Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Daniel Gehn

SaaS mit Spring und Docker

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Gehn

SaaS mit Spring und Docker

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 16. Februar 2017

Daniel Gehn

Thema der Arbeit
SaaS mit Spring und Docker

Stichworte
SaaS, Spring, Docker, Microservices, OAuth, Containerisierung, REST, HATEOAS, HAL

Kurzzusammenfassung
Ziel dieser Bachelorarbeit ist die Entwicklung einer Software as a Service Anwendung für Modelagenturen. Im Verlauf der Arbeit werden sowohl die Anforderungen als auch das darauf aufbauende Konzept behandelt. Auf diesem basierend werden die Umsetzung wichtiger Komponenten und der Betrieb in Docker erläutert, um eine lauffähige SaaS Anwendung zu erstellen. Die Arbeit setzt hierbei grundlegende Kenntnis über die Software Entwicklung und Architektur voraus.

Daniel Gehn

Title of the paper
SaaS with Spring and Docker

Keywords
SaaS, Spring, Docker, Microservices, OAuth, Containerization, REST, HATEOAS, HAL

Abstract
Target of this bachelor thesis is the development of a Software as a Service application for model agencies. Throughout this thesis the requirements will be discussed as well as the concept build upon it. Based on this, the implementation of important components and the operation with Docker to create a working SaaS application will be explained. The paper requires basic knowledge of software development and architecture.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die Modelwelt befindet sich stets in Bewegung und die Agenturen müssen sowohl mit ihren Kunden in Kontakt stehen als auch die Models koordinieren. Während die Models von einem Termin zum nächsten eilen, arbeiten die Mitarbeiter der Modelagenturen im Hintergrund, um neue Aufträge zu generieren, Unterkünfte und Reisen zu buchen und die Abrechnung durchzuführen.

An dieser Stelle kann eine entsprechende Software viel Arbeit abnehmen und Fehler vermeiden. Termine, Kunden, Models und Weiteres kann einfach verwaltet und teilweise automatisiert abgearbeitet werden. Hierfür existieren bereits einige Lösungen auf dem Markt, welche jedoch entweder veraltet sind oder nicht genügend der täglichen Aufgaben abdecken. Dies führt zu einem mehr an Arbeit und Stress – sowohl bei den Models als auch in der Agentur.

Im Gespräch mit mehreren deutschen Agenturen wurden die Anforderungen an eine solche Software und die Missstände von aktuellen Lösungen analysiert. Um einen Überblick dieser Gespräche zu gewähren, werden im Folgenden die Wichtigsten dieser Anforderungen erläutert.

Als nachgefragtester Punkt stellt sich hierbei die Buchhaltung heraus. Dieser Bereich besteht in bisherigen Lösungen nur rudimentär oder ist für die Agenturen vom Umfang her nicht ausreichend. Die Buchhaltung in Deutschland stellt sich meist komplexer und genauer dar, als es die Implementationen ermöglichen und zulassen. Dies erzeugt in den meisten Fällen eine Zettelwirtschaft in Verbindung mit einer doppelten Datenpflege in einer zusätzlichen Buchhaltungssoftware.

Nachfolgend stellt die Kommunikation mit den Models eine Baustelle dar, welche nicht ausreichend gelöst ist. Besonders die großen Distanzen und die fehlende Einbindung des Models – abgesehen von den Daten des Models – in die Software führt oft zum Verpassen oder dem

Verspäten bei Terminen. Gründe hierfür sind unter anderem fehlende Wegbeschreibungen und nicht kommunizierte Terminänderungen, aber auch nach einem Termin endet die Kommunikation mit dem Model nicht. Belege für Taxifahrten und andere Unkosten müssen mit entsprechenden Belegen aufgeführt werden, da diese sonst nicht mit dem Kunden abgerechnet werden können. Diese Dinge werden meist nur "mal eben" per Telefon kommuniziert oder in einer kurzen E-Mail ausgetauscht. Da dies außerhalb der Systeme geschieht, geht oftmals unter, welcher Beleg schon vorhanden ist oder ob eine Terminänderung schon mitgeteilt wurde.

Diese Probleme führten zu der Idee eine neue Software zu entwickeln, welche alle Aufgaben einer Model- oder auch allgemeinen Castingagentur vereinfacht, automatisiert und die Schwächen der bisherigen Produkte löst. Ein Teil des Entwurfs und der Entwicklung dieser Software wird im Rahmen dieser Bachelorarbeit behandelt.

