evaluierung des Frontends und Whitebox-, sowie Blackbox-Tests gezeigt.
9. Abschließend gibt der Autor eine **Zusammenfassung** und einen Ausblick über die weitere Entwicklung der Partnerdatenbank.

Kapitel 2

Serviceorientierte Architektur und Microservices

Das folgende Kapitel behandelt Microservices, ihre Vor- und Nachteile und die Abgrenzung zu serviceorientierten Architekturen sowie die Unterschiede zu monolithischen Architekturen.

2.1 Definition und Abgrenzung

Im Folgenden wird die Definition von Microservices und die Abgrenzung zu serviceorientierten Architekturen beschrieben.

2.1.1 Definition von Microservices

Microservices sind ein Ansatz, um große Softwarearchitekturen in kleine, konsistente, voneinander klar abgegrenzte Services zu zerlegen. Diese Services sind isoliert und kommunizieren miteinander. [Posta, 2016]

Microservices werden typischerweise von kleinen Teams implementiert, gebaut und deployt. Diese kleinen Services sind so autonom, dass das Team, welches für den Service zuständig ist, die Implementierungsdetails ändern kann, ohne großen Einfluss auf das gesamte restliche System zu nehmen. Jedes Team ist selbst für sein Service verantwortlich. Das Team muss für die Aufgabenstellung die richtige Technologie wählen, das Service deployen und managen und etwaige Fehler beheben. Bei Microservice-Architekturen kann die Abgrenzung der verschiedenen Services ganz klar definiert werden. [Posta, 2016] Dies hilft, um [Posta, 2016]:

1. die Logik des Services einfach zu verstehen, ohne den Kontext der gesamten Applikation kennen zu müssen.
2. den Service schnell lokal bauen und ausführen zu können.
3. die richtige Technologie für ein Problem zu nützen.
4. den Service einfach und schnell testen zu können, ohne die gesamte Applikation dabei hochfahren zu müssen.
5. schnellere Releasezeiten zu ermöglichen.
6. schnellere horizontale Skalierung zu ermöglichen.
7. die Belastbarkeit des Systems zu verbessern.

Die oben angeführten Punkte werden im Folgenden näher ausgeführt [Posta, 2016]:

1 Logik: Jedes einzelne Microservice sollte nur einen Teil der gesamten Anwendung ausmachen. Ein Microservice deckt meist nur einen Geschäftsfall der gesamten Businesslogik ab. Dieser Geschäftsfall ist für sich einfacher zu verstehen als das gesamte System. Kommt ein neuer Entwickler in das Team, muss er sich nur in den Code des Microservice, an dem er arbeitet, einlesen und muss nicht das gesamte System verstehen.

2 Bauen und Ausführen: Muss bei einem neuen Feature die gesamte Anwendung gebaut, deployt und ausgeführt werden, ist dies sehr aufwändig und kostet sehr viel Zeit. Ein Microservice hingegen steht für sich selbst und kann einfach lokal gestartet und getestet werden. Dies erleichtert dem Entwickler die Arbeit und spart Zeit.

3 Technologie: Meist werden für bestimmte Geschäftsfälle verschiedene Technologien benötigt. Ist die gesamte Businesslogik in einem Monolithen abgebildet, kann lediglich eine Technologie eingesetzt werden. Ist aber z.B. für einen bestimmten Geschäftsfall Machine Learning nötig, ist eventuell Java die falsche Lösung und man setzt lieber auf Python. Microservices sind unabhängig voneinander und kommunizieren meist durch REST oder SOAP miteinander. Dadurch kann in jedem Service die gewünschte Technologie eingesetzt werden.

4 Testen: Ähnlich zu 2, wo das Bauen und Ausführen eines Microservice beschrieben wird, kann ein Service auch unabhängig vom gesamten System getestet werden.

5 Releasezeiten: Heutzutage müssen neue Features schnell ausgeliefert werden, ansonsten ist womöglich der Konkurrent schneller und man verliert Kunden. Muss bei einem neuen Release das gesamte System neu deployt werden, sind damit verschiedenste Risiken verbunden. Wird z.B. ein fehlerhafter Code deployt, stürzt der gesamte Monolith ab. Bei Microservices ist in dieser Zeit lediglich das neu deployte Service nicht verfügbar.

6 Skalierung: Hat man ein monolithisches System und möchte einen neuen Geschäftsfall implementieren, muss dieser in das bestehende System eingeflochten werden, was dazu führen kann, dass ein bestehender Code nicht mehr so funktioniert wie gewollt. Bei einer Microservice-Architektur wird ein neues Service für diesen Geschäftsfall angelegt und bestehender Code nicht oder kaum für Aufrufe des neuen Service geändert.

7 Belastbarkeit: Fällt ein Service aus, heißt das nicht, dass das gesamte System nicht mehr verfügbar ist. Teile des Systems können meist ohne Probleme weiterverwendet werden.

2.1.2 Abgrenzung serviceorientierter Architekturen und Microservices

Microservices und service-orientierte Architekturen gehören beide zu den service-basierten Architekturen. Das heißt, dass beide Architekturen einen großen Schwerpunkt auf Services als primäre Architekturkomponenten legen. Obwohl Microservices und serviceorientierte Architekturen sehr unterschiedliche Muster aufweisen, haben sie auch vieles gemeinsam [Richards, 2016].

Alle serviceorientierten Architekturen sind verteilte Architekturen. Die Servicekomponenten werden über Remote-Protokolle, wie z.B. Representational State Transfer (REST), Simple Object Access Protocol (SOAP) oder Java Message Service (JMS) angesprochen. Beide bieten viele Vorteile im Vergleich zu monolithischen und schichten-basierten Architekturen. Sie bieten höhere Skalierbarkeit, losere Kopplung zwischen den Services und bessere Kontrolle über Entwicklung, Tests und Deployment. Komponenten in einer verteilten Anwendung tendieren mehr dazu, in sich geschlossen zu sein und bieten daher besseres Änderungsmanagement und einfachere Wartung, was zu mehr Flexibilität und zu robusteren Anwendungen führt. Diese Architekturen führen auch zu loser gekoppelten und modulareren Anwendungen. Modularität

ist das Kapseln einzelner Teile der Anwendung in abgeschlossene Services, die dadurch individuell designt, entwickelt, getestet und deployt werden können, ohne großen Einfluss auf andere Services zu nehmen [Richards, 2016].

Sowohl bei Microservices als auch bei serviceorientierten Architekturen müssen *Service Contracts* vereinbart werden. Service Contracts werden zwischen einem Service und einem Servicekonsument (Client) vereinbart. Dadurch wird das Format der eingehenden und ausgehenden Daten spezifiziert (z.B. XML, JavaScript Object Notation [JSON], Java Object, etc.) [Richards, 2016].

Microservice-Teams bestehen meist aus maximal sieben Personen. Das Team muss die Entwicklung, das Deployment und den Betrieb des Service alleine bewältigen. Je größer das Team, desto höher ist der Kommunikationsaufwand. Jedes Teammitglied sollte die gesamte Codebasis überblicken und warten können. Zu große Teams sind meist ein Indikator dafür, dass das Service aufgesplittet werden sollte [Richards, 2016].

Service-orientierte Architekturen sind eher für große, komplexe Enterprise-Systeme geeignet, die eine Interaktion mit vielen heterogenen Applikationen und Services benötigen. Für Anwendungen, die viele Komponenten benötigen und von mehreren Services geteilt werden, sind service-orientierte Architekturen sinnvoll. Für Applikationen mit einem wohldefinierten Arbeitsablauf und wenig gemeinsam geteilten Komponenten, wie z.B. Security-Komponenten, sind Microservice-Architekturen sinnvoll. Microservice-Architekturen sind besser geeignet für kleine, wohl definierte web-basierte Systeme, wie für groß skalierbare Enterprise-Systeme. Die fehlende Middleware ist ein Faktor, weshalb Microservice-Architekturen nicht oder schlecht für komplexe Businessanwendungen geeignet sind. Microservices sollten dort eingesetzt werden, wo sich die Businesslogik auf kleine abstrahierte Geschäftsfälle herunterbrechen lässt [Richards, 2016].

Microservices und service-orientierte Architekturen sind auch bezüglich Teilen von Komponenten sehr unterschiedlich. Service-orientierte Architekturen basieren auf dem Prinzip *share-as-much-as-possible*, wohingegen Microservices auf dem Prinzip *share-as-little-as-possible* basieren. Angenommen ein großes Warenhaus benötigt zum Verwalten der Einkäufe und Verkäufe jeweils ein *Bestellservice*. Bei service-orientierten Architekturen, wie in Abbildung 2.1 zu sehen, würde man dazu das Bestell-Service einmal implementieren und die Einkaufs- und Verkaufskomponente teilen sich dieses Service. Das Bestell-Service selbst muss in diesem Fall wissen, welche Aktionen es bei den verschiedenen Anfragen ausführen muss. Obwohl das *share-as-much-as-possible*-Prinzip das Problem der Codeduplizierung löst, ist damit eine enge Kopplung von Businesskomponenten verbunden und erhöht das Risiko von Änderungen. Angenommen man ändert den Code im Business-Service, ist diese Änderung schwer zu testen, da das Service global verfügbar ist und je nach Aufruf andere Aktionen auslöst [Richards, 2016].

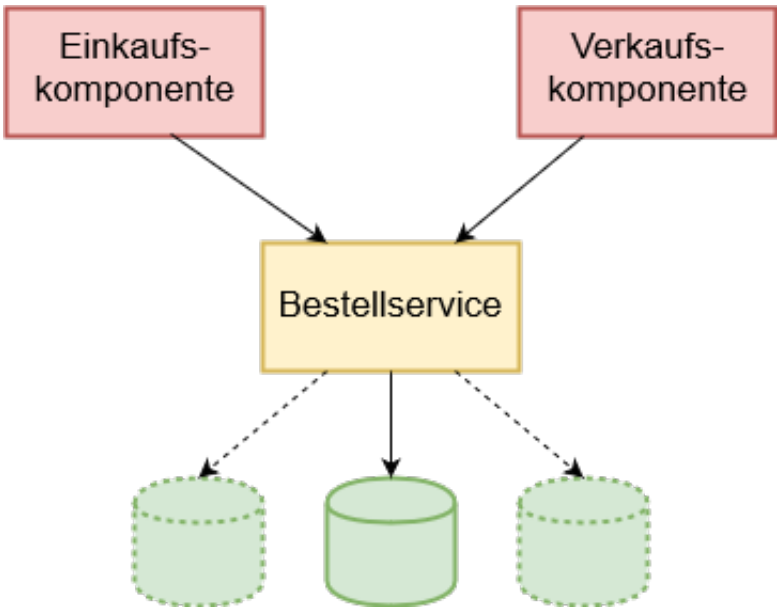

Abbildung 2.1: Service-orientierte Architektur - Bestellservice

[Richards, 2016]

Microservices basieren auf dem Konzept des *share-as-little-as-possible* und verstärken dadurch das Domain-driven Designkonzept *Bounded Context*. Microservices sind lose gekoppelt und agieren daher als einzelne geschlossene Einheit mit wenig Abhängigkeiten. Es gibt lediglich ein wohldefiniertes Interface und wohldefinierte Vereinbarungen nach außen. Realistischerweise wird es immer gemeinsam geteilte Services geben, auch in Microservice-Architekturen, z.B. Sicherheits- oder Infrastrukturservices. Service-orientierte Architekturen maximieren das Teilen von Komponenten, wohingegen Microservice-Architekturen das Teilen durch den klar definierten Bounded Context minimieren [Richards, 2016].

Es gibt viele Vorteile, die das Verstärken des Bounded Context bringt. Die Wartung der Services wird wegen der geringeren Abhängigkeiten weitaus einfacher. Dies erleichtert auch das Ändern und Weiterentwickeln des Service. Auch das Deployment gestaltet sich dadurch um einiges einfacher, da weniger Code zu deployen ist und es weniger Risiken gibt, andere Bereiche des gesamten Systems zu brechen. Dies fördert auch die Robustheit des gesamten Systems [Richards, 2016].

2.2 Vergleich zu monolithischer Architektur

Microservices und Monolithen sind sehr unterschiedliche Designstrategien, um eine Anwendung zu entwickeln. Beide Strategien haben ihre Vor- und Nachteile. Die Strategie sollte deshalb mit Bedacht gewählt werden. Die monolithische Architektur ist derzeit noch der Standardweg eine Applikation zu entwickeln [Tilkov, 2015].

Im Folgenden werden die wichtigsten Unterschiede von monolithischen und von Microservice-Architekturen beschrieben [Jangla, 2018]

2.2.1 Wartung

Microservices sind viel einfacher zu warten, da sich die Komplexität eines Services in Grenzen hält. Microservices sind modulare und unabhängige Services. Neue Entwickler können sich in die Codebasis schneller einlesen und schneller neue Features implementieren als bei monolithischen Architekturen.

Ein Monolith besteht aus einer Codebasis, in der alle Geschäftsfälle abgedeckt sind. Wird ein neues Feature hinzugefügt, wird es dabei in die bestehende Codebasis eingewebt. Dies erhöht die Komplexität des Codes und erschwert die Entwicklung und Wartung. Neue Entwickler müssen sich in die gesamte Codebasis und in den gesamten Workflow einlesen, um neue Features entwickeln zu können.

2.2.2 Deployment

Ein kontinuierliches Deployment gestaltet sich bei Monolithen sehr schwierig, da der Monolith immer größer und komplexer wird. Zudem muss immer der gesamte Monolith deployt werden, auch wenn lediglich wenig Codestücke geändert wurden. Dies ist natürlich sehr zeitaufwändig, aber auch gefährlich. Wird ein fehlerhafter Code deployt, bringt dieser den gesamten Monolith zum Stillstand.

Bei Microservices hingegen wird die Codebasis in kleine, überschaubare Services unterteilt, die einzeln und jederzeit schnell deployt werden können. Wird bei Microservice-Architekturen ein fehlerhafter Code deployt, ist es möglich, dass andere Teile des gesamten Systems noch ohne Probleme laufen.

2.2.3 Tests

Auch beim Testen eines Monolithen werden die große Codebasis und die vielen verschiedenen Anwendungsfälle zum Problem. Das Testen aller Szenarien bei einer derart großen Codebasis wird dabei schnell unübersichtlich.

Bei Microservices kann durch den Bounded Context jede Komponente individuell getestet werden. Dies erleichtert das Testen eines jeden Services ungemein und verringert zusätzlich die Fehleranfälligkeit eines Systems.

2.2.4 Startup-Zeit

Bei monolithischen Architekturen wird der gesamte Monolith hochgefahren. Je komplexer und größer dieser wird, desto mehr steigt die Startzeit.

Die Startzeit bei Microservices ist viel schneller, da jedes Service einzeln hochgefahren werden kann. Werden alle Services gemeinsam gestartet, bringt dies bezüglich der Startzeit wieder wenig.

2.2.5 Technologie

Ein Monolith wird in einer Programmiersprache entwickelt und greift meist nur auf eine Datenbank zu. Neue Technologien können nicht, bzw. nur sehr umständlich hinzugefügt werden. Die Einarbeitungszeit gestaltet sich jedoch bezüglich Technologie einfacher, da nur eine Sprache zum Einsatz kommt und neue Entwickler nur diese beherrschen müssen.

Mit Microservices können Entwickler die Technologie jedes Services selbst wählen. Da Microservices so klein und überschaubar sind, können diese schnell ersetzt werden und es kann dadurch immer die aktuellste Version oder Technologie verwendet werden.

2.2.6 Skalierbarkeit

Microservices können einfach skaliert werden. Platform-as-a-Service-Plattformen, wie OpenShift, bieten eine einfache Möglichkeit, Services je nach Bedarf hoch- und runterskalieren zu können.

Einen komplexen Monolithen zu skalieren, gestaltet sich sehr schwierig. Zum Beispiel wird ein kontinuierliches Deployment mit vielen Entwicklern und einer sehr großen Codebasis sehr kompliziert.

2.3 Charakteristiken

Im Folgenden werden die wichtigsten Charakteristiken von Microservices beschrieben, die auch für service-orientierte Architekturen gelten [Rajesh, 2016].

2.3.1 Service-Vereinbarung

Microservices werden durch wohldefinierte Service-Vereinbarungen beschrieben. Zum Datenaustausch kann z.B. REST oder SOAP verwendet werden. Als Markup-Sprache kann JSON, XML oder auch YAML eingesetzt werden. In der Welt der Microservices wird meist die Kombination von REST und JSON eingesetzt. Es gibt aber mehrere Möglichkeiten zur Definition von Service-Kontrakten. JSON Schema, WADL, Swagger und RAML sind nur ein paar wenige Beispiele.