1.2. Überblick der Arbeit

Die Arbeit wird zu Beginn einige Grundlagen für die folgenden Kapitel vermitteln, welche besonders das Konzept und die Implementation betreffen. Den Grundlagen folgt eine fachliche Analyse einiger Funktionen – siehe auch Abschnitt 1.1 (S. 1) – , welche im Gespräch mit den Agenturen aufgekommen sind. Auf diesem Wissen aufbauend entsteht dann das Konzept der Software, welches sowohl die Architektur als auch die technischen Anforderungen behandelt. Die Realisierung dieses Konzepts wird einige Elemente aus den jeweiligen Teilbereichen und Technologien der Software enthalten, gefolgt von einer abschließenden Evaluierung über den Verlauf des Prozesses inklusive einer Aussicht auf mögliche Erweiterungen des Systems.

2. Grundlagen

In diesem Kapitel werden die wichtigsten Konzepte und Technologien, welche im Verlauf der Arbeit verwendet werden, erläutert. Ziel ist es einen Überblick und ein besseres Verständnis für die folgenden Kapitel über **Software as a Service (SaaS)**, **Microservices**, **Containerisierung**, **Docker** und **Spring** zu vermitteln.

2.1. Software as a Service

Das Konzept SaaS, auf welches viele Unternehmen sowohl als Anbieter (vgl. IAO (2010a)) als auch Nutzer (vgl. Bitkom (2016)) schon seit einigen Jahren setzen und das auch zunehmend umsatzstärker wird (vgl. Gartner (2016), Experton (2016)), stellt einen völlig anderen Ansatz zu klassisch lizenzierter Software dar. Der Unterschied zwischen klassisch lizenzierter Software und einem SaaS System beginnt schon beim Betrieb der Software.

Bisher musste ein Kunde meist eine Lizenz für die zu verwendende Software kaufen und diese dann auf eigener Hardware betreiben oder für den Managed Betrieb zusätzlich bezahlen. Dies bedeutete sowohl viel Aufwand, Kosten und einen hohen Investitionsfaktor. Auch fallen eine Installation und die Aktualisierung von einem Client bei dem Kunden weg, da SaaS Systeme typischerweise als Webapplikation im Browser verfügbar sind und somit nur diesen benötigen. Durch den Betrieb des Systems beim Anbieter kann der Kunde Kosten und Risiko einsparen und sich auf das eigentliche Geschäft konzentrieren. Doch auch dem Betreiber erspart das SaaS Prinzip einige Probleme. Alle Kunden nutzen garantiert die gleiche Version der Software und gemeldete Fehler können seltener von veralteten, nicht gepflegten Installationen stammen. Updates können für alle Kunden einfacher verteilt werden und clientseitige Fehlerquellen stellen hierbei meist veraltete Browser Versionen dar. (vgl. IAO (2010b), Koch (2009))

Neben diesen und weiteren Vorteilen gibt es auf beiden Seiten auch Nachteile. Der Kunde macht sich vom Anbieter komplett abhängig, hat weniger Anpassungsmöglichkeiten sowie eine geringere Kontrolle über seine Daten. Auch die Frage über den Datenschutz stellt oft eine Hürde für Kunden beim Wechsel zu SaaS Lösungen dar. Währenddessen besteht für den

Anbieter das höhere Investitionsrisiko in Form der Infrastruktur, sowie ein hohes Risiko bei Ausfällen oder Problemen im System, welches meist alle Kunden betrifft. (vgl. Eurostat (2014), Koch (2009))

2.2. Microservices vs. Monolith

Während es verschiedene Ansätze gibt ein SaaS System umzusetzen, werden viele der oben genannten Punkte von Microservices erfüllt und können mit einer monolithischen Software nur erschwert realisiert werden.[1]

Eine Microservice Architektur kann zur Entwicklung einer einzelnen Anwendung genutzt werden, die aus einer Sammlung von kleinen Diensten besteht. Diese Dienste wiederum laufen in ihrem eigenen Prozess und kommunizieren meist über einen simplen Weg, wie HTTP APIs.[2]

Durch diese Eigenschaften bieten Microservices verschiedene Vor- und Nachteile im Vergleich zum klassischen Monolithen. Während Microservices eine klare und modulare Trennung besitzen, ist dies in monolithischen Anwendungen nicht unbedingt gegeben. Des Weiteren können Dienste – da diese autonom arbeiten – einfacher aufgesetzt werden und erzeugen im Fehlerfall seltener Systemausfälle. Dank mehrerer Instanzen und einer in der Regel verfügbaren Fehlerbehandlung in den Diensten sind Ausfälle einzelner Dienste oder Instanzen weniger problematisch als der Ausfall ganzer Monolithen. Als zusätzlicher Vorteil erweist sich die Unabhängigkeit, da unterschiedliche Sprachen, Frameworks oder auch Systeme zur Datenhaltung genutzt werden können. Dies ermöglicht eine flexible Umsetzung der Anforderungen, wobei eine zu große Diversität schnell unübersichtlich werden und die Wartung erschweren kann. Zusätzlich können in der Regel hierdurch auch Infrastruktur Kosten verringert werden, da auf der Hardware Ebene pro Dienst anstatt pro Instanz eines Monolithen skaliert werden kann.[3]