2.3.2 Lose Kopplung

Microservices sind unabhängig und lose gekoppelt. Meist akzeptieren Microservices ein Event als Input und reagieren darauf erneut mit einem Event. Nachrichten, HTTP und REST werden häufig zur Interaktion zwischen den einzelnen Services eingesetzt. Nachrichten-basierte Endpunkte bieten dabei ein höheres Level an Entkopplung.

2.3.3 Service-Abstraktion

Service-Abstraktion bei Microservices ist nicht nur die abstrahierte Realisierung des Service, sondern bietet auch eine komplette Abstraktion von allen Bibliotheken und Umgebungsdetails.

2.3.4 Wiederverwendung

Microservices sind wiederverwendbare Business-Services. Sie werden von mobilen Geräten, Desktop-Anwendungen und auch anderen Microservices verwendet.

2.3.5 Zustandslosigkeit

Wohl-designte Microservices sind zustandslos und teilen sich nichts. Wird ein gemeinsamer Zustand in verschiedenen Services benötigt, wird dieser meist in einer Datenbank abgebildet.

2.3.6 Interoperabilität

Services sind durch Standardprotokolle und Nachrichtenaustausch-Standards interoperabel. Nachrichtenaustausch, HTTP und REST werden zum Nachrichtentransport verwendet. Eine Kombination aus REST und JSON ist die populärste Methode zum Entwickeln von interoperablen Services. Wird weitere Optimierung benötigt, kommen auch andere Protokolle, wie RabbitMQ, Avro, Zero MQ oder Protocol Buffers zum Einsatz. Der Einsatz dieser Protokolle kann dabei aber auch die Interoperabilität der Services einschränken.

2.3.7 Zusammensetzung

Microservices sind zusammensetzbar. Dies wird durch Service-Orchestrierung und Service-Choreografie erreicht. Service-Orchestrierung und Service-Choreografie sind zwei Designvarianten von Microservices. Auf diese wird im nächsten Abschnitt näher eingegangen.

2.4 Designvarianten

Im Folgenden werden die beiden Varianten *Service Orchestrierung* und *Service Choreografie* zum Design von Microservices vorgestellt.

2.4.1 Service-Orchestrierung

Bei der Service Orchestrierung gibt es ein zentrales Kompositionsservice, das das entsprechende Microservice aufruft und von diesem die Antwort erwartet. Diese Nachrichten können mittels HTTP oder Socketaufrufen durchgeführt werden. Wichtig hier ist, dass diese Aufrufe sowohl synchron als auch asynchron durchgeführt werden kann. Das aufgerufene Microservice muss also nicht sofort antworten, sondern kann auch später mit der Antwort ein Event im Kompositionsservice auslösen. In diesem Muster reden die Microservices nicht miteinander, sondern nur mit dem Kompositionsservice. Dabei muss jedoch nicht für jedes Microservice dasselbe Protokoll verwendet werden, z.B. kann das Kompositionsservice mit Service A über HTTP und mit Service B über das XML/RPC Protokoll kommunizieren [Sharma, 2017].

Wie in Abbildung 2.2 zu sehen, wird dabei das Kompositionsservice zur zentralen Autorität. Hier beginnt auch die gesamte Logik, was schlechtes Design für Microservices bedeutet. Die gesamte Logik im Kompositionsservice zu haben, führt zu einer Fülle von Abhängigkeiten zwischen den Services. Z.B. kann die Logik sein, dass wenn Service A antwortet, Service B danach aufgerufen werden muss. Dies führt zu Abhängigkeiten zwischen Service A und Service B [Sharma, 2017].

Da das Kompositionsservice der zentrale Punkt der Anwendung ist, ist dieser auch die Schwachstelle. Dieses Service muss verfügbar sein, ansonsten steht die gesamte Anwendung still. Das Kompositionsservice ist der *Single-Point-of-Failure* in dieser Architektur. Dies kann aber durch Tools und Cloud-basierte Lösungen, wie Skalieren oder Clustering, gelöst werden.

Zusätzlich ist das Orchestrierungsmuster schwer zu implementieren. Hat man einen Entscheidungspunkt, wird es schwer, dem verteilten Design zu folgen [Sharma, 2017].

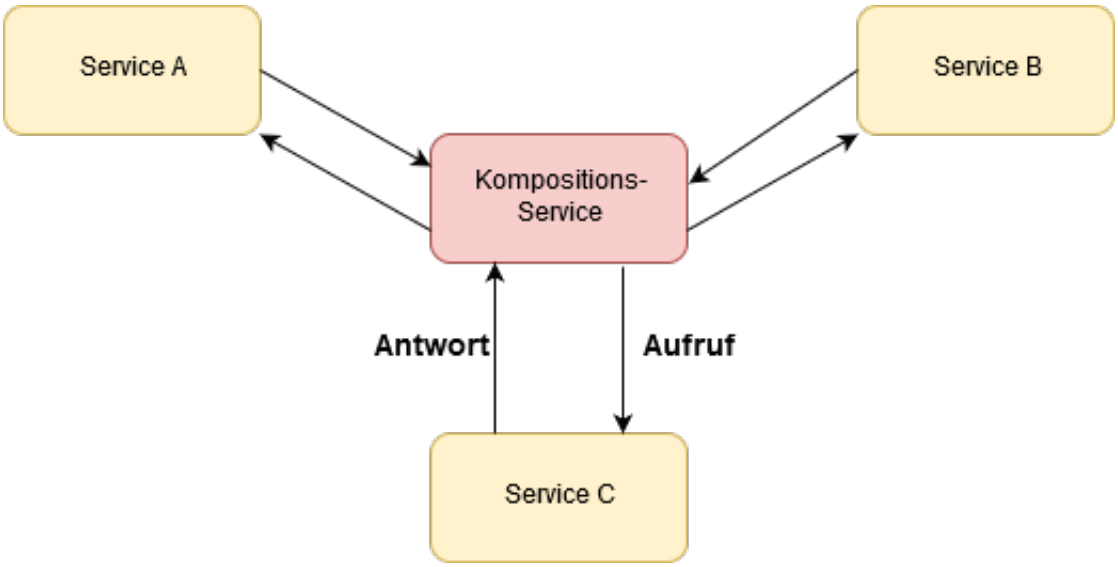

Abbildung 2.2: Varianten - Service-Orchestrierung

[Sharma, 2017]

2.4.2 Service-Choreografie

Anders als bei der Service Orchestrierung ist bei der Service-Choreographie jedes Service in Koordination mit den anderen Services auf einen bestimmten Anwendungsfall zugeschnitten. Service Orchestrierung repräsentiert die Kontrolle von einem einzelnen Standpunkt aus. Bei der Choreografie wird jedes Service in die Interaktion mit eingebunden [Sharma, 2017].

Wie in Abbildung 2.3 zu sehen, redet bei der Service Choreografie jedes Service nach Abschluss des Tasks mit einem anderen Service, um die nächste Aufgabe zu triggern. Dies kann in den Mustern *one-to-one*, *one-to-many* oder *many-to-many* geschehen. Dies kann ebenfalls synchron oder asynchron durchgeführt werden. Idealerweise gibt es ein globales Protokoll, über das die Services miteinander kommunizieren. Meistens werden diese Aufrufe asynchron durchgeführt. Da es kein zentrales Service gibt, löst jedes Service ein Event nach Abschluss seines Tasks aus. Dies macht die Services unabhängig voneinander und fördert die lose Kopplung. Andere Services registrieren sich auf dieses Event und reagieren darauf, wenn es ausgelöst wird [Sharma, 2017].

Durch nachrichtenorientierte Kommunikation kann das Choreografiemuster sehr nützlich werden. Zwischen den Services wird eine Nachrichtenschlange implementiert. Dadurch wird das Loslösen, Hinzufügen und Entfernen von Services sehr einfach gelöst. Auch die ganze Reihe an Abhängigkeiten, die oft bei Choreografiemustern entstehen, wird dadurch gelöst. Service Choreografie erhöht die Komplexität eines Systems. Jedes Service erzeugt und verarbeitet Nachrichten zur Laufzeit. Dadurch kann der exakte Stand einer Transaktion nicht herausgefunden werden [Sharma, 2017].

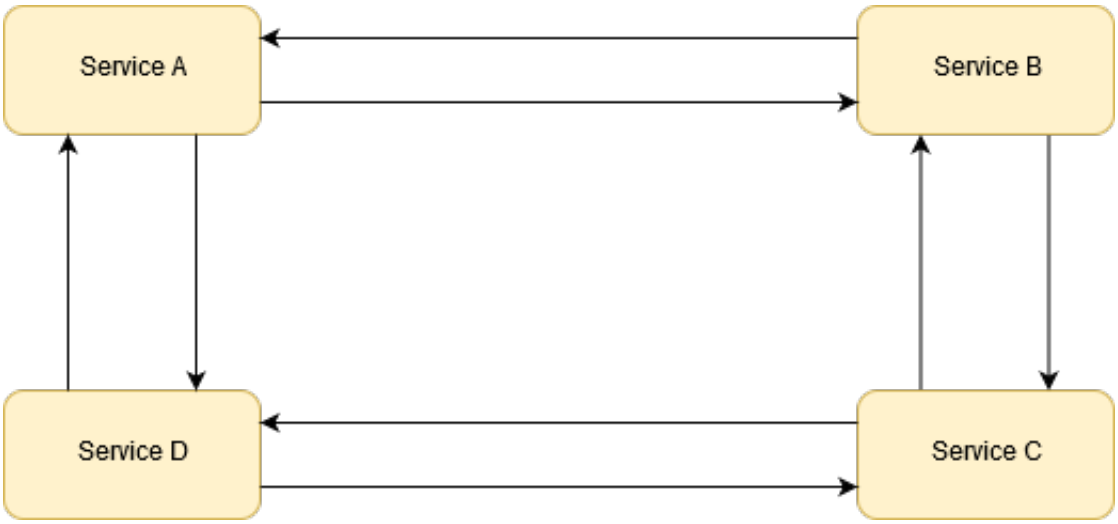

Abbildung 2.3: Varianten - Service-Choreografie

[Sharma, 2017]

2.5 Vor- und Nachteile

Im folgenden Abschnitt werden die Vor- und Nachteile von Microservices behandelt.

2.5.1 Vorteile

Technologieunabhängigkeit

Microservices werden als eigenständige Einheit betrachtet. Dadurch können die einzelnen Services verschiedene Technologien verwenden. Greift ein Service auf eine Datenbank zu und ein anderes Service führt Shell-Skripte aus, beeinflussen sich die beiden Services nicht und können jederzeit ausgetauscht werden, ohne Einfluss auf andere Services zu nehmen. Performance ist in diesem Zusammenhang ebenfalls sehr wichtig. Wird eine Technologie zu langsam, kann sie ohne Probleme durch eine neue ersetzt werden [Newman, 2015].

Skalierung

Wie erwähnt, muss bei Monolithen die gesamte Software als Ganzes deployt und skaliert werden. Dies bedeutet viel Overhead. Bei Microservices können einzelne Teile hoch- und runterskaliert werden. Cloud-basierte Technologien, wie z.B. OpenShift, bieten diese Möglichkeit an. Es kann aber auch ein Microservice auf einer besseren Hardware laufen. Die anderen Services bleiben auf der billigeren Hardware. Monolithen müssten als Ganzes auf der besseren Hardware laufen. Dies verringert die Kosten der Skalierung bei Microservices drastisch [Newman, 2015].

Austauschbarkeit

Bei Microservices ist es sehr einfach, den bestehenden Code zu ändern oder auszutauschen. Da diese unabhängig sind, sind keine anderen Teile des Systems betroffen. Bei Monolithen stellt die Austauschbarkeit oft ein großes Risiko dar. Der Code ist oft sehr verwoben und kleine Änderungen haben große Auswirkungen [Newman, 2015].

Entwicklerteam

Entwickler greifen bei monolithischen Architekturen oft auf dieselben Teile des Codes zu. Bei Versionierungsprogrammen, wie Git oder Subversion, führt dies oft zu Konflikten. Microservices können getrennt voneinander bearbeitet und der Code ohne Probleme zusammengefügt werden [Newman, 2015].

2.5.2 Nachteile

Hohe Latenzzeiten

Microservices reden über das Netzwerk miteinander. Dieser Kommunikationsweg ist natürlich viel langsamer, als wenn diese in derselben JVM miteinander reden würden. Dies führt zu hohen Latenzzeiten. Diese müssen bei der Architektur von Microservices berücksichtigt werden [Newman, 2015].

Wiederverwendbarkeit von Code

Microservices sind unabhängige und eigenständige Einheiten. Sie haben im besten Fall keine Abhängigkeiten zueinander. Dadurch verringert sich die Wiederverwendbarkeit von Code. Dies führt auch zu einer Codeverdopplung. Wird dasselbe Codestück in zwei verschiedenen Services benötigt, muss es in beiden vorhanden sein [Newman, 2015].

Deployment

Jedes Service muss einzeln deployt werden. Dazu wird ein eigener Deployment-Prozess pro Service benötigt. Dies bedeutet Aufwand für den Entwickler und es muss eine eigene Infrastruktur geschaffen werden, was zusätzlich Kosten generiert [Newman, 2015].

Kapitel 3

Containerisierung mit Docker

In folgendem Kapitel wird Docker, die Bestandteile von Docker und der Unterschied zwischen Virtualisierung und Containerisierung erklärt. Anschließend wird auch die Notwendigkeit von Containerisierung für Cloud-Plattformen aufgezeigt.

3.1 Docker

Docker ist ein Container-Management-System, das es erleichtert, Linux Container zu managen. Docker erlaubt das Erstellen von Docker Images in virtuellen Umgebungen am lokalen PC. Auf diesen Umgebungen können Kommandos und Operationen ausgeführt werden. Diese Aktionen unterscheiden sich am lokalen PC nicht von den Aktionen am Produktivsystem. Dies hilft bei der Entwicklung lokal, sowie beim Deployment auf der Produktivumgebung am Server, da die auszuführenden Kommandos dieselben sind [Gallagher, 2015].

3.1.1 Komponenten von Docker

Im Folgenden werden die Kernkomponenten von Docker beschrieben.

Docker Engine

Docker ist eine Client-Server Anwendung. Der Docker-Server wird auch als *Docker Daemon* oder *Docker Engine* bezeichnet. Bei der Installation von Docker werden eine Command Line Binary und eine RESTful API mitgeliefert, um mit dem Docker Daemon zu interagieren. Der Docker Daemon und der Docker Client können am selben Host laufen. Der lokale Docker Client kann aber auch mit einem Remote Docker Daemon verbunden werden. Der Docker Daemon erzeugt und managt die Docker Container am Docker Host. Die Kommunikation des Docker Client mit dem Docker Daemon läuft über die von Docker bereitgestellte REST-API. [Turnbull, 2015]

Docker Images

Docker Images sind die Bausteine im Docker System. Docker Container werden aus Docker Images gestartet. Docker Images sind mehrschichtig und werden schrittweise aus einer Serie von Instruktionen erstellt. Beispiele für solche Instruktionen können sein:

- Eine Datei hinzufügen.

- Einen Befehl im Container ausführen.
- Einen Port öffnen.

Docker Images werden auch als „Source Code“ für Container bezeichnet. Docker Images sind sehr portabel und können einfach verbreitet, gespeichert und aktualisiert werden [Turnbull, 2015].

Die Instruktionen zum Bauen eines Docker Image werden in einem *Dockerfile* beschrieben. Kommandos, Aufrufe von Skripten, das Setzen von Environment-Variablen, das Hinzufügen von Dateien und das Setzen von Rechten können alle im Dockerfile beschrieben werden. Im Dockerfile muss auch das Basisimage für den Build spezifiziert werden. Ein Dockerfile könnte wie folgt aussehen [Gallagher, 2015]:

Listing 3.1: Dockerfile

```
1  FROM ubuntu:latest
2  MAINTAINER Scott P. Gallagher <email@somewhere.com>
3
4  RUN apt-get update && apt-get install -y apache2
5
6  ADD 000-default.conf /etc/apache2/sites-available
7  RUN chown root:root /etc/apache2/sites-available/000-default.conf
8
9  EXPOSE 80
10 CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Dies sind typische Befehle, die in den meisten Dockerfiles zu finden sind. Im *FROM*-Befehl wird das Basis-Image, auf dem das Docker Image aufgebaut ist, definiert. Jede weitere Zeile fügt eine neue Schicht im Docker Image hinzu. Mit dem Befehl:

```
1 $ docker build <DOCKERFILE>
```

kann dieses Docker Image gebaut werden. Aus diesem Docker Image kann danach ein Docker Container erzeugt werden [Gallagher, 2015].

Registries

Die gebauten Docker Images werden in Registries gespeichert. Dabei gibt es zwei verschiedene Arten von Registries: **public** und **private**. Docker, Inc., verwaltet die public Registry, auch Docker Hub genannt. In der public Registry befinden sich auch bereits vordefinierte Docker Images für zum Beispiel eine MySQL Datenbank oder einen Nginx Web Server. Es kann auch eine eigene private Registry erstellt werden. Diese erlaubt das Speichern von Images, ohne dass von außen auf diese zugegriffen werden kann [Turnbull, 2015].

Docker Container

In Docker Containern können Applikationen und Services laufen. Docker Container werden aus Docker Images erstellt und können einen oder mehrere Prozesse enthalten. Ein Docker Container besteht aus [Turnbull, 2015]:

- Einem Docker Image Format.
- Einem Set von Standardoperationen.
- Einer Umgebung zum Ausführen der Operationen.

Jeder Docker Container enthält ein Software Image, dass das Erstellen, Starten, Stoppen, Neustarten und Löschen des Docker Containers erlaubt. Docker kümmert sich dabei nicht um den Inhalt des Containers. Jeder Container, egal ob eine MySQL Datenbank oder ein Nginx Web Server, wird gleich gestartet oder gelöscht. Docker ist dabei unabhängig von der Umgebung in der der Docker Container läuft. Es kann lokal am PC gebaut werden, in die public Registry hochgeladen werden oder der Container in der Cloud laufen. Dadurch kann mit Docker schnell ein Application Server, ein Message Bus oder ein Continous-Integration Test erstellt und getestet werden [Turnbull, 2015].

3.1.2 Unterschied Virtualisierung und Containerisierung

Virtualisierung

Eine virtuelle Maschine, welche die Virtualisierung auf Hardware-Level repräsentiert, ist grundsätzlich ein komplettes Operating System, das in einem Host Operating System läuft. Dabei gibt es zwei Arten von „Virtualization Hypervisors“: **Type 1** und **Type 2**. Type 1 Hyperivsors bieten Server Virtualisierung auf Hardware-Level. Dabei gibt es kein traditionelles End-User Operating System. Type 2 Hypervisor werden zur Desktop Virtualisierung verwendet. Die Virtualisierungs-Engine läuft dabei auf einem eigenen Operating System. Der größte Vorteil von virtuellen Maschinen ist, dass mehrere unterschiedliche Operating Systeme auf einem Host laufen können [Krochmalski, 2016].

Virtuelle Maschinen sind voll isoliert und daher sehr sicher. Sie enthalten jedoch alle Bestandteile, die ein Operating System aufweisen muss: Treiber, Systembibliotheken, etc. Virtuelle Maschinen sind daher sehr schwergewichtig und ressourcenhungrig. Auch die Installation virtueller Maschinen ist sehr komplex und aufwändig. Um eine Applikation in einer virtuellen Maschine zu starten, muss der Hypervisor die virtuelle Maschine zuerst importieren und danach starten. Weiters können nur wenig virtuelle Maschinen auf einer einzelnen Maschine gestartet werden, da die Performance mit jeder neuen virtuellen Maschine drastisch abnimmt [Krochmalski, 2016].

Containerisierung

Die Abbildung 3.1 zeigt einen der größten Vorteile von Docker. Docker benötigt kein neues Operating System sobald ein neuer Container gestartet wird. Dies verringert die Größe der einzelnen Container drastisch. Docker baut dabei auf den Linux Kernel des Host OS auf. Dies macht es möglich jedes Linux OS als Host OS zu verwenden. Beispielsweise kann auf der linken App in der Abbildung auf der Docker-Seite Red Hat laufen und auf der anderen Debian. Dazu muss weder Red Hat, noch Debian am Host installiert sein. Docker Images werden dadurch so klein wie möglich gehalten. Sie sind dadurch sehr kompakt und einfach zu deployen [Gallagher, 2015].

Docker Container sind nicht nur vom darunterliegenden Operating System isoliert, sondern auch von anderen Docker Containern. Die Startzeiten von Docker Containern sind in der Regel durch den geringen Overhead sehr schnell. Docker Container ersetzen jedoch virtuelle Maschinen nicht ganz. Es muss vor der Entwicklung evaluiert werden, welches die beste Wahl ist. Auf der einen Seite gibt es völlig isolierte, sichere virtuelle Maschinen mit durchschnittlicher Performanz und auf der anderen Seite hoch performante, schnell deploybare Docker Container [Krochmalski, 2016].

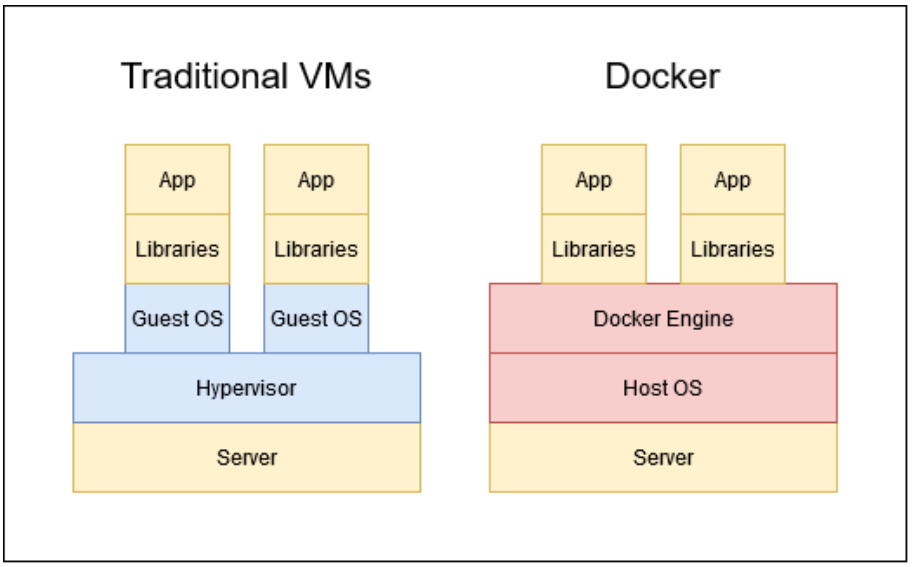

Abbildung 3.1: Unterschied Traditionelle VMs und Docker

[Gallagher, 2015]

3.1.3 Vorteile von Docker

In diesem Abschnitt werden die Vorteile von Docker und Containerisierung im Gegensatz zu virtuellen Maschinen und Virtualisierung beschrieben.

Schnelligkeit und Größe

Docker Container können schnell und einfach erstellt und wieder gelöscht werden. Docker teilt sich nur den Kernel mit dem Operating System. Auch Schichten von Docker Images können wiederverwendet werden. Dies führt zu sehr leichtgewichtigen Docker Containern. Die Resultate sind schnelles Deployment, einfache Migration und kurze Startzeiten [Krochmalski, 2016].

Reproduzierbarkeit und portable Builds

Docker erlaubt das Deployment von ready-to-use Software, welche portabel und sehr einfach zu verteilen ist. Die containerisierte Applikation läuft im Docker Container, daher wird keine Installation benötigt. Docker Images definieren alle Abhängigkeiten die das System benötigt. Dies verringert Fehler mit Versionskonflikten und Kompatibilität. Entwickler können dasselbe Docker Image, das in der Produktionsumgebung läuft, lokal testen. Dies verschnellert auch den Prozess der Continous Integration. Es gibt keine endlose Build-Test-Deploy-Zyklen. Docker stellt dabei sicher, dass sich die Applikationen in Developement-, Test- und Produktionsumgebungen ident verhalten [Krochmalski, 2016].

Die Portabilität ist eine der größten Stärken von Docker. Docker Container können fast überall deployt und gestartet werden: am lokalen PC, am weit entfernten Server, auf der privaten und öffentlichen Cloud. Nahezu alle Cloud Computing Provider unterstützen Docker als Containerisierungsplattform. Ein Docker Container, der z.B. auf einer Amazon EC2 Instanz läuft, kann sehr einfach auf einer Google Compute Engine Instanz deployt werden. Durch den zusätzlichen Level von Abstraktion funktioniert Docker sehr gut mit vielen unterschiedlichen Cloud Providern [Krochmalski, 2016].

Unveränderbare und agile Infrastruktur

Eine idempotente Codebasis speziell im Bereich Konfigurationsmanagement zu warten ist sehr komplex und zeitintensiv. Die Codebasis wächst und wird immer komplexer. Deshalb wird unveränderbare Infrastruktur immer wichtiger. Containerisierung hilft dabei ungemein. Durch die Verwendung von Containern wird der Prozess des Entwickelns und des Deployments vereinfacht. Docker benötigt wenig Konfigurationsmanagement. Die Applikationen werden durch einfaches Deployment und Redeployment gemanagt. Docker bietet durch vorgefertigte Docker Images den Großteil der Konfiguration. Diese vorgefertigten Docker Images können mittels einem Dockerfile erweitert werden. Docker folgt dabei dem Ansatz *Infrastructure as Code*, wo die Infrastruktur des Servers als Code vorliegt. Bei Docker ist dies das Dockerfile [Krochmalski, 2016].

Tools und APIs

Docker ist nicht nur ein Dockerfile Prozessor und eine Runtime Engine. Docker ist ein komplettes Paket mit einer Vielzahl an Tools und APIs, die die tägliche Arbeit von Entwicklern und DevOps unterstützen. Die Docker Toolbox ist ein Installer, um schnell und einfach eine Docker Umgebung auf dem eigenen Rechner zu installieren. Kinematic ist eine Desktop Entwicklerumgebung zur Verwendung von Docker auf Windows und Mac OS X Rechnern. Docker enthält auch eine Vielzahl an Kommandozeilentools zur Entwicklung von Applikationen mit Docker [Krochmalski, 2016].

3.2 Notwendigkeit von Containerisierung

Containerisierung ist gerade bei Cloud Plattformen, wie OpenShift sehr wichtig. Durch die Instanzierung eines Image wird ein Container erzeugt. In diesem läuft die Applikation. Applikationen, die in einem Container laufen, sind sicherer als Applikationen, die nicht in einem Container laufen. Container nutzen dabei die Sicherheit des Linux Kernel Namespace, um Applikationen, die am selben Rechner und unter denselben **control groups (cgroups)** laufen, zu sandboxen. Dabei wird dem Risiko, dass eine Applikation alle Ressourcen eines Servers nutzt, entgegen gewirkt. Einem Container kann eine maximale Verwendung von CPU-Ressourcen zugewiesen werden. Dadurch wird sichergestellt, dass alle Anwendungen genug Ressourcen bekommen [Schenker, 2019].

Dadurch, dass die Images unveränderbar sind, können sie auch nach möglichen Angriffsstellen und Veralterung überprüft werden. Weiters kann auch **content trust** verwendet werden. Dabei wird sichergestellt, dass der Ersteller des Image auch wirklich derjenige ist, der er vorgibt zu sein. Dadurch werden die Images auch gegen **man-in-the-middle**-Attacken gesichert, wo während dem Versenden das Image ohne Wissen des Empfängers geändert wird [Schenker, 2019].

Durch die Containerisierung kann auch einfach am PC des Entwicklers eine produktionsähnliche Umgebung geschaffen werden. Dadurch, dass jede Applikation containerisiert werden kann, können auch z.B. Datenbanken, wie Oracle oder MS SQL Server schnell für die lokale Entwicklung verwendet werden. Diese müssen dann nicht am PC des Entwicklers aufwändig installiert und konfiguriert werden [Schenker, 2019].

Container sind auch schneller und einfacher zu erstellen, zu deployen und zu managen als virtuelle Maschinen. Container benötigen auch weniger Rechenleistung. Daher ist es möglich mehrere Container am selben Rechner laufen zu lassen. Dies spart natürlich auch Kosten beim Hosten von Applikationen in der Cloud [Schenker, 2019].

Ein weiterer wichtiger Punkt ist das Aufsetzen und Hochfahren der gesamten Applikation von z.B. Projektleitern bei Präsentationen beim Kunden. Projektleiter sind oft bei der eigentlichen Entwicklung der Anwendung nicht involviert und haben deshalb die Infrastruktur am eigenen Rechner nicht eingerichtet. Durch die schnelle und einfache Verwendung von Docker ist es für Projektleiter um einiges einfacher die Software beim Kunden zu präsentieren [Schenker, 2019].

Kapitel 4

OpenShift

In Folgenden wird die Open Source Container Application Platform OpenShift und die dazugehörigen Komponenten beschrieben. OpenShift setzt auf Kubernetes auf. Deshalb wird auch Kubernetes in diesem Kapitel erklärt.

4.1 Kubernetes

Container bieten eine Vielzahl an Vorteilen bezüglich der Deploymentzeit, Skalierbarkeit und Größe gegenüber Virtualisierung. Container allein reichen jedoch nicht aus beim Managen von komplexen Cloud Systemen. Es ist sehr einfach ein paar wenige Container zu starten oder zu stoppen. Wächst die Zahl an Containern wird das manuelle Management sehr umständlich und aufwändig. **Container Orchestration Engines (COE)** helfen beim Management vieler Container. COE´s bieten Mechanismen zum schnellen Deployen, Zerstören und Skalieren von vielen Containern. Es gibt mehrere Container Management Lösungen. Die beiden bekanntesten sind Kubernetes und Docker Swarm. Da OpenShift jedoch auf Kubernetes aufbaut, wird in diesem Abschnitt lediglich Kubernetes behandelt. [Zuev, 2018].

Grundsätzlich gibt es in Kubernetes zwei verschiedene Arten von Nodes. Wie in Abbildung 4.1 zu sehen, sind dies der Kubernetes Master und mehrere Kubernetes Nodes.

4.1.1 Master

Die Master-Node ist für das Cluster-Management, die Netzwerk-Allokierung, die Synchronisation und die Kommunikation verantwortlich. Master-Nodes agieren als Hauptansprechspartner für die Clients. Im einfachsten Setup gibt es lediglich einen Master-Node. Hochverfügbare Cluster benötigen aber mindestens zwei Master-Nodes, um häufige Fehlersituationen zu vermeiden. Das wichtigste Service, das am Master läuft, ist die API [Zuev, 2018].

4.1.2 Nodes

Nodes erledigen die eigentliche Arbeit beim Hosten von Docker Containern. Nodes bieten dabei ein Runtime Environment in dem die Pods laufen. Auch das Kubelet-Service, das die Pods managt, läuft auf den Nodes.

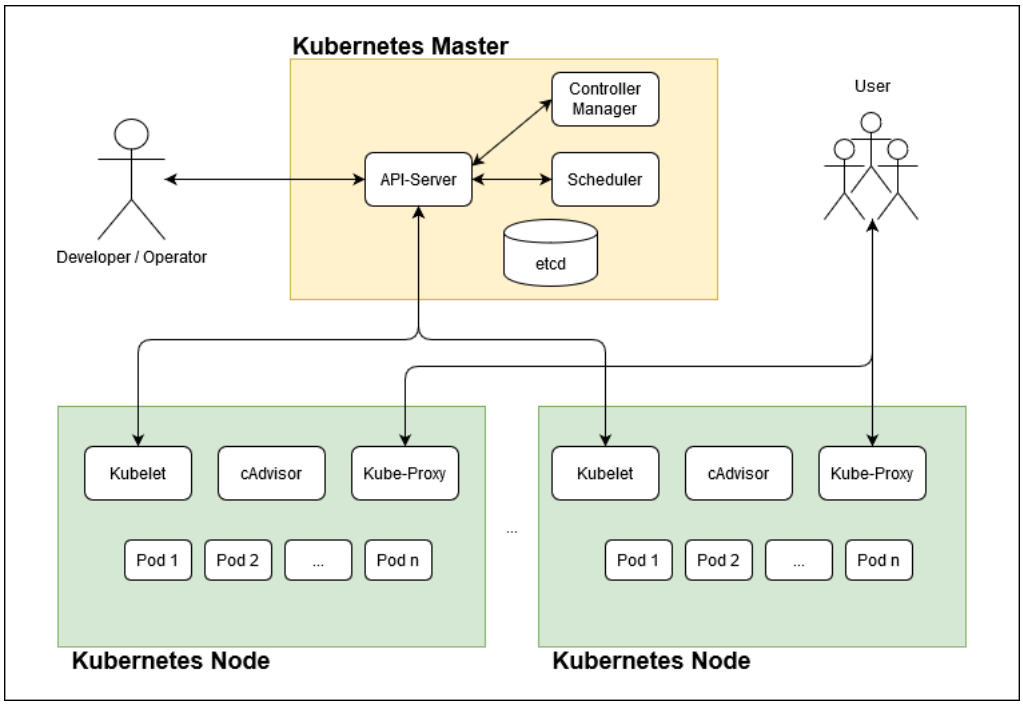

Abbildung 4.1: Kubernetes Architektur

[Zuev, 2018]

4.1.3 Kubernetes API

Die Kubernetes API bietet eine Vielzahl an Ressourcen, um Mechanismen von Kubernetes zu nutzen. Diese Ressourcen können in Form von YAML- oder JSON-Dateien definiert werden. Im Folgenden werden die wichtigsten Ressourcen beschrieben [Zuev, 2018]:

- **Namespaces**: Namespaces dienen zur Separierung von verschiedenen Projekten in der Kubernetes Umgebung. Sie dienen auch zur Definition von fein granularen Zugriffsrechten. Alle Kubernetes Ressourcen, außer Volumes und Namespaces selbst, leben in einem Namespace. Das heißt, dass deren Name im gegebenen Namespace eindeutig sein muss.
- **Pods**: Pods repräsentieren eine Sammlung von Containern. Jeder Pod ist für eine bestimmte Management Unit in Kubernetes zuständig. Alle Container in einem Pod teilen sich denselben Speicher, dieselben Volumes und dasselbe Netzwerk.
- **Services**: Services repräsentieren ein Interface zwischen Clients und der eigentlichen Applikation, die auf den Pods läuft. Ein Service ist ein Paar aus IP-Adresse und Port, das Netzwerkverkehr zu den Backend Pods im Round-Robin Verfahren weiterleitet. Dadurch dass Services ein konsistentes Paar aus IP-Adresse und Port sind, werden Clients vor vorübergehenden Änderungen im Cluster geschützt.
- **Replication Controller**: Replication Controller definieren wie viele Pods repliziert werden müssen. Die Definition von Replication Controller inkludiert Pod Templates, welche die Pods beschreiben. Ein Parameter der Replication Controller ist wie viele Pods mindestens aufrecht erhalten werden müssen. Stirbt ein Pod, fährt Kubernetes wieder so viele hoch, dass das Minimum erreicht ist.
- **Persistent Volumes**: Persistent Volumes abstrahieren physische Speichersysteme, wie NFS oder iSCSI. Persistent Volumes werden in der Regel von Cluster Administratoren

erstellt und können durch Persistent Volume Claims Binding Mechanismen in einem Pod gemountet werden.

- **Persistent Volume Claims**: Persistent Volume Claims repräsentieren eine Anfrage auf Speicherressourcen. Pod Definitionen verwenden Persistent Volume Claims nicht direkt. Pods vertrauen auf das Binding von Persistent Volumes zu Persistent Volume Claims.
- **Secrets**: Secrets werden zu Übertragung von sensitiven Daten, wie Schlüssel, Tokens oder Passwörtern innerhalb von Pods verwendet.
- **Labels**: Labels bieten einen Mechanismus zum Spezifizieren eines Gültigkeitsbereichs von Ressourcen. Services verwenden Labels, um den hereinkommenden Netzwerkverkehr zu den richtigen Pods weiterzuleiten. Werden neue Pods mit demselben Label gestartet, werden sie dynamisch dem Service, welches das Label spezifiziert, zugewiesen.

4.1.4 Kubernetes Master

Folgende Services laufen auf dem Kubernetes Master [Zuev, 2018]:

- **etcd**: Der etcd ist ein verteilter Key-Value Konfigurationsspeicher, der alle Metadaten und Cluster Ressourcen enthält. Zur Wahrung der Integrität, wird empfohlen immer eine ungerade Anzahl an etcd Nodes, beginnend bei drei bei einem hochverfügbaren Setup, laufen zu lassen.
- **Kube-Apiserver**: Der Kube-Apiserver ist ein Service, das die Kubernetes API nach außen frei gibt. Da dieses Service keinen Zustand benötigt, unterstützt es die horizontale Skalierung in hochverfügbaren Clusterkonfigurationen.
- **Kube-Scheduler**: Der Kube-Scheduler ist eine Komponente, die das Erstellen der neu gestarteten Pods überwacht. Dabei muss auf Hardware-Limitierungen, Lokalitätseigenschaften der Daten und Beziehungsregeln geachtet werden.
- **Kube-Controller-Manager**: Der Kube-Controller-Manager managt die Controller. Manche sind Replication Controller, die eine minimale Anzahl an Pods aufrecht erhalten müssen. Weiters werden auch Node Controller gemanagt. Diese überwachen den Zustand der Nodes. Auch Volume Controller, die für das Binding von Persistent Volumes zu Persistent Volume Claims zuständig sind und Endpoint Controller, die Services und Pods verbinden, werden vom Kube-Controller-Manager gemanagt.
- **Cloud-Controller-Manager**: Der Cloud-Controller-Manager bietet eine Integration zu darunterliegenden Cloud Providern, wie DigitalOcean und Oracle Cloud Infrastructure.

4.1.5 Kubernetes Nodes

Folgende Services laufen auf den Kubernetes Nodes [Zuev, 2018]:

- **Kubelet**: Dieses Service verwendet eine Pod Spezifikation, um seine Pods zu managen und periodische Health Checks durchzuführen.
- **Kubeproxy**: Diese Komponente implementiert eine Service Abstraktion durch anbieten von TCP und UDP Forwarding über ein Set von Backend Pods.
- **Container Runtime Environment**: Die Container Runtime lädt Images herunter und startet Container. Kubernetes unterstützt dabei Docker und rkt als Container Runtimes.

4.2 OpenShift