All den positiven Punkten stehen einige Nachteile entgegen. Zunächst ist die Umsetzung eines verteilten Systems oftmals komplexer als eines Monolithen. Auch Performanceprobleme, welche bei Aufrufen innerhalb eines Prozesses eher selten auftreten, können durch Asynchronität – eine Antwort ist nur so schnell wie die langsamste Anfrage – und die ungewisse Zuverlässigkeit von externen Anfragen entstehen. Prozess interne Aufrufe liefern normalerweise immer ein Ergebnis, während ein aufgerufener Dienst ausfallen kann und somit eine

[1] vgl. Villamizar u. a. (2015)
[2] Fowler
[3] Vorteile aus Fowler (2015), Infrastuktur Kosten vgl. Villamizar u. a. (2015) *Table II & Table III*

Antwort ausbleibt. Hinzu kommt, dass die Arbeit mit Asynchronität und das Sicherstellen der Konsistenz die Umsetzung erschweren. Durch eine fehlende zentrale Datenhaltung führen Änderungen an mehreren Ressourcen zu einer Eventual consistency im Kontrast zur garantierten Konsistenz. Zuletzt stellt sich der Betrieb als komplexer dar, da die Anzahl der zu verwaltenden und überwachenden Elemente steigt und die Schnelllebigkeit von Diensten und Tools schnell unübersichtlich werden können.[4]

Insgesamt stellen sich Microservices jedoch als die bessere Lösung für ein SaaS System dar, da ihre Flexibilität auf lange Sicht den Betrieb erleichtern. Es kann präziser auf Laständerungen eingegangen, Betriebskosten können verringert und Updates, wie beispielsweise Hotfixes, können schneller veröffentlicht werden.

2.3. Containerisierung

Nachdem Microservices zur Umsetzung des Systems dienen sollen, werden die unterschiedlichen Varianten des Betriebs betrachtet. Sowohl in der Entwicklung als auch im späteren Produktiveinsatz zeigen sich die Vor- und Nachteile der existierenden Lösungen. Die klassische Variante stellt der Betrieb auf einem Server pro Dienst und Instanz oder mehrerer Dienste und Instanzen dar. Hierbei ist es nicht relevant, ob diese Server dediziert oder virtuell sind. Doch birgt dieser Ansatz einige Probleme, insbesondere bei der Trennung einzelner Dienste und deren Dateisystemen, welche der Sicherheit und Stabilität entgegen stehen. Eine Lösung, um diese Konflikte zu verringern oder gar auszuschließen, stellt der Betrieb jeder Instanz auf einem eigenen System – wieder unabhängig ob virtuell oder dediziert – dar. Eine dedizierte Lösung wäre technisch möglich, jedoch nicht wirtschaftlich. Somit ist aus wirtschaftlicher Sicht eine Lösung mit virtuellen Maschinen der bessere Weg.

Wie es die Abbildung 2.1 (S. 6) aufzeigt, befindet sich in diesem Fall auf dem Host-Betriebssystem ein Hypervisor, welcher die einzelnen Gast Betriebssysteme (die virtuellen Maschinen) verwaltet. Auf diesem befinden sich die Binaries, Bibliotheken und Anwendungen. Dies führt zu einer strikten Trennung einzelner Instanzen, benötigt jedoch ein komplettes Gastbetriebssystem pro Dienstinstanz inklusive Kernel und weiteren Systemprozessen.

Dieser Umstand kann noch optimiert werden, indem das Prinzip der Containerisierung eingesetzt wird. Abbildung 2.2 (S. 7) zeigt, dass auch hier die Trennung der Anwendungen inklusive

[4] Nachteile aus Fowler (2015)