OpenShift setzt auf Kubernetes auf. Kubernetes bietet Möglichkeiten für die Orchestrierung von Containern, die Elastizität von Pods, Service Definitionen und Deployment Konstrukte, um den Status einer Microservice-basierten Applikation zu beschreiben. Es sind jedoch weitere Komponenten nötig, um eine Platform-as-a-Service Plattform zu managen. Kubernetes bietet zum Beispiel kein software-definiertes Netzwerk (SDN) oder Methoden, um den Netzwerkverkehr zu den laufenden Containern zu verteilen. All diese Methoden liefert OpenShift mit. Die OpenShift Plattform wurde im Mai 2011 gestartet. Der Source Code war als Open Source Projekt frei zugänglich. Red Hat bietet auch eine Enterprise Version für Deployments an. Dieser Service heißt OpenShift Online. OpenShift ist eine Plattform, die Entwicklern beim Deployment von Applikationen zu einem oder mehreren Hosts hilft. Diese können jegliche Art von Applikation sein, sei es eine Web Applikation oder eine Backend Applikation. Auch Microservices und Datenbanken können in OpenShift deployt werden. Die einzige Voraussetzung ist, dass diese Applikation in einem Container laufen kann. OpenShift kann auf jedem System laufen. Sei es eine öffentliche oder private Infrastruktur, direkt auf physischer Hardware oder einer virtuellen Maschine. OpenShift bietet dabei auch eine CLI und eine Web Console zur Interaktion mit OpenShift an [Dupleton, 2018].

4.2.1 OpenShift Master

Wie in Abbildung 4.2 zu sehen ist, managt der OpenShift Master die Kubernetes Master im OpenShift Cluster. Der OpenShift Master bietet dabei eine REST-Api für externe Clients zum Managen des OpenShift Masters an. OpenShift bietet dabei auch Features, die von Kubernetes nicht zur Verfügung gestellt werden. Dazu gehören ein rollen- und gruppenbasiertes Sicherheitsmodell und Controller zum Managen von OpenShift Objekten.

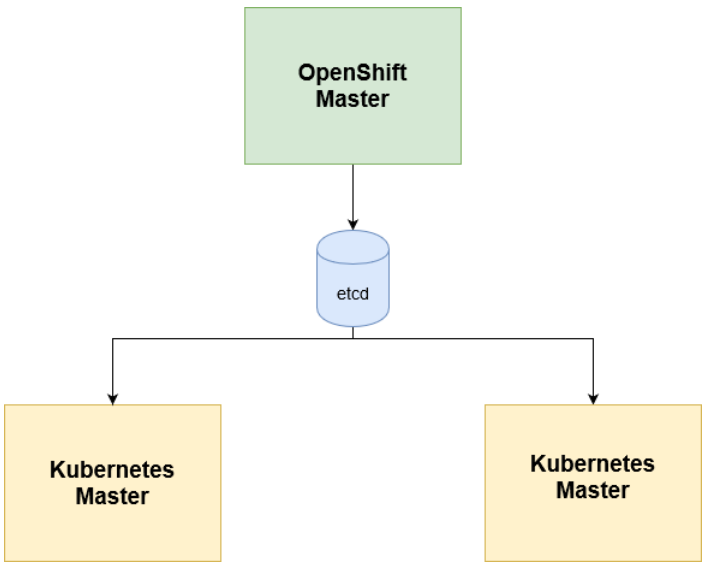

Abbildung 4.2: OpenShift Master

[RedHatInc, 2019]

Im Folgenden werden die Kernkonzepte und wichtigsten Objekte von OpenShift erklärt.

4.2.2 Container und Images

Container sind die Grundbestandteile von OpenShift. Linux Container sind leichtgewichtige Mechanismen zur Isolierung von Prozessen, sodass Prozesse in der Kommunikation miteinander und bei der Nutzung gemeinsamer Ressourcen eingeschränkt sind. Viele Applikationen können in Containern auf einem Host laufen, ohne dass sie auf die Prozesse, Dateien oder das Netzwerk anderer Container zugreifen können. OpenShift und Kubernetes ermöglichen dabei die Orchestrierung von Docker Containern über mulit-host Umgebungen hinweg [RedHatInc, 2019].

Container in OpenShift basieren auf Docker **Images**. Ein Image ist dabei ein Binary, das alle Anforderungen für das Erstellen und Starten eines Containers, sowie Metadaten, die Anforderungen der Container beschreiben, beinhaltet. Container haben nur Zugriff auf Ressourcen, die im Image definiert wurden, außer dem Container werden weitere Rechte für den Zugriff auf Dateien eingeräumt. Beim Deployment eines Image auf mehreren Containern über verschiedene Hosts hinweg, kann OpenShift Redundanzen und horizontale Skalierung bieten [RedHatInc, 2019].

4.2.3 Pods und Services

Ein **Pod** ist eine Einheit aus einem oder mehreren Containern auf einem Host und die kleinste Recheneinheit, die definiert, deployt und gemanagt werden kann. Jedem Pod wird eine interne IP-Adresse zugewiesen. Container innerhalb eines Pods können dadurch ihren lokalen Speicher und das Netzwerk teilen. Pods haben einen Lebenszyklus. Pods werden definiert und danach einer Node zugewiesen. Dann laufen sie solang, bis sie gelöscht werden oder aus anderen Gründen entfernt werden. Pods können nach dem Stoppen entweder gelöscht oder behalten werden. Dadurch kann auch nach Absturz eines Pods auf die Log-Files zugegriffen werden [RedHatInc, 2019]. OpenShift behandelt Pods als wären sie unveränderbar. Änderungen an laufenden Pods können nicht vorgenommen werden. OpenShift implementiert Änderungen an Pods durch terminieren des Pods und starten eines neuen geänderten Pods. Ein Pod behält dabei nicht seinen Zustand [RedHatInc, 2019].

Ein Kubernetes **Service** dient als interner Load Balancer. Es leitet Verbindungen, die es bekommt, an ein Set von replizierten Pods weiter. Pods können dabei zum Service hinzugefügt und entfernt werden. OpenShift vergibt Default Service Cluster IP-Adressen. Dadurch wird Pods die Kommunikation mit anderen Pods ermöglicht. Um auch externen Zugriff auf die Services zu ermöglichen, können dem Service auch *externalIP*- und *ingressIP*-Adressen zugewiesen werden. Diese externalIP-Adressen können auch virtuelle IP-Adressen sein, die hochverfügbaren Zugriff auf das Service ermöglichen. Services wird ein Paar aus IP-Adresse und Port zugewiesen. Falls darauf zugegriffen wird, wird die Verbindung auf einen passenden Pod weitergeleitet. Ein Service verwendet dabei einen Label Selector, um alle laufenden Container, die ein spezifisches Netzwerk-Service auf einem spezifischen Port laufen lassen, zu finden [RedHatInc, 2019].

4.2.4 User, Namespaces und Projekte

Die Interaktion mit OpenShift ist mit **Usern** assoziiert. Ein OpenShift *User*-Objekt repräsentiert einen Aktor, dem verschiedene Rechte im System durch das Hinzufügen von Rollen zu den Usern oder deren Gruppen eingeräumt werden [RedHatInc, 2019]. Dabei gibt es grundsätzlich drei verschiedene Arten von User Objekten [RedHatInc, 2019]:

1. **Regular User**: Die meisten User in OpenShift gehören zu dieser Gruppe. Regular User werden automatisch beim ersten Login vom System erstellt oder können durch die API selbst erstellt werden. Regular User werden durch das User-Objekt repräsentiert. Beispiele: *joe*, *alice*
2. **System User**: Viele der System User werden automatisch erstellt, sobald die Infrastruktur definiert wurde. Hauptsächlich für die sichere Interaktion mit der API. Dazu gehören ein Cluster Administrator, der Zugriff auf alle Komponenten hat, ein per-node User und User für die Nutzung von Router und Registries. Es gibt auch einen *anonymous* System User, der bei unautorisierten Requests verwendet wird. Beispiele: *system:admin*, *system:openshift-registry*, *system:node:node1.example.com*
3. **Service Accounts**: Service Accounts sind spezielle System User, die mit einem Projekt assoziiert sind. Manche werden beim Erstellen eines Projekts automatisch erstellt. Projekt Administratoren können mehr Rechte für den Zugriff auf das Projekt definieren. Service Accounts werden durch das ServiceAccount-Objekt repräsentiert. Beispiele: *system:serviceaccount:default:deployer*, *system:serviceaccount:foo:builder*

Jeder User muss sich vor der Nutzung von OpenShift authentifizieren. API Requests ohne Authentifizierung oder einer invaliden Authentifizierung werden mit dem *anonymous*-User durchgeführt. Sobald der User authentifiziert ist, definiert eine Policy, welche Aktionen der User durchführen darf [RedHatInc, 2019].

Ein Kubernetes **Namespace** bietet einen Mechanismus zur Isolierung von Ressourcen in einem Cluster. Namespaces bieten einen Isolationsbereich für [RedHatInc, 2019]:

- Benannte Ressourcen, um Namenskollisionen zu vermeiden.
- Management Authorities für User.
- Die Möglichkeit Ressourcennutzung zu limitieren.

Die meisten Objekte im System sind durch Namespaces isoliert. Nodes und User unterliegen jedoch keinem Namespace.

Ein **Projekt** ist ein Kubernetes Namespace mit zusätzlichen Annotationen. In Projekten wird auch der Zugriff von Usern auf Ressourcen gemanagt. Ein Projekt erlaubt einer Menge von Usern deren Ressourcen von anderen Usern getrennt zu organisieren und zu managen. User müssen die Rechte für Projekte von Administratoren zugewiesen bekommen, außer ein User erstellt selbst ein Projekt. Dann hat er auf dieses Projekt automatisch Zugriff. Projekte können separate *Namne*, *Anzeigenamen* und *Beschreibungen* haben.

- Der zwingend notwendige **Name** ist ein eindeutiger Indikator für das Projekt und ist sichtbar bei der Verwendung der CLI Tools oder der API. Die maximale Länge des Namens beträgt 63 Zeichen.
- Der **Anzeigename** zeigt, wie das Projekt in der Web Console dargestellt wird. Ist dieser Wert nicht vorhanden, wird der eindeutige Name angezeigt.
- Die optionale **Beschreibung** ist eine detaillierte Beschreibung des Projekts und ist auch in der Web Console sichtbar.

Jedes Projekt isoliert ein Set von [RedHatInc, 2019]:

- **Objekten**: Pods, Services, Replication Controller, etc.
- **Policies**: Regeln für User zur Interaktion mit Objekten.
- **Constraints**: Teile von Objekten, die limitiert werden können.
- **Service Accounts**: Bieten einen flexiblen Weg, um den Zugriff auf die API zu kontrollieren.

Cluster Administratoren können Projekte erstellen und administrative Rechte für Projekte vergeben. Cluster Administratoren können auch Entwickler das Erstellen von Projekten erlauben. Entwickler und Administratoren können durch die CLI oder die Web Console mit Projekten interagieren.

4.2.5 Builds und Image Streams

Ein **Build** ist der Prozess der Transformation von Input Parametern in Objekte. Meistens werden Builds zur Transformation von Input Parametern oder Source Code in ausführbare Images genutzt. Ein BuildConfig-Objekt beschreibt dabei den Build Prozess [RedHatInc, 2019].

OpenShift bietet dabei erweiterbaren Support für Build Strategien. Es stehen vier Build Strategien zur Verfügung [RedHatInc, 2019]:

1. **Docker Build**: Die Docker Build Strategie führt einen Docker Build aus und benötigt dadurch ein Repository mit einem Dockerfile und allen Artifakten, um das Docker Image zu erstellen.
2. **Source-to-Image (S2I) Build**: S2I ist ein Tool zum Bauen von reproduzierbaren Docker Containern und Docker Images. Dadurch werden startfähige Docker Images durch injizieren des Source Code der Applikation erstellt. Das neue Docker Image übernimmt das Basis Image, auch Builder Image genannt, und die gebauten Sourcen. Auf das neue Docker Image kann dann ein *docker run* ausgeführt werden. S2I unterstützt inkrementelle Builds, welche vorher heruntergeladene Abhängigkeiten wiederverwenden.
3. **Custom Build**: Die Custom Build Strategie erlaubt Entwickler spezifische Builder Images zu definieren. Dadurch können die Build Prozesse angepasst werden. Ein Custom Builder Image ist ein Docker Container Image eingebettet in Build-Prozess-Logik, z.B. zum Erstellen von Basis Images. Als Beispiel für ein Custom Builder Image steht auf der Seite von Docker Hub das Image *openshift/origin-custom-docker-builder* zur Verfügung.
4. **Pipeline Build**: Die Pipeline Build Strategie erlaubt Entwickler eine Jenkins Pipeline zu erstellen. Dieser Build kann ebenso wie die anderen Build Strategien gestartet, gemonitort und gemanagt werden. Pipeline Workflows werden in einem Jenkinsfile definiert - entweder eingebettet in der Build Konfiguration oder eigens im Git Repository definiert. Bei der ersten Definition einer Build Konfiguration mit der Pipeline Strategie instanziert OpenShift einen Jenkins Server, der die Pipeline ausführt. Darauffolgende Pipeline Build Konfigurationen nutzen diesen Jenkins Server.

Image Streams und die dazugehörigen Image Stream Tags bieten eine Abstraktion des Docker Images. Die Image Streams und Image Stream Tags erlauben das Anzeigen der verfügbaren Images und stellen sicher, dass das richtige Image verwendet wird, auch wenn sich das Repository ändert. Image Streams enthalten keine Image Daten, repräsentieren aber eine virtuelle Sicht auf die zusammenhängenden Images, ähnlich zu einem Image Repository [RedHatInc, 2019].

Es können auch Builds und Deployments konfiguriert werden, um Notifizierungen zu bekommen, sobald neue Images hinzugefügt werden. Wenn zum Beispiel ein Deployment ein spezifisches Image benutzt und eine neue Version dieses Image erstellt wird, kann automatisch ein neues Deployment mit der neuen Version getriggert werden. Solange der Image Stream Tag nicht upgedatet wird, benutzt das Deployment dieselbe Version des Image, auch wenn eine neue Version verfügbar wäre [RedHatInc, 2019].

4.2.6 Replication Controller, Jobs und Deployments

Replication Controller stellen sicher, dass eine in der Konfiguration spezifizierte Anzahl an Replicas eines Pods immer laufen. Falls ein Pod abstürzt oder gelöscht wird, instanziert der Replication Controller so viele bis das Minimum wieder erreicht ist [RedHatInc, 2019]. Die Konfiguration eines Replication Controller besteht aus [RedHatInc, 2019]:

- Der gewünschten Anzahl an Replicas. Dieser Wert kann auch zur Laufzeit geändert werden.
- Einer Pod Definition zum Erstellen eines Pods
- Einem Selector zum Identifizieren der gemanagten Pods

Ein Selector ist dabei ein Set von Labels, die einem Pod zugewiesen werden. Die Labels sind in der Pod Definition enthalten. Durch die Selectors bestimmt der Replication Controller die Anzahl an Pods, die gerade laufen und fügt bei Bedarf neue hinzu oder löscht welche. Der Replication Controller übernimmt nicht die Autoskalierung basierend auf der Load oder dem Traffic. Dazu muss ein externer Auto-Scaler definiert werden [RedHatInc, 2019].

Ein **Job** verhält sich ähnlich zu einem Replication Controller. Auch ein Job erstellt Pods für spezielle Zwecke. Der Unterschied zum Replication Controller ist, dass Replication Controller designt sind für Pods die durchgehend laufen und Jobs lediglich für Pods zuständig sind, die einen gewissen Arbeitsauftrag erledigen und danach wieder herunterfahren. Der Job überwacht die erfolgreichen Fertigstellungen und fährt herunter sobald alle von ihm überwachten Pods erfolgreich fertiggestellt sind.

OpenShift bietet durch **Deployments** und Deployment Configurations erweiterten Support für die Softwareentwicklung und den Deployment Lifecycle. Im einfachsten Fall erstellt ein Deployment lediglich einen neuen Replication Controller und lässt diesen Pods starten. Deployments bieten auch die Möglichkeit aus bestehenden Deployments aus Images neue Deployments zu erstellen. Auch Hooks, die vor oder nach dem Erstellen eines Replication Controllers laufen, können definiert werden [RedHatInc, 2019]. Ein DeploymentConfig-Objekt definiert dabei folgende Details [RedHatInc, 2019]:

- Die Definition eines Replication Controllers.
- Trigger, um neue Deployments automatisch zu erstellen.
- Eine Strategie zum wechseln zwischen Deployments.
- Life-Cycle-Hooks.

Jedes Mal wenn ein Deployment getriggert wird, egal ob automatisch oder manuell, managt ein Deployer Pod das Deployment. Dazu gehören auch das Herunterskalieren des alten Replication Controller, das Hochskalieren des neuen Replication Controller und das Ausführen der Hooks. Der Deployment Pod bleibt nach Fertigstellung auf unbestimmte Zeit am Leben, um die Logs des Deployments aufrufen zu können [RedHatInc, 2019].

4.2.7 Routes

OpenShift gibt einen Service durch **Routes** nach außen frei, sodass externe Clients ihn erreichen können. DNS Auflösung für den Hostnamen wird separat übernommen. Jede Route besteht aus einem Namen, limitiert durch 63 Zeichen, einem Selector und einer optionalen Sicherheitskonfiguration. Administratoren können Router zu Nodes deployen. Dadurch können externe Clients die Routen, die von Entwicklern erstellt wurden, aufrufen. Ein Router benutzt dabei einen Service Selector, um die Services und die Endpoints zu finden. Wenn beide, Router

und Service, Load Balancing erlauben, wird das Load Balancing automatisch eingerichtet [RedHatInc, 2019].

4.2.8 Templates

Templates beschreiben ein Set von Objekten, die parametrisiert und prozessiert werden können, um eine Liste von Objekten zu erstellen. Ein Template kann jedes Objekt zu dessen Erstellung der User die Rechte besitzt, beschreiben. Dazu gehören Service, Build Konfigurationen und auch Deployment Konfigurationen. Ein Template kann auch ein Set von Labels definieren, die dann auf die erzeugten Objekte angewandt werden. Eine Liste von Objekten durch ein Template kann über die CLI oder die Web Console erstellt werden [RedHatInc, 2019].

Kapitel 5

Partnerdatenbank

Im Folgenden wird der Grundaufbau und Zweck der Partnerdatenbank, sowie die Funktionsweise des Backends und des Frontends beschrieben.

5.1 Grundaufbau und Zweck der Partnerdatenbank

Die 3 Banken IT GmbH verwaltet ihre Partner in verschiedenen Systemen. Dazu gehören die Art28_Verträge_Fernwartungszugänge_VPE-Liste, in der die Angebotsanfragen und die Ausschreibungen dokumentiert sind und die FiPe-Liste, in der Kontaktpartner mit Kürzel, Name und laufender Nummer erfasst sind. Beide Listen sollen durch die Partnerdatenbank abgelöst werden.

Die Partnerdatenbank ist ein Prototyp für die 3 Banken IT GmbH, um zukünftig ihre Partner, sowie ihre Anwendungsfälle besser zu managen. In diesem Prototyp werden die Anwendungsfälle, die in der Abbildung 5.1 zu sehen sind, **Unternehmen anlegen**, **Kontaktperson anlegen**, **Unternehmen anzeigen** und **Kontaktperson anzeigen** implementiert. Grundsätzlich kann jeder Mitarbeiter diese vier Geschäftsfälle erledigen, sofern dieser eingeloggt ist.

Dieser Login wird rollenbasiert implementiert. Das bedeutet, dass in Zukunft Geschäftsfälle auch nur von bestimmten Usern mit bestimmten Rollen durchgeführt werden können. Alle REST-Aufrufe, außer der Login, können lediglich als eingeloggter User aufgerufen werden.

Die Partnerdatenbank dient auch als Anreiz und Motivation für die 3 Banken IT GmbH zukünftig Anwendungen mit Microservice-Architekturen zu entwickeln. Diese Form der Architektur bietet, wie bereits beschrieben, sehr viele Vorteile bei der Abwicklung von Geschäftsprozessen.

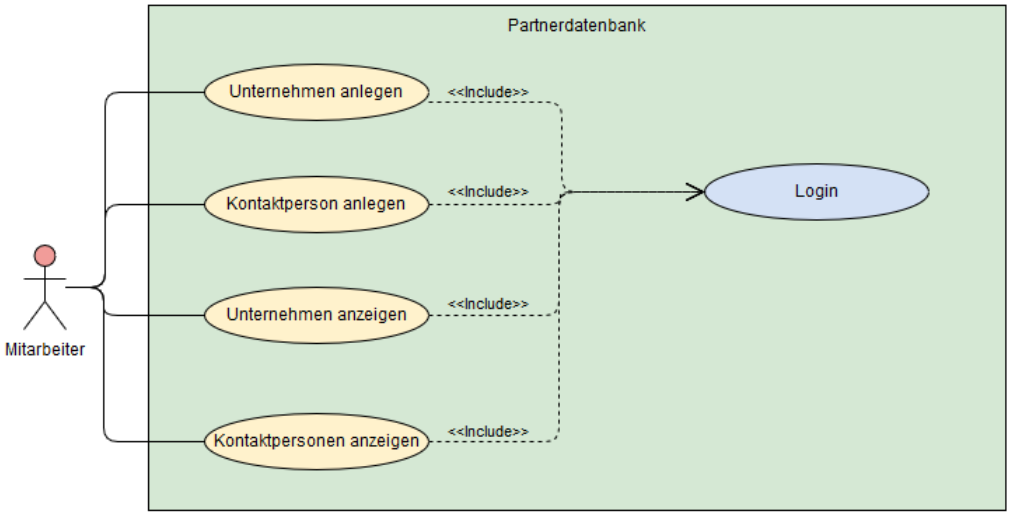

Abbildung 5.1: Anwendungsfalldiagramm der Partnerdatenbank

5.1.1 Lifecycle der Daten

Weitere Geschäftsprozesse, die die Daten der 3 Banken IT GmbH durchlaufen und in die Partnerdatenbank integriert werden, sind:

1. Erstkontakt
2. Angebotsanfrage/Ausschreibung
3. Angebotslegung
4. Verpflichtungserklärung
5. Auftrag
6. Lieferung
7. Rechnung
8. Installation
9. Inbetriebnahme
10. Kündigung
11. Deaktivierung
12. Deinstallation

Diese werden jedoch nicht vom Autor dieser Anwendung integriert, sondern von der 3 Banken IT GmbH selbst. Diese Arbeit dient lediglich als Vorlage und Schablone für die 3 Banken IT GmbH zur Integration weiterer Services bzw. Geschäftsfälle.

5.2 Beschreibung der Datenbank

Die Datenbanken für die 3 Banken IT GmbH bestehen aus Microsoft SQL Server Datenbanken. Ein Microsoft SQL Server ist ein relationales Datenbankmanagementsystem (DBMS), das von Microsoft entwickelt wurde. Dieses DBMS wurde von der 3 Banken IT GmbH vorgegeben. Der Autor beschreibt deshalb die Charakteristiken und Vor- und Nachteile dieses DBMS nicht genauer.

5.2.1 Lokal

Die Datenbank wird lokal über folgendes Dockerfile definiert:

```
1  FROM mcr.microsoft.com/mssql/server:2017-latest
2  EXPOSE 1433
3  COPY ./create-db.sql .
4  USER root
5  ENV MSSQL_DEVELOPER=sa
6  ENV MSSQL_SA_PASSWORD=Ruhsi@1234
7  ENV ACCEPT_EULA=Y
8  RUN chgrp -R 0 /var/opt && \
9    chmod -R g=u /var/opt && \
10   chown -R 10001:0 /var/opt && \
11   ( /opt/mssql/bin/sqlservr --accept-eula & ) | \
12     grep -q "Service Broker manager has started" && \
13   /opt/mssql-tools/bin/sqlcmd -S localhost -U \
14     sa -P Ruhsi@1234 -d master -i create-db.sql
```

Im Folgenden werden die Befehle des Dockerfile beschrieben:

1. Der **FROM**-Befehl definiert das Basis Image, das bereits einen vollständigen MS SQL Server beinhaltet. Dieses Image ist in der Registry von Microsoft zu finden und kann einfach verwendet werden.
2. Der **EXPOSE**-Befehl gibt den Port 1433 des Containers nach außen frei, sodass auch externe Clients auf dieses Service Zugriff haben.
3. Der **COPY**-Befehl kopiert die *create-db.sql*-Datei in den Container. Diese Datei muss unter dem angegebenen Pfad verfügbar sein und enthält in diesem Fall lediglich zwei Zeilen:

   Listing 5.1: create-db.sql

   ```
   1  CREATE DATABASE PartnerDB;
   2  CREATE DATABASE DocsisDB;
   ```

   Zur Speicherung der Partner, sowie Unternehmen ist die *PartnerDB* vorgesehen. Für die Speicherung der Links der Partner zu den entsprechenden Dokumenten ist die *DocsisDB* vorgesehen. Die Dokumente zu den Partnern kommen aus dem Doxis-System der 3 Banken IT GmbH. Da der Autor auf dieses System keinen Zugriff hat, wird beschlossen, dass die Daten in einer eigenständigen Datenbank gemockt werden.
4. Der **USER**-Befehl setzt den User mit dem die darauffolgenden Befehle ausgeführt werden. In diesem Fall wird der root-User verwendet, da laut Dokumentation dieses Basis Image, nur mit Root-Rechten ausgeführt werden kann.
5. Der **ENV**-Befehl setzt Environment-Variablen. Die beiden Environment-Variablen *MSSQL_DEVELOPER* und *MSSQL_SA_PASSWORD* setzen die Login-Parameter, mit denen auf den SQL Service zugegriffen werden kann. Mit der Environment-Variable *ACCEPT_EULA* wird das „End-User License Agreement“akzeptiert.
6. Im **RUN**-Befehl werden gleich mehrere Schritte ausgeführt. Die Befehle in Zeile 8, 9 und 10 setzen den Besitzer der Dateien und Unterordner des angegebenen Pfades und gewähren ihm Zugriff auf alle Dateien darin. Dadurch können die Befehle in Zeile 11 und 12 mit dem root-User ausgeführt werden.
7. Der Befehl in Zeile 11 startet den MS SQL Server und akzeptiert das EULA. Danach wird mit dem grep-Befehl solange gewartet bis der SQL Server hochgefahren ist und in den Logs die Ausgabe „Service Broker manager has started“ zu sehen ist.

8. In Zeile 12 verbindet sich der User mithilfe des sqlcmd-Tools und den angegebenen Parametern zum MS SQL Server und führt das oben angeführte *create-db.sql*-Skript aus.

Dieses Dockerfile kann mit folgendem Befehl gebaut werden:

```
$ docker build -t docsisdb .
```

Der Parameter *-t docsisdb* vergibt einen Namen für das Image. Der Punkt am Ende des Befehls ist der Pfad zum Dockerfile, in diesem Fall wird der Befehl im selben Verzeichnis, wie das Dockerfile, ausgeführt.

Mit dem Befehl:

```
$ docker run -d -p 1434:1433 --name docsisdb docsisdb
```

wird ein Docker Container aus dem Docker Image gestartet. Die Option *-d* startet den Container im Hintergrund (detached). Die Option *-p 1434:1433* leitet alle Aufrufe des Ports 1434 auf den container-internen Port 1433 weiter. Auf diesem Port ist der MS SQL Server erreichbar. Mit der Option *--name docsisdb* wird dem Docker Container der Name „docsisdb“ zugewiesen. Dadurch startet der Docker Container und der MS SQL Server ist unter dem Port 1434 des Hosts verfügbar.

Zum vereinfachten Ausführen mehrerer Docker Container ist es bei Docker auch möglich ein *docker-compose.yml* zu definieren, in dem mehrere Services definiert und ausgeführt werden. In folgender Datei wird das docker-compose.yml zum Ausführen der beiden Dockerfiles der docsis-database und der partner-database beschrieben:

Listing 5.2: docker-compose.yml

```
version: '3'
services:
  partner-database:
    build: ./partner-database
    container_name: "partner-database"
    ports:
      - 1433:1433

  docsis-database:
    build: ./docsis-database
    container_name: "docsis-database"
    ports:
      - 1434:1433
```

Im **build** wird der Pfad zu den jeweiligen Dockerfiles gesetzt. Mit **container_name** kann der Name des Containers gesetzt werden und im **ports**-Abschnitt werden die Ports freigegeben.

Mit dem Befehl:

```
$ docker-compose up
```

werden nun beide Docker Container gestartet und sind unter den jeweiligen Ports aufrufbar.

Die beiden Docker Container können mit folgendem Befehl wieder beendet werden:

```
$ docker-compose down
```

5.2.2 OpenShift

Das Dockerfile und die create-db.sql-Datei sehen beim Deployment der Datenbank in OpenShift gleich aus. Sie verändern sich nicht. Dadurch ist sichergestellt, dass sich die Umgebungen lokal und in OpenShift gleich verhalten. Das Deployment gestaltet sich jedoch etwas aufwändiger als lediglich ein „docker run“-Befehl. Da es für MS SQL Server kein vordefiniertes Template in OpenShift gibt, muss das Docker Image in die OpenShift interne Docker Registry gepusht.

Folgende Schritte müssen dafür unternommen werden:

1. **Freigeben der Route**: Die OpenShift interne Docker Registry muss nach außen frei gegeben werden, um diese von der CLI aus aufzurufen. Dies kann in der Web Console von OpenShift von einem Administrator erledigt werden.
2. **Holen der Route**: Mit dem Befehl:

   ```
   $ oc get route
   ```

   kann die Route der Docker Registry geholt werden.
3. **Einloggen in die Docker Registry**: Mit dem Befehl:

   ```
   $ docker login -u <openshift_username> -p <openshift_password>
     <registry_route>
   ```

   erfolgt der Login in die die OpenShift interne Docker Registry.
4. **Bauen des Docker Images lokal**: Mit folgendem Befehl wird das Docker Image mit dem Namen mssql versehen und gebaut:

   ```
   $ docker build -t mssql .
   ```

5. **Taggen des Images in der Registry**: Mit dem Befehl:

   ```
   $ docker tag mssql <registry_route>/<namespace>/mssql
   ```

   wird ein Tag in der OpenShift internen Registry erstellt. Dadurch wird eine Referenz des lokalen Docker Image zum OpenShift internen Docker Image erstellt.
6. **Pushen des Images in die Registry**: Durch den Befehl:

   ```
   $ docker push <registry_route>/<namespace>/mssql
   ```

   wird das Docker Image in die OpenShift interne Registry gepusht. Dadurch kann das Docker Image in OpenShift referenziert und ausgeführt werden.
7. **Ausführen des Image**: Mit folgendem Befehl kann das Docker Image in OpenShift gebaut werden:

   ```
   $ oc new-app mssql
   ```

   Dadurch wird der Docker Container in OpenShift ausgeführt und kann intern durch die Route *mssql:1433* erreicht werden. Muss die App auch von extern erreicht werden, muss die Route vorher freigegeben werden. Dies kann einfach über die Web Console erreicht werden.

5.3 Backend-Beschreibung

In diesem Abschnitt wird der grundsätzliche Aufbau einer Backend-Applikation beschrieben.

5.3.1 Paketstruktur

In Abbildung 5.2 ist die Paketstruktur des *partner-service* zu sehen.

Im Folgenden werden die wichtigsten Pakete näher beschrieben:

- **src.main**
  - **fabric8**:
    - Im *fabric8*-Ordner befinden sich das **deployment.yml** und das **route.yml**, die für das Deployment in OpenShift notwendig sind. Diese werden in einem späteren Abschnitt näher beschrieben.
  - **java.at.fh.se.master.partner**
    - **aspects**: In diesem Paket befindet sich die Klasse *LoggingAspect.java*, die für das Logging jeder Methode im Paket **rest** zuständig ist.
    - **configuration**: In diesem Paket befindet sich die Klasse *SimpleCORSFilter.java*. Diese setzt die Origin-Header, damit das Frontend auf die Ressourcen zugreifen kann.
    - **rest**: Im rest-Paket befinden sich die Interfaces und Implementierungen der REST-Api. Auch die Repositories für den Zugriff auf die Datenbank und eine *RestTemplateProvider*-Klasse, die das RestTemplate für REST-Aufrufe zum Docsis-Service zu Verfügung stellt, sind in diesem Paket enthalten.
    - **security**: In diesem Paket sind die Sicherheitskonfigurationen für den Login und die weiteren authentifizierten Aufrufe enthalten. Zudem sind auch noch Modelklassen für den User und die Rolle enthalten.
    - **service**: In diesem Paket ist ein Service für die Interaktion mit dem Docsis-Service enthalten. Damit können alle Links von einem Partner geholt, hinzugefügt und gelöscht werden.
  - **resources**
    - In diesem Paket ist das *application.yml* enthalten, das zur Konfiguration der Applikation benötigt wird. Dieses wird später näher erläutert.
- **src.test**
  - **java.at.fh.se.master.partner**
    - **rest.controller**:
      - In diesem Paket befinden sich die Tests für die Controller-Klassen.

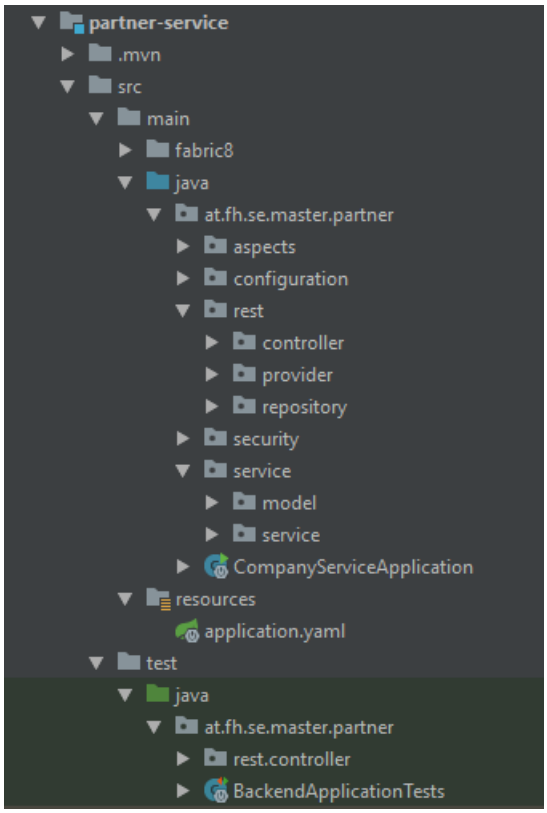

Abbildung 5.2: Paketstruktur des Partnerservice

5.3.2 Logging

Das Logging wird mithilfe von aspektorientierter Programmierung gelöst. Aspektorientierte Programmierung ist ein Paradigma, das darauf abzielt die Modularität von Applikationen zu erhöhen. Dadurch wird zusätzliches Verhalten zum Code hinzugefügt, ohne den eigentlichen Code verändern zu müssen. Der neue Code und das zusätzliche Verhalten werden getrennt voneinander deklariert.

In diesem Service wird aspektorientierte Programmierung zum Loggen der Ausführung von REST-Methoden genutzt. Dazu wird folgende Maven-Dependency benötigt:

```
1  <dependency>
2    <groupId>org.springframework.boot</groupId>
3    <artifactId>spring-boot-starter-aop</artifactId>
4  </dependency>
```

Die Deklaration des Aspekts sieht wie folgt aus:

Listing 5.3: LoggingAspect.java

```
1  @Aspect
2  @Component
3  public class LoggingAspect {
4    @Around("execution(public * at.fh.se.master.partner.rest..*(..))")
5    public Object profileAllMethods(ProceedingJoinPoint proceedingJoinPoint);
6  }
```

Die Klasse wird mit der Annotation *@Aspect* als Aspekt gekennzeichnet. In der Klasse selbst wird ein Around-Advice deklariert. Dadurch werden alle public-Methoden mit jeglichem Rückgabeparameter im Verzeichnis *at.fh.se.master.partner.rest* geloggt. Statt dem eigentlichen Aufruf der Methode wird der Aspekt aufgerufen. In diesem erfolgt das Logging. Danach muss die eigentliche Methode ausgeführt werden. Über den Parameter *ProceedingJoinPoint proceedingJoinPoint* können Informationen über die Methodensignatur geholt werden. Über diesen Parameter kann auch die eigentliche Methode aufgerufen werden. Der Rückgabewert der eigentlichen Methode wird auch im Advice zurückgegeben.

5.3.3 CORS-Filter

Um einen Cross-Origin Resource Sharing (CORS) Fehler beim Aufruf vom Frontend zu vermeiden, muss ein CORS-Filter implementiert werden. Dazu kann einfach das Interface *Filter* implementiert werden. Danach muss die Methode *doFilter* überschrieben werden. In dieser können die entsprechenden Header gesetzt werden.

In der Implementierung sieht dies wie folgt aus:

Listing 5.4: SimpleCORSFilter.java

```
1  @Component
2  public class SimpleCORSFilter implements Filter {
3    @Override
4    public void doFilter(ServletRequest req, ServletResponse resp,
5    FilterChain chain) throws IOException, ServletException {
6      HttpServletResponse response = (HttpServletResponse) resp;
7      HttpServletRequest request = (HttpServletRequest) req;
8      response.setHeader("Access-Control-Allow-Origin", request.getHeader("Origin"));
9       ...
10   }
11 }
```

Durch die beiden Parameter *ServletRequest req* und *ServletResponse resp* kann auf Informationen des Http-Requests und der Http-Response zugegriffen werden. In diesem Prototyp wird der Header *Access-Control-Allow-Origin* der Response mit dem Request-Header *Origin* gesetzt. Dies erlaubt jeder Domain Zugriff auf diesen Server. Dies muss von der 3 Banken IT GmbH natürlich auf die entsprechende Domäne, in der das Frontend in der Produktivumgebung läuft, geändert werden.

5.3.4 Controller

Mit Controllern kann eine REST-Api bereitgestellt werden. Mit *@RequestMapping* wird der grundsätzliche Pfad zu diesem Controller gesetzt. Die Annotation *@CrossOrigin* ist ein Mechanismus der von Spring Boot bereitgestellt wird und beim Handling von CORS unterstützt. Diese Annotation kann auch auf Klassenebene verwendet werden. Durch die Annotation *@PostMapping* wird die Methode als POST-Aufruf definiert. Diese ist in diesem Beispiel unter dem Pfad „company“ verfügbar. Der Body des Requests muss in der Form von JSON vorliegen und diese Methode liefert auch Responsedaten in Form von JSON zurück. Die beiden Annotationen *@ResponseBody* und *@RequestBody* mappen den Rückgabewert bzw. den Inputparameter zu JSON. Der Inputparameter darf aufgrund der Annotation *@NotNull* nicht null sein. Dieser muss auch durch die *@Valid*-Annotation valide sein. In der Klasse Company

können für einzelne Member Beschränkungen angegeben werden. Diese werden durch die Annotation @Valid validiert.

Listing 5.5: CompanyControllerApi.java

```
@RequestMapping("/")
public interface CompanyControllerApi {
    ...

  @CrossOrigin
  @PostMapping(value = "company",
    produces = MediaType.APPLICATION_JSON_VALUE,
    consumes = MediaType.APPLICATION_JSON_VALUE)
  @ResponseBody
  ResponseEntity<Company> addCompany(@RequestBody @NotNull
    @Valid Company company);

  ...
}
```

5.3.5 Sicherheitskonfiguration

Durch erweitern der Klasse *WebSecurityConfigurerAdapter* kann die Sicherheitskonfiguration für Spring Boot Applikationen einfach definiert werden. Dadurch muss die Methode *configure* überschrieben werden. Diese liefert die Klasse **HttpSecurity**, mit der die Konfiguration vorgenommen werden kann.

Listing 5.6: SecurityConfiguration.java

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .cors().configurationSource(corsConfigurationSource()).and()
      .authorizeRequests()
      .antMatchers("/login").permitAll()
      .antMatchers("/actuator/health").permitAll()
      .anyRequest().authenticated()
      .and()
      .formLogin()
      .loginProcessingUrl("/login")
      .usernameParameter("username")
      .passwordParameter("password")
      .successHandler(this::loginSuccessHandler)
      .failureHandler(this::loginFailureHandler)
      .and()
      .logout()
      .logoutUrl("/logout")
      .logoutSuccessHandler(this::logoutSuccessHandler)
      .invalidateHttpSession(true).and()
      .exceptionHandling()
      .authenticationEntryPoint((httpServletRequest, httpServletResponse, e) ->
```

```
25              httpServletResponse.setStatus(HttpStatus.UNAUTHORIZED.value()));
26      }
27  }
```

Im Folgenden werden die einzelnen Konfigurationseinstellungen beschrieben:

- **cors** und **configurationSource**: Mit diesen beiden Parametern können weitere Einstellungen bezüglich CORS vorgenommen werden.
- **authorizeRequests**: Mit dieser Einstellung können jegliche REST-Aufrufe nur autorisiert aufgerufen werden.
- **antMatchers**: Durch diese beiden Einstellungen können Ausnahmen für die Autorisierung definiert werden. Diese sind nötig für den Login-Aufruf und den Health-Check.
- **formLogin**: Mithilfe der Zeilen 12 bis 17 im Listing 5.6 wird der Login als Form-Login unter der URL *login* mit den Parametern „username“ und „password“ definiert. Dazu wird auch ein *successHandler* und *failureHandler* für den erfolgreichen bzw. fehlerhaften Login definiert.
- **logout**: Auch für den Logout wird eine URL und ein *successHandler* definiert.
- **exceptionHandling**: Durch die Zeilen 23 und 24 wird eine Default-Nachricht, bei unautorisierten Aufrufen definiert.

5.3.6 Docsis-Service

Im Unterpaket *service* befindet sich die Klasse **DocsisService**. Diese dient zum Managen der Links im Docsis-Service. Da das Docsis-Service ein eigenständiges Microservice ist, werden dazu REST-Aufrufe benötigt. Spring Boot stellt dabei die Klasse RestTemplate zur Verfügung. Damit können die REST-Aufrufe einfach wie in folgendem Listing erledigt werden:

Listing 5.7: DocsisService.java

```
1   @Component
2   public class DocsisService {
3
4     private final RestTemplate restTemplate;
5
6     @Value(value = "${base.service.docsis}")
7     private String restBaseServiceDocsis;
8
9     @Autowired
10    public DocsisService(RestTemplate restTemplate) {
11      this.restTemplate = restTemplate;
12    }
13
14    public ResponseEntity<List<Link>> getAllLinksOfPartner(Long partnerId) {
15      List<Link> links = restTemplate.getForObject(restBaseServiceDocsis +
16        "links/partner/" + partnerId, ArrayList.class);
17      return new ResponseEntity<>(links, HttpStatus.OK);
18    }
19  }
```

Das RestTemplate wird dabei über Konstruktorinjektion injiziert. Die URL zum Docsis-Service befindet sich als Konfigurationsparameter im application.yml und wird über die Annotation

@Value geladen. Mit der Methode *getForObject* des RestTemplate-Objektes kann ein GET-Aufruf zur angegeben URL gemacht werden. Der Parameter *ArrayList.class* gibt an, dass der Body der Response vom Typ *ArrayList* ist. Das Speichern eines Links und Löschen eines Links sehen sehr ähnlich aus und können auch mit dem RestTemplate einfach implementiert werden.