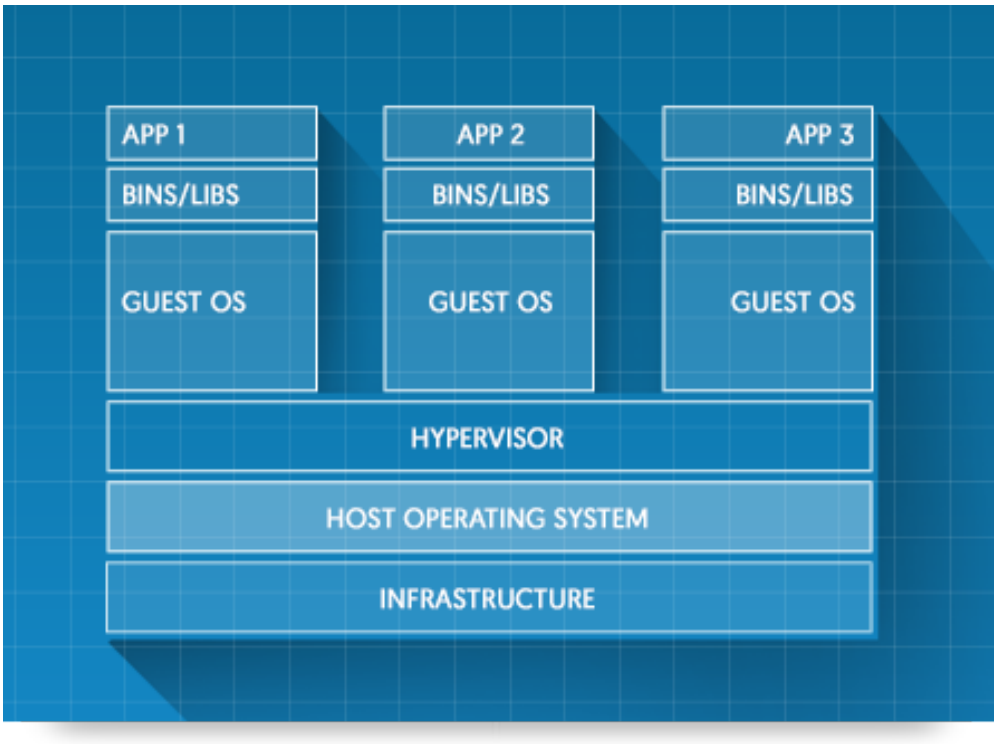

Abbildung 2.1.: Virtuelle Maschinen (Quelle: Docker (2016) *Comparing Containers and Virtual Machines*)

Binaries und Bibliotheken besteht, jedoch werden Gast OS und Hypervisor – am Beispiel von Docker – durch die Docker Engine ersetzt. Die Docker Engine führt die einzelnen Container isoliert voneinander, ohne den mehrfachen Betrieb eines kompletten Gast OS, aus, wodurch alle Container über den Kernel des Host-Betriebssystem ausgeführt werden. (Quelle: Docker (2016))

Neben diesen Vorteilen besitzt die Containerisierung unter anderem die Nachteile, dass sie nicht auf unterschiedlichen Kerneln ausgeführt werden können – da alle Container einen Kernel teilen – und ein erhöhtes Sicherheitsrisiko – im Kontrast zu virtuellen Maschinen – durch die gemeinsame Nutzung eines Kernels entsteht. Jedoch bietet diese Reduzierung des Overheads – im Vergleich zu virtuellen Maschinen – eine Reduktion von CPU Last und Latenz, was in der Praxis unter anderem Kosten im Bereich der Infrastruktur einspart sowie unter hoher Last die Antwortzeiten verbessert. [5]

[5] Spoiala u. a. (2016)

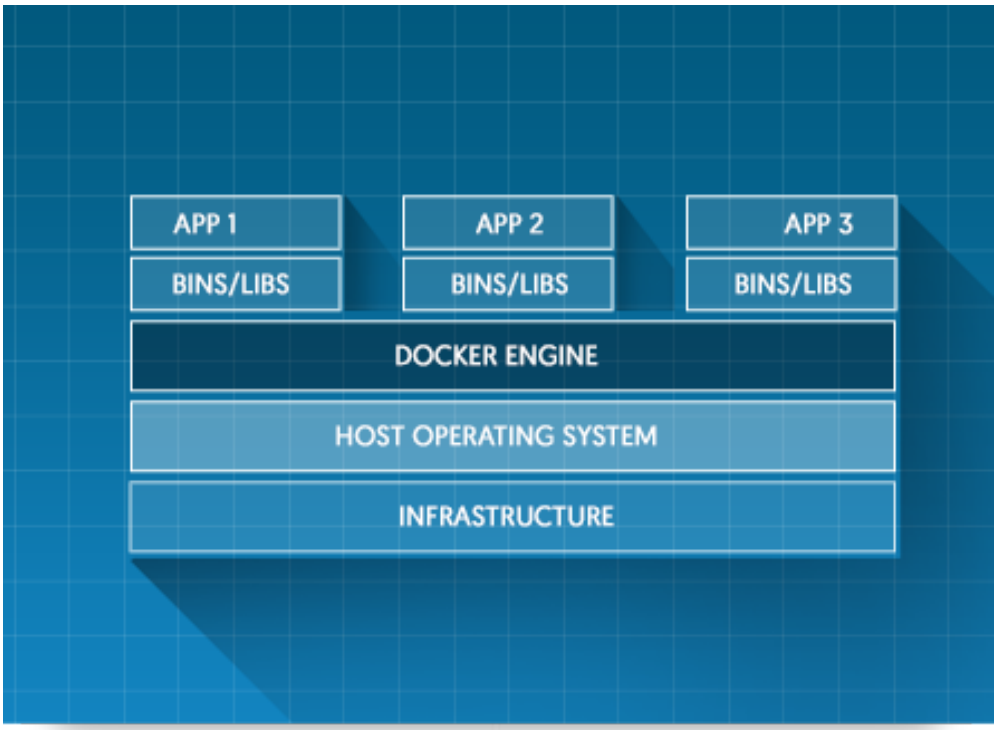

Abbildung 2.2.: Container (Quelle: Docker (2016) *Comparing Containers and Virtual Machines*)

2.4. Docker

Eine Umsetzung des Container basierten Prinzips bietet Docker an. Um Docker nutzen zu können, muss nur auf einem Server oder Entwickler PC die Docker Engine eingerichtet werden. Dies geschieht entweder auf Systemebene oder in einer virtuellen Maschine, je nach Unterstützung des Host-Systems. Eine Übersicht über die Plattformen, auf denen Docker verfügbar ist, gibt es auf der Get Docker Seite.

Jeder Container, der ausgeführt wird, basiert auf einem Image, welches aus einzelnen Ebenen besteht. Diese Ebenen können nur gelesen werden und enthalten die Änderungen am Dateisystem des Grund Images. Beim Erstellen eines Containers erhält dieser eine eigene Ebene zum Lesen und Schreiben, was einen gemeinsamen Zugriff mit anderen Containern auf das gleiche Image erlaubt. Dies ermöglicht eine getrennte Datenhaltung, welche ein konfliktfreies Lesen von den Imageebenen ermöglicht, während das Schreiben auf der Container Ebene stattfindet. Zusätzlich bedeutet es auch, dass die Dateien eines Images nur einmal pro Docker Engine

existieren und erst bei einer Veränderung durch den Container auf die Container Ebene kopiert werden. (vgl. Docker (e))

Neben der Umsetzung dieses Prinzips bietet Docker ab Engine Version 1.12 den Swarm Mode an. Dieser ermöglicht den verteilten Betrieb von Docker auf mehreren Hosts (Nodes) und der Verteilung und Verwaltung von Containern (Tasks) in Form von Diensten (Services). Der Swarm Cluster stellt hierbei eine verteilte Dockerumgebung dar, welche wie eine Umgebung auf einem einzelnen Rechner verwendet werden kann. Auch die manuelle Einrichtung eines komplexen virtuellen Netzwerkes ist obsolet, da diese Funktion bereits integriert ist.[6] Die Nutzung von Docker und dem Swarm Mode ist Anbieter unabhängig, es können somit Dienste auf verschiedene Server Anbieter und Rechenzentren hinweg verteilt und miteinander verknüpft werden. Mit dem im Swarm Mode integrierten Load Balancing werden Anfragen unabhängig davon, auf welchem Host diese ausgeführt werden, zu den einzelnen Containern eines Dienstes geleitet.[7]

Der Aufbau und Betrieb von Entwicklungsumgebungen wird stark durch Docker vereinfacht. Eine einfache Installation der Docker Engine ermöglicht es sowohl die Dienste auf einem lokalen System auszuführen, als auch die dafür benötigten Build Schritte auszuführen. Es muss kaum weitere Software auf dem Host System installiert werden, da viele Schritte in einem Container ausgeführt werden können. Dies vermindert auch Fehlerquellen, welche durch unterschiedliche Versionen der verwendeten Tools auftreten können. Änderungen an den Containern bzw. Scripten zum Erstellen von Containern können einfach über die Versionsverwaltung verteilt werden, sodass alle Entwickler auf dem gleichen Stand sind.

Um diese Umgebungen noch einfacher zu verwalten, bietet Docker das Tool Docker Compose an. Dieses Tool kann anhand einer Konfigurationsdatei alle benötigten Images bauen, Container starten, diese mit einem gemeinsamen Netzwerk verknüpfen und nimmt viele der manuellen Schritte – auch im Bezug auf Benennung von Images und Containern – ab. (Siehe auch Docker (b))

[6] Docker (a)
[7] Docker (d) & Docker (c)

2.5. Spring

Das Spring Projekt setzt sich aus vielen einzelnen Projekten zusammen. Diese bauen auf dem Spring Framework auf, bei dem es sich um ein Java Framework handelt, das grundlegende Funktionen wie Dependency Injection, Spring MVC für Web Anwendungen oder die Unterstützung von Java Database Connectivity (JDBC) bietet. Alle im Laufe der Arbeit genutzten Spring Projekte werden im Abschnitt 4.4 (S. 16) genauer erläutert.

3. Fachliche Analyse

Nachdem die Grundlagen der Technologien behandelt wurden, folgt die fachliche Analyse. Diese wurde – wie bereits in Abschnitt 1.1 (S. 1) erläutert – in der Zusammenarbeit mit Hamburger Modelagenturen erarbeitet. Hierfür wurden in einer ersten Phase die fachlichen Bedingungen für einen Prototypen festgehalten. In weiteren Phasen wird dieser Prototyp um Funktionalität und Anforderungen der Agenturen erweitert, sodass am Ende ein vollwertiges und zufriedenstellendes Produkt entsteht. Um nicht zu sehr ins Detail zu gehen, werden in diesem Kapitel nur wichtige Eigenschaften der eingeführten Entitäten erwähnt.

3.1. Anforderungen

Am Anfang stehen die Benutzer des Systems, welche die Mitarbeiter der jeweiligen Agentur darstellen. Jede Agentur besteht hierbei aus mindestens einem Administrator Konto, welches weitere Benutzer verwalten kann. Diesen Benutzern können verschiedene Rollen zugeteilt werden, um deren Zugriffsrechte zu beschränken. Während der Administrator – meist der Agenturchef – Zugriff auf alle Bereiche hat, dürfen Booker nur bestimmte Bereiche und Funktionen nutzen. Diese Entscheidung muss vollständig beim Administrator liegen, da sich diese Rollenverteilung zwischen den Agenturen unterscheiden kann. Für die Umsetzung des Prototyps reichen einfache Operationen zum Erstellen, Lesen, Ändern und Löschen – auch bekannt als Create, Read, Update, Delete (CRUD) – über eine Oberfläche und die serverseitige Verarbeitung aus. Hinzu kommen die Relationen zwischen den einzelnen Entitäten, wie Agentur und Benutzer oder auch Agentur und Model. Auch die kommenden Elemente benötigen in der Regel nur die CRUD Operationen und Verknüpfungen untereinander.

Nachdem die Konten angelegt sind, müssen die Stammdaten von Models und Kunden (z. B. Name, Anschrift) hinterlegt werden können. Hierbei gilt zu beachten, dass Agenturen auch Models von anderen Agenturen in ihrem Portfolio besitzen. Diese Models dürfen nicht direkt kontaktiert werden, weshalb eine Unterscheidung zwischen eigenen Models und fremden Models existieren muss. Zudem müssen diese Agenturen beim jeweiligen Model hinterlegt

werden, damit der Kontakt zum Model über die Fremdagentur stattfinden kann. Diese Unterscheidung muss auch bei weiteren Anwendungsfällen berücksichtigt werden.

Ein Model kann zudem mehrere Anschriften (Privat, Beruflich, Ausland) besitzen, sowie Bilder und Galerien (bzw. Books). Zusätzlich muss eine Möglichkeit geben sein die Primäradresse festlegen und Bilder mit den Galerien verknüpfen zu können.

Um Kunden verwalten zu können, wird zusätzlich zu den Stammdaten eine Verwaltung von Kontaktpersonen benötigt. Diese Kontakte sollen auch zwischen den Agenturen verschoben werden können, da dies die Datenpflege vereinfacht.

Sind Kunden- und Modeldaten eingepflegt worden, gibt es zwei typische Anwendungsfälle für Booker. Entweder einen allgemeinen Termin für das Model oder einen Kundenauftrag erstellen. Aus einem Kundenauftrag wiederum kann eine Option oder aber nur ein Termin – im Falle eines Castings – für ein oder mehrere Models entstehen. Diese Optionen oder Termine müssen unabhängig voneinander sein, da diese pro Model unterschiedlich sein können. Termine, Aufträge und Optionen enthalten typische Informationen wie Ort, Beginn und Ende.

Sobald ein Model gebucht wird, handelt es sich nicht mehr um eine Option, sondern um eine Buchung. Da die Abrechnung von Aufträgen pro Model stattfindet, müssen die Positionen von Gebühren und entstandenen Kosten (z. B. Taxifahrten) pro Buchung erfasst werden. Hinzu kommen Provisionen für die Agentur, der Anteil, welchen das Model erhält, und weitere Werte.