5.3.7 application.yml

Die **applicationy.yml**-Datei dient zur Konfiguration der Anwendung und ist in verschiedene Profile aufgeteilt. Dadurch kann einfach lokal eine andere Datenbankverbindung als in OpenShift verwendet werden. Im Folgenden wird das Profil *dev* näher beschrieben:

Listing 5.8: application.yml

```yaml
spring:
  profiles: dev
  datasource:
    url: jdbc:sqlserver://127.0.0.1:1433;databaseName=PartnerDB
    username: <username>
    password: <password>
base:
  service:
    docsis: http://localhost:8080/
server:
  port: 8090
opentracing:
  jaeger:
    udp-sender:
      host: localhost
      port: 6831
```

Das Profil *dev* wird für die Entwicklung lokal verwendet. Bei der Konfiguration der Datenbank müssen zumindest die *url*, der *username* und das *password* angegeben werden. Der Parameter *base.service.docsis* dient zum Erreichen des Docsis-Service. Dadurch kann auch auf verschiedenen Umgebungen der REST-Endpoint des Docsis-Service einfach konfiguriert werden. Dieser wird in Listing 5.7 mit der Annotation *@Value* verwendet. Der Parameter *server.port* dient zur Spezifikation des Ports hinter dem die Applikation läuft. Dieser ist standardmäßig 8080. Da das Docsis-Service auf Port 8080 konfiguriert ist, muss das Partner-Service auf einem anderen Port laufen, um Kollisionen zu vermeiden. Die *opentracing*-Parameter dienen zur Konfiguration von Jaeger. Diese werden in einem späteren Abschnitt weiter beschrieben.

5.4 Frontend-Beschreibung

Das Frontend ist eine Angular 8 Applikation und dient zu erleichterten Durchführung der Geschäftsfälle. Auch der rollenbasierte Login ist im Frontend implementiert. Grundsätzlich kann jeder Mitarbeiter jede Aktion durchführen. Dies soll in Zukunft, sobald weitere Geschäftsfälle integriert werden, eingeschränkt werden. Loggt sich ein User ein, sieht er alle Unternehmen und Partner der 3 Banken IT GmbH. Durch eine eingebaute Navigation können weitere Reiter, wie Partner anlegen oder Unternehmen anlegen aufgerufen werden. Im Header der Anwendung kann sich der User wieder ausloggen und gelangt dadurch zum Login zurück. Nur eingeloggte User können Aktionen durchführen.

5.4.1 Paketstruktur

In Abbildung 5.3 ist die Paketstruktur vom *Frontend* zu sehen.

Im Folgenden werden die wichtigsten Verzeichnisse genauer beschrieben:

- **src.main.angular.src**
  - **app**: Im app-Verzeichnis befindet sich die eigentliche Anwendung. Diese besteht aus den folgenden Verzeichnissen:
    - **main-pages**: Mit dem Frontend können die Geschäftsfälle *Unternehmen anlegen*, *Parnter anlegen*, *Partner anzeigen*, *Unternehmen anzeigen* abgewickelt werden. Im Verzeichnis main-pages befinden sich die Komponenten für die jeweiligen Geschäftsfälle.
    - **models**: In diesem Verzeichnis befinden sich die Modelklassen für den User, ein Unternehmen, einen Partner und einen Link.
    - **services**: Im Verzeichnis services befinden sich die Angular-Services zur Durchführung der REST-Calls für die einzelnen Geschäftsfälle.
    - **shared**: Im Verzeichnis shared befinden sich viele unterschiedliche Komponenten, die für die Anwendung nötig sind. Dazu gehören Guards zur Absicherung der Routen, Interceptors, wie z.B. der HTTP-Interceptor zum Setzten von Header und Komponenten zur Darstellung der Navigation und des Headers.
    - **signin**: Das Verzeichnis signin besteht aus einer Login-Komponente und einem Authentifizierungsservice.
  - **configuration**: Im Verzeichnis configuration befindet sich die Konfiguration der einzelnen Unterseiten, sowie ein HTTP-Header Objekt zur Verwendung im Interceptor. Diese Konfiguration ist bei allen environments gleich.
  - **environments**: In diesem Verzeichnis befindet sich die Konfiguration der Backend-URL. Diese ist environment-spezifisch und kann daher nicht im configuration-Verzeichnis gewartet werden.

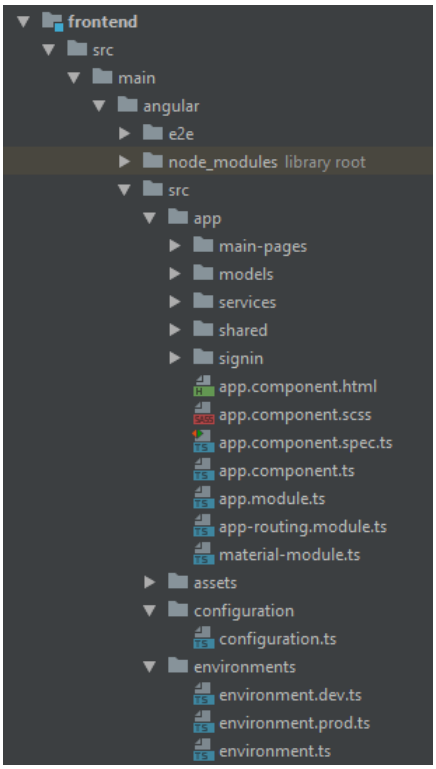

Abbildung 5.3: Paketstruktur Frontend

5.4.2 Environment-Konfiguration

Die Konfiguration der Environments gestaltet sich sehr einfach. Durch das Hinzufügen des Codesnippets in Listing 5.9 in der *angular.json*-Datei wird beim Starten der Anwendung mit dem Parameter *--configuration=<environment>* die Datei *environment.ts* mit der Datei environment.<environment>.ts ausgetauscht. Z.B. beim Start der Anwendung mit dem Befehl:

```
$ ng serve --configuration=dev
```

wird anstatt der environment.ts die Datei environment.dev.ts verwendet. In dieser Datei sind die URL´s zum Backend beim Deployment in OpenShift konfiguriert.

Listing 5.9: angular.json

```
...
"configurations": {
  "dev": {
    "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.dev.ts"
    }
  ]
},
...
```

5.4.3 Konfiguration

In der configuration.ts-Datei sind die einzelnen Seiten und ein HTTP-Header definiert.

Listing 5.10: configuration.ts

```
export const configuration = {
    ...
    PAGES: {
      HOME: '/home',
      ADDPARTNER: '/addpartner',
      PARTNER: '/partner',
      ADDCOMPANY: '/addcompany',
      COMPANY: '/company',
      LOGIN: '/login'
    },
    ...
}
```

Auf den Parameter *HOME* im Listing 5.10 kann im TypeScript-Code mittels *configuration.PAGES.HOME* zugegriffen werden.

5.4.4 Anwendung

In der Anwendung können die Geschäftsfälle *Unternehmen hinzufügen*, *Unternehmen löschen*, *Unternehmen anzeigen* und *Partner anzeigen* abgewickelt werden. Die Logik für diese Geschäftsfälle ist im Verzeichnis *main-pages* enthalten. Je nach derzeitiger URL wird auf die richtige Komponente geroutet. Die Routing-Regeln sind in der Datei *app-routing.module.ts* definiert. Ein Ausschnitt dieser Datei wird in Listing 5.11 gezeigt:

Listing 5.11: app-routing.module.ts

```
export const routes: Routes = [
    ...
    {
      path: 'home',
      component: MainLayoutComponent,
      loadChildren: () => HomePagesModule,
      data: {pageTitle: 'Home'},
      canActivate: [UserAuthenticationGuard]
    },
    ...
```

Diese Komponente ist unter dem Pfad „/home“ erreichbar Die Komponente *MainLayoutComponent* wird dadurch inklusive dem Kindmodul *HomePagesModule* gerendert. Der Header und die Navigation sind in der MainLayoutComponent definiert. Diese lädt danach die HomePagesModule hinzu, in der die eigentliche Logik stattfindet. Durch den *UserAuthenticationGuard* wird überprüft, ob der eingeloggte User diese Seite aufrufen darf.

Für die REST-Aufrufe werden Services verwendet. In diesen wird der HttpClient von Angular injiziert. Dadurch können einfach verschiedene REST-Methoden ausgeführt werden. Das Listing 5.12 zeigt einen Ausschnitt der Datei *partner.service.ts*:

Listing 5.12: partner.service.ts

```
export class PartnerService {
  ...
  private readonly URL = environment.backendOriginSegment + ":" +
    environment.backendOriginPort;

  constructor(private http: HttpClient) {}

  addOrUpdatePartner(id: number, partner: Partner): Observable<Partner> {
    return this.http.post<Partner>(this.URL + "/company/" + id + "/partner",
      JSON.stringify(partner, partner.constructor.prototype),
        .pipe(
          retry(1)
        );
  }
  ...
}
```

Durch die environment-Datei wird die URL aus Host und Port zusammengebaut. Dadurch wird sichergestellt, dass lokal und in OpenShift die richtige URL für das Backend verwendet wird, ohne Code ändern zu müssen.

Beim Aufruf der Methode *addOrUpdatePartner* wird ein POST-Aufruf zur entsprechenden URL mit dem Objekt *Partner* im Body durchgeführt. Die retry-Pipe gibt an wie oft dieser Aufruf bei einem Fehlversuch wiederholt werden soll. Diese Methode gibt ein Observable zurück, auf das sich registriert werden kann.

Der Aufruf dieser Methode sieht wie folgt aus:

Listing 5.13: addPartner-Methode

```
addPartner(): void {
  this.addOrUpdatePartnerSubscription =
    this.partnerService.addOrUpdatePartner(this.selectedCompany.id, this.partner)
      .subscribe((partner: Partner) => {
        ...
      })
}
```

Die *addOrUpdatePartner*-Methode im PartnerService wird mittels der UnternehmensID und dem veränderten oder neu hinzugefügten Partner aufgerufen. Dadurch dass diese Methode ein Observable zurückliefert, kann man sich mittels der Methode *subscribe* auf dieses Observable registrieren. Sobald die Response vom Server kommt, wird der Code in der *subscribe*-Methode aufgerufen.

5.4.5 Starten der Anwendung

Lokal kann das Frontend über folgenden Kommandozeilenbefehl gestartet werden:

```
$ ng serve --configuration=dev
```

Mittels dem Parameter *--configuration=dev* wird das lokale Environment gestartet. Mittels *ng serve* wird ein von Angular mitgelieferter Webserver hochgefahren. In diesem läuft die Applikation und kann im Browser unter der URL *http://localhost:4200* aufgerufen werden.

Mit folgendem Befehl kann die Applikation in **OpenShift** deployt werden:

```
1 $ npx nodeshift --dockerImage=nodeshift/ubi8-s2i-web-app
2   --imageTag=10.x --build.env OUTPUT_DIR=dist/frontend --expose
```

Nodeshift ist ein Kommandozeilentool und programmierbare API zum Deployment von Node.js-Anwendungen in OpenShift. Dabei wird das Docker Image *nodeshift/ubi8-s2i-web-app* mit dem Image Tag *10.x* verwendet. Das Output-Directory wird auf *dist/frontend* gesetzt. Dies ist das Verzeichnis, in dem die Applikation gebaut wird. Mittels *--expose* wird die Route der Applikation in OpenShift freigegeben und ist damit durch einen Browser erreichbar.

Kapitel 6

Design und Implementierung der Partnerdatenbank ... 17 Seiten

6.1 Microservice-Architektur ... 2 Seiten

Die gesamte Applikation wird, wie in Abbildung 6.1 zu sehen, als Microservice-Architektur designt. Dabei wird die Designvariante *Service-Orchestrierung* gewählt. Das Partner-Service fungiert dabei als Kompositionsservice. Das Partner-Service wird dabei zur zentralen Autorität. Dies ist jedoch auch die Schwachstelle einer Microservice-Anwendung. Fällt dieses Service aus, steht die gesamte Anwendung. Dadurch wird das Partner-Service zum *Single-Point-of-Failure* in dieser Architektur. Durch das Replizieren, Skalieren und Clustering dieses Service in OpenShift kann dieses Problem einfach gelöst werden.

Wird am Frontend ein Partner- oder Unternehmens-Objekt benötigt, ruft das Frontend das Partner-Service auf. Dieses holt sich aus der PartnerDB das entsprechende Objekt und liefert dieses als Response zurück. Werden auch die Links zu den verschiedenen Partner benötigt, leitet das Partner-Service den Aufruf zum Docsis-Service weiter. Dieses holt sich die entsprechenden Links zum jeweiligen Partner aus der DocsisDB und liefert diese an das Partner-Service zurück. Das Partner-Service verpackt die Links in ein Partner-Objekt und liefert das Partner-Objekt an das Frontend zurück.

Werden weitere Services hinzugefügt, ruft das Partner-Service auch diese auf und liefert die Response an das Frontend zurück. In Abbildung 6.1 ist zu sehen, wie das Angebotslegungs-Service in die Applikation integriert werden kann. Dieses verwaltet die Angebote in der AngebotsDB. Da die Dokumente der Angebotslegung wahrscheinlich auch in der DocsisDB gespeichert sind, könnte das Partner-Service auch das Docsis-Service aufrufen. Um die Struktur der Anwendung so einfach wie möglich zu halten, kann auch der Umweg über das Partner-Service erfolgen. Somit redet jedes Service lediglich mit dem Partner-Service und es ist ein klarer Workflow ersichtlich.

In OpenShift wird dann lediglich die Route des Partner-Services freigegeben. Alle anderen Services kommunizieren nur intern und sind auch nur innerhalb des Clusters aufrufbar. Dadurch ist auch der Login und die Authentifizierung, sowie die Autorisierung lediglich für das Partner-Service zu implementieren, da externe Clients nur Zugriff auf dieses Service haben und alle Geschäftsfälle zuerst über das Partner-Service abgewickelt werden.

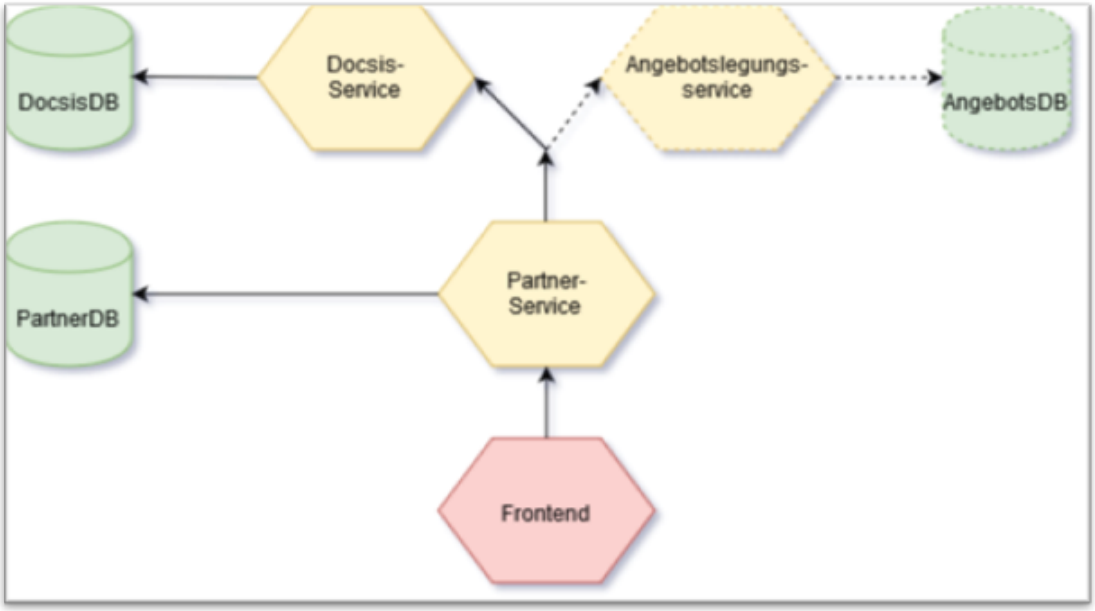

Abbildung 6.1: Design der Microservice-Architektur

6.2 Spring Boot Microservices

Spring Boot unterstützt Entwickler bei der Erstellung von Microservice-Technologien ungemein. Spring Boot bietet dabei eine Fülle an Abhängigkeiten, die besonders für die Entwicklung von Microservices geeignet sind.

6.2.1 Fehlerbehandlung mit Spring Retry ... 2 Seiten

Spring Retry bietet die Möglichkeit eine fehlgeschlagene Operation erneut ausführen zu lassen. Dies ist besonders bei REST-Aufrufen sehr wichtig. Dabei können Fehler einfacher und schneller auftreten, als würde eine Methode eine andere Methoden in derselben JVM aufrufen. Spring Retry bietet dabei die Kontrolle über den Prozess und lässt sich einfach erweitern und anpassen.