Nachdem der Auftrag abgeschlossen ist, muss aus der Buchung eine Rechnung für die Buchhaltung erstellt werden können. Diese enthält alle Positionen inklusive der Provisionen.

4. Konzeption

Nachdem die Fachlichkeiten für den Prototypen geklärt sind, wird die Konzeption des Systems betrachtet. Wie bereits im Abschnitt 2.1 (S. 3) erläutert wurde, besitzt ein SaaS System andere Anforderungen als eine beim Kunden betriebene Software. Es besteht die Herausforderung ein System umzusetzen, welches von allen Kunden gemeinsam genutzt wird, sich möglichst wie traditionelle Software bedienen lässt und dabei zuverlässig und schnell arbeitet. Dies erfordert eine permanente Verfügbarkeit, die klare Trennung der Daten einzelner Kunden und gleichzeitig den wirtschaftlichen Betrieb des Gesamtsystems. Des Weiteren muss das System den immer neuen Anforderungen standhalten, da z. B. die Zugriffe bei einer Expansion der Zielgruppe auf einzelne Bereiche des Systems stark steigen können.

4.1. Architektur

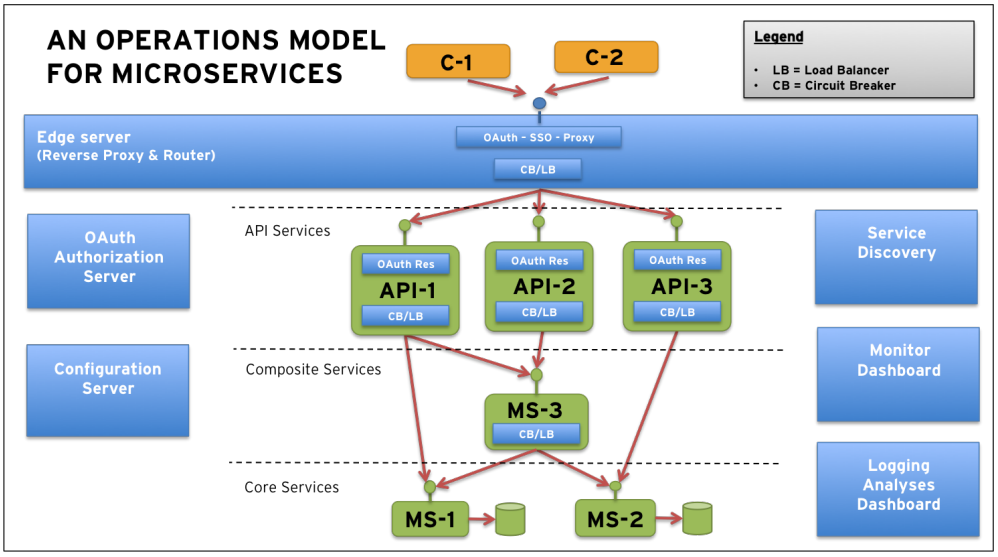

Abbildung 4.1.: Referenzarchitektur (vgl. Larsson (2015) Punkt *5. A REFERENCE MODEL*)

Eine wichtige Überlegung ist die Architektur der Microservices. Als Referenzarchitektur dient ein von Callista Enterprise (siehe Abbildung 4.1 (S. 12)) erstellter Aufbau. Dieser teilt das System in 4 Ebenen auf, welche im Folgenden beschrieben und um Anpassungen ergänzt werden.

Die unterste Ebene beherbergt die Core Services, welche den Kern der Anwendung darstellen. Ein Core Service spiegelt eine Entität wieder und besitzt meist einen Speicher zur Persistenz. Diese Ebene soll jedoch aufgeteilt werden in Core Services und Persistence, sodass die Trennung von Datenhaltung und Schnittstelle klarer dargestellt wird. Alle Instanzen eines Core Service greifen hierbei auf den gleichen Persistenzdienst zu, welcher auch einen Datenbank Cluster darstellen kann. Somit haben alle Instanzen eines Core Services die gleiche Datenbasis, müssen sich jedoch nicht um die Synchronisierung oder Transaktionen der Datensätze kümmern.

Folgend dienen die Composite Services in der Referenzarchitektur als Bindeglied zwischen den verschiedenen Core Services. Zusammen mit Load Balancer und Circuit Breaker werden die Anfragen auf die verschiedenen Instanzen der Core Services aufgeteilt und die erhaltenen Daten dann miteinander verknüpft. Im geplanten System sollen diese in möglichst wenigen Fällen eingesetzt werden, da beispielsweise Relationen in der Regel auf Core Service Ebene behandelt werden können und somit ein erneutes Aufbereiten der Daten entfällt.

Als zweit höchste Ebene sind die Application Programming Interface (API) Services zu sehen. Diese dienen der Vereinigung von mehreren Composite und Core Services zu einer API, welche extern verfügbar und durch OAuth geschützt sein soll. Diese werden nicht in die geplante Architektur übernommen, da sie weniger Nutzen haben, als an Komplexität, Rechenaufwand und Latenz durch ihren Einsatz erzeugt wird.

Die oberste Ebene stellt der Edge Server als Einstiegspunkt von außen dar. Bestehend aus einem OAuth Proxy, Load Balancer und Circuit Breaker wird an dieser Stelle die Authentifizierung und Verteilung auf die Instanzen der nächsten Ebene durchgeführt. Das OAuth Token wird hierbei an die unteren Ebenen übergeben. In der geplanten Architektur wird es in dieser Ebene zwei Gateways und den OAuth Server geben, welche später genauer erläutert werden.

Neben diesen Ebenen befinden sich noch einzelne Dienste, welche keiner speziellen Ebe-

ne angehören, sondern allgemein dem System dienen. Diese können Discovery, Logging oder Configuration Server sein. Dieser Ansatz wird unverändert übernommen.

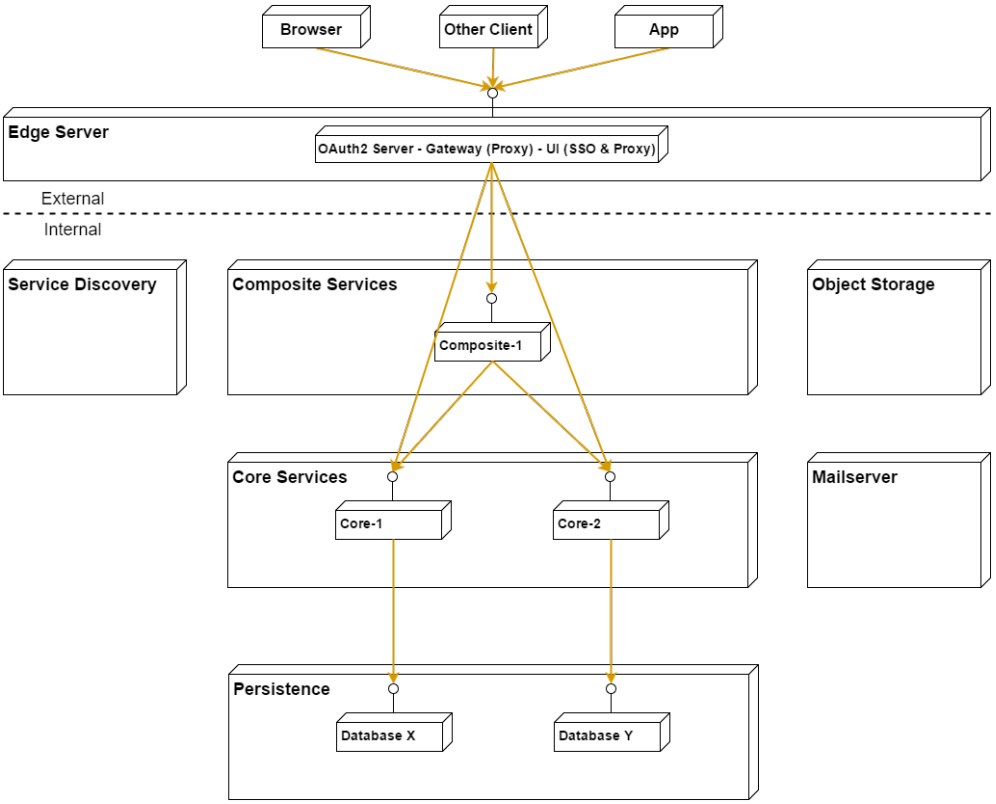

Abbildung 4.2.: Entwurf der Microservice Architektur

Basierend auf der oben gezeigten Architektur und den aufgeführten Änderungen resultiert die Architektur aus der Abbildung 4.2 (S. 14).

Ebene 1 als Edge Server bzw. Gateway beinhaltet einen OAuth2 Dienst, ein API Gateway und einen UI Dienst, welcher als Gateway für die Weboberfläche dient. Diese Aufteilung resultiert hauptsächlich aufgrund folgender Restriktionen:

1. Der Spring OAuth Server konnte zu diesem Zeitpunkt nicht