Bei der Verwendung von Spring Retry wird folgende Dependency benötigt:

Listing 6.1: pom.xml

```
1  <dependency>
2    <groupId>org.springframework.retry</groupId>
3    <artifactId>spring-retry</artifactId>
4  </dependency>
```

Zur Verwendung von Spring Retry muss eine Konfigurationsklasse mit *@EnableRety* annotiert werden. Dadurch kann jede Methode mit einer der folgenden zwei Annotationen annotiert werden.

@Retryable

Listing 6.2: DocsisService.java

```
@Retryable(value = {RestClientException.class, WebApplicationException.class},
    maxAttempts = 5,
    backoff  = @Backoff(delay = 2000)
)
public ResponseEntity<List<Link>> getAllLinksOfPartner(Long partnerId) {
    List<Link> links = restTemplate.getForObject(restBaseServiceDocsis +
      "links/partner/" + partnerId, ArrayList.class);
    return new ResponseEntity<>(links, HttpStatus.OK);
}
```

Durch den Parameter *value = RestClientException.class, WebApplicationException.class* wird der Retry nur ausgeführt, sobald der Fehler vom Typ RestClientException oder WebApplicationException ist. Bei anderen Exceptions wird der Retry-Mechanismus nicht ausgelöst.

Durch *maxAttempts* kann die maximale Anzahl an Retry-Versuchen angegeben werden. Diese Methode wird bei immer wieder auftretenden RestClientExceptions oder WebApplicationExceptions maximal fünf mal wiederholt.

Damit die Methode nicht sofort nach einem Fehlversuch neu ausgeführt wird, kann ein *Backoff* angegeben werden. In diesem Fall wird zwei Sekunden nach dem Fehlversuch die Methode neu ausgeführt.

@Recover

Eine weitere nützliche Annotation von Spring Retry ist *@Recover*. Schlägt die Methode aus Listing 6.2 zum fünften Mal fehl, wird die Methode aus Listing 6.3 aufgerufen. Da sich die Behandlung bei unterschiedlichen Exceptions natürlich unterscheidet, kann die Recover-Methode auch für jede Exceptions explizit definiert werden. Es wird dabei automatisch die richtige Methode, die zum Typ der geworfenen Exception passt, aufgerufen.

Listing 6.3: DocsisService.java

```
@Recover
public void recoverRestClientException(RestClientException ex){
    ...
}

@Recover
public void recoverWebApplicationException(WebApplicationException ex){
    ...
}
```

RetryTemplate

Spring Retry bietet auch ein Template zur generellen Konfiguration von Retry-Methoden. Im Listing 6.4 wird eine mögliche Konfiguration von Spring Retry gezeigt.

Listing 6.4: RetryConfig.java

```
@Configuration
public class RetryConfig {
    ...
  @Bean
```

```
5     public RetryTemplate retryTemplate() {
6       RetryTemplate retryTemplate = new RetryTemplate();
7
8       FixedBackOffPolicy fixedBackOffPolicy = new FixedBackOffPolicy();
9       fixedBackOffPolicy.setBackOffPeriod(2000l);
10      retryTemplate.setBackOffPolicy(fixedBackOffPolicy);
11
12      SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
13      retryPolicy.setMaxAttempts(2);
14      retryTemplate.setRetryPolicy(retryPolicy);
15
16      return retryTemplate;
17    }
18    ...
19  }
```

Die *RetryPolicy* bestimmt, wann eine Methode wiederholt werden soll. In diesem Fall zwei mal. Die *BackOffPolicy* wird zur Definition des Backoffs verwendet.

Listing 6.5: RetryOperations

```
1  public interface RetryOperations {
2    <T> T execute(RetryCallback<T> retryCallback) throws Exception;
3    ...
4  }
```

Das *RetryTemplate* ist ein Klasse, die das Interface RetryOperations aus Listing 6.5 implementiert. Dadurch kann das Template wie folgt aufgerufen werden:

Listing 6.6: Aufruf mit anonymer Funktion

```
1  retryTemplate.execute(new RetryCallback<Void, RuntimeException>() {
2    @Override
3    public Void doWithRetry(RetryContext arg0) {
4      myService.templateRetryService();
5       ...
6    }
7  });
```

Alle Methoden innerhalb der *doWithRetry-Methode* unterliegen nun der Spezifikation von RetryConfig. Der Aufruf kann natürlich auch einfacher mit einer Lambda Expression statt einer anonymen Funktion aufgerufen werden:

Listing 6.7: Aufruf mit Lambda

```
1  retryTemplate.execute(arg0 -> {
2    myService.templateRetryService();
3    ...
4  });
```

6.2.2 REST-Schnittstellenbeschreibung mit Swagger ... 2 Seiten

Gerade bei Microservice-Applikationen ist es essentiell passende Spezifikationen und Dokumentationen für die REST-Schnittstellen zu erstellen. Diese Dokumentationen sollten informativ, lesbar und einfach zu folgen sein.

Wird diese Dokumentation manuell erstellt, werden oft Änderungen in der Api nicht sofort in der Dokumentation nachgezogen. Dies ist natürlich für Aufrufer der Api und Anwender des Service fatal, da Aufrufe plötzlich nicht mehr funktionieren und es scheinbar keine Änderungen gab.

Dieses Problem löst Swagger. Swagger ist ein Tool zur automatischen Dokumentation der REST-Api. Dadurch ist die Dokumentation stets aktuell.

In der Partnerdatenbank wird die Springfox Implementierung von Swagger verwendet. Dazu wird folgende Dependency benötigt:

Listing 6.8: pom.xml

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
</dependency>
```

Swagger wird durch die Annotation *@EnableSwagger2* aktiviert. Das Docket-Bean muss definiert werden. Die *select*-Methode des Docket-Bean liefert eine Instanz der Klasse *ApiSelectorBuilder* zurück. Diese bietet einen Weg, um die Endpoints von Swagger zu kontrollieren. Durch die *RequestHandlerSelections* wird angegeben, dass lediglich REST-Methoden im Basispaket „at.fh.se.master.company“ durch Swagger dokumentiert werden.

Listing 6.9: Swagger-Konfiguration

```
@SpringBootApplication
@EnableSwagger2
public class CompanyServiceApplication {
  ...
  @Bean
  public Docket productApi() {
    return new Docket(DocumentationType.SWAGGER_2)
    .select()
    .apis(
      RequestHandlerSelectors
        .basePackage("at.fh.se.master.company")
    ).build();
  }
  ...
}
```

Die Swagger-Dokumentation ist nach dem Start der Applikation unter dem Link *http://localhost:8080/spring-security-rest/api/v2/api-docs* erreichbar. Dies ist ein JSON-Objekt mit Key-Value Paaren. Diese Datei ist leider für Menschen nicht gut lesbar. Springfox bietet daher mit folgender Dependency eine UI zur besseren Darstellung der REST-Methoden:

Listing 6.10: pom.xml

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
</dependency>
```

Die UI ist nach dem Starten der Applikation unter *http://localhost:8080/swagger-ui.html* erreichbar. Sie zeigt alle REST-Methoden der Applikation in einer grafischen Darstellung an. Mit dieser UI können die einzelnen REST-Methoden auch aufgerufen und getestet werden.

6.2.3 Tracing mit Jaeger ... 1 Seite

Jaeger ist eine Ende-zu-Ende verteiltes Tool zum Tracen von Serviceaufrufen in Anwendungen. Bei der Implementierung einer Microservice-Architektur treten viele Probleme und Hürden auf. Diese Probleme betreffen hauptsächlich die beiden Gebiete **Netzwerke** und **Aufspürbarkeit von Services**. Es ist sehr viel komplexer ein System bestehend aus kleinen Untersystem zu debuggen, als eine einzelne Applikation.

Jaeger addressiert dabei folgende Probleme:

- Verteiltes Monitoring von Transaktionen.
- Performanz- und Latenzoptimierungen.
- Ursachenanalyse.
- Analyse der Serviceabhängigkeiten.
- Verteilte Propagierung von Kontexten.

Folgenden Features bietet Jaeger, um die Nutzung noch einfacher zu machen:

- **Hohe Skalierbarkeit**: Das Jaeger Backend besitzt keinen Single-Point-of-Failure und skaliert mit den Businessanforderungen. Dies ist gerade bei der Entwicklung von Microservices von großer Bedeutung.
- **Nativer Support für OpenTracing**: Das Jaeger Backend, die Web UI und die Bibliotheken von Jaeger sind so designt, dass sie von Grund auf den OpenTracing Standard unterstützen. Dazu gehören:
  - Das repräsentieren der Traces als direkte azyklische Graphen über Span-Referenzen.
  - Der Support von typisierten Span Tags und strukturierten Logs.
  - Der Support von verteilter Kontextpropagierung.
- **Mehrere Speicher-Backends**: Jaeger unterstützt zwei populäre OpenSource NoSQL Datenbanken als Speicher-Backends: Cassandra und Elasticsearch. Jaeger enthält auch einen In-Memory Speicher für Test Setups.
- **Modernes Web UI**: Die Jaeger Web UI wurde mit JavaScript und React entwickelt. Verschiedene Performanzverbesserungen wurden im Laufe der Jahre eingebaut, um tausende Spans zu tracen.
- *Cloud-Native Deployment*: Das Jaeger Backend besteht aus einer Sammlung aus Docker Containern. Das Deployment zu Kubernetes Cluster wird durch *Kubernetes Operatoren*, *Kubernetes Template* und *Helm Charts* vereinfacht.
- **Beobachtbarkeit**: Alle Jaeger Backend Komponenten bieten standardmäßig Metriken an. Logs werden dabei über das Standard-Out mittels strukturiertem Logging ausgegeben.
- **Rückwärtskompatibilität mit Zipkin**: Sofern bestehende Anwendungen Zipkin-Bibliotheken verwenden, sind diese auch mit Jaeger kompatibel. Anwendungen, die bereits mit Zipkin getraced werden, müssen daher nicht neu geschrieben werden.

Auch für Jaeger gibt es eine UI zur Darstellung der Serviceaufrufe. Um die *JaegerUI*, den *collector*, die *query* und den *agent* inklusive einer Speicherkomponente zu starten, kann folgender Docker-Befehl in der Kommandozeile ausgeführt werden:

```
1 $ docker run -d --name jaeger \
2   -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
3   -p 5775:5775/udp \
4   -p 6831:6831/udp \
5   -p 6832:6832/udp \
```

```
6   -p 5778:5778 \
7   -p 16686:16686 \
8   -p 14268:14268 \
9   -p 14250:14250 \
10  -p 9411:9411 \
11  jaegertracing/all-in-one:1.17
```

Dieser Befehl startet einen Docker Container aus dem Docker Image *jaegertracing/all-in-one:1.17* und gibt die entsprechenden Ports frei. Diese Applikation kann nun unter der URL *http://localhost:16686* aufgerufen werden und kann Traces über die entsprechenden Ports entgegennehmen.

Für die Verwendung von Jaeger in einer Spring Boot Applikation muss folgende Dependency hinzugefügt werden:

Listing 6.11: pom.xml

```
1  <dependency>
2    <groupId>io.opentracing.contrib</groupId>
3    <artifactId>opentracing-spring-jaeger-cloud-starter</artifactId>
4  </dependency>
```

Listing 6.12: application.yml

```
1  spring:
2    application:
3      name: partner-service
4  opentracing:
5    jaeger:
6      udp-sender:
7        host: localhost
8        port: 6831
```

Um die JaegerUI lokal erreichen zu können, müssen im *application.yml* zumindest die in Listing 6.12 gezeigten Parameter angegeben werden. Die Traces dieser Applikation werden nun unter dem Namen „partner-service“ angezeigt. Der *host*- und *port*-Parameter geben den Endpoint der JaegerUI, hinter der der Collector sitzt, an.

Für das Deployment der JaegerUI in OpenShift steht ein Template zur Verfügung. Mit folgendem Befehl kann die JaegerUI einfach deployt werden:

```
1 $  oc process -f jaeger-all-in-one-template.yml | oc create -f -
```

Dadurch prozessiert OpenShift das Template und erstellt auch die Applikation. Auch die Route wird freigegeben und die JaegerUI ist im Browser erreichbar. Für die Verwendung der JaegerUI in OpenShift müssen im application.yml die Parameter wie folgt geändert werden:

Listing 6.13: application.yml

```
1 spring:
2   application:
3     name: partner-service
4 opentracing:
5   jaeger:
6     udp-sender:
7       host: jaeger-agent
8       port: 6831
```

Dadurch werden die Traces an die JaegerUI im selben Cluster, wie die Anwendung, gesendet.

Grundsätzlich werden alle REST-Methoden getraced, es kann jedoch mit der Annotation *@Traced* eine Methode ausgeschlossen werden.

Listing 6.14: LinkControllerApi.java

```
1  @RequestMapping("/")
2  @Traced(false)
3  public interface LinkControllerApi {
4    ResponseEntity<List<Link>> getAllLinksOfPartner(@PathVariable("id")
5      Long partnerId);
6
7    @Traced(true)
8    ResponseEntity addLink(@RequestBody @NotNull @Valid Link link);
9
10   ResponseEntity deleteLink(@PathVariable Long id);
11 }
```

Die Annotation @Traced kann auch auf der Klassenebene hinzugefügt werden. Dadurch sind alle Methoden dieser Klasse betroffen. Durch dieselbe Annotation auf Methodenebene kann eine bestimmte Methode aus der Konfiguration der Klassenebene ausgenommen werden. In Listing 6.14 wird dadurch lediglich die Methode *addLink* getraced.

6.2.4 Metrics

6.3 Einsatz von Docker zur Containerisierung der Anwendung ... 1 Seite

6.4 Automatisierte Test-, Build- und Deployment-Pipelines mit Jenkins ... 3 Seiten

6.5 Konfiguration und Deployment-Deskriptoren von OpenShift ... 4 Seiten

6.6 Deployment in OpenShift mit Fabric8 ... 2 Seiten

Kapitel 7

Evaluierung der Anwendung ... 7 Seiten

7.1 Evaluierung des Frontends ... 1.5 Seiten

7.2 Whitebox-Tests ... 2 Seiten

7.3 Blackbox-Tests ... 1.5 Seiten

7.4 Architekturevaluierung ... 2 Seiten

Kapitel 8

Zusammenfassung ... 2-3 Seiten

8.1 Resümee

8.2 Ausblick

Quellenverzeichnis

Literatur

[Dupleton, 2018] Graham Dupleton. *Deploying to OpenShift*. O'Reilly Media, Inc., 2018.

[Gallagher, 2015] Scott Gallagher. *Mastering Docker*. Packt Publishing Limited, 2015.

[IndrasiriSiriwardena, 2018] Kasun Indrasiri und Prabath Siriwardena. *Microservices for the Enterprise*. Apress, 2018.

[Jangla, 2018] Kinnary Jangla. *Accelerating Development Velocity Using Docker*. Apress, Berkeley, CA, 2018.

[Krochmalski, 2016] Jaroslaw Krochmalski. *Developing with Docker*. Packt Publishing Limited, 2016.

[Newman, 2015] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[Posta, 2016] Christian Posta. *Microservices for Java Developers*. O'Reilly Media Inc., 2016.

[Rajesh, 2016] Rajesh. *Spring Microservices*. Packt Publishing Limited, 2016.

[Richards, 2016] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media Inc., 2016.

[Schenker, 2019] Gabriel Schenker u. a. *Getting Started with Containerization*. Packt Publishing Limited, 2019.

[Sharma, 2017] Umesh Ram Sharma. *Practical Microservices*. Packt Publishing Limited, 2017.

[Turnbull, 2015] James Turnbull. *The Docker Book: Containerization Is the New Virtualization*. O'Reilly Media, Inc., 2015.

[Wang, 2012] Lizhe Wang u. a. *Cloud Computing*. Taylor und Francis Group, 2012.

[Zuev, 2018] Denis Zuev u. a. *Learn Openshift*. Packt Publishing Limited, 2018.

Online-Quellen

[RedHatInc, 2019] RedHatInc. *OpenShift Container Platform 3.5 Architecture*. 2019. URL: https://docs.openshift.com/container-platform/3.5/architecture (besucht am 25. 03. 2020).

[Tilkov, 2015] Stefan Tilkov. *Don't Start with a Monolith*. 2015. URL: https://martinfowler.com/articles/dont-start-monolith.html (besucht am 02. 01. 2020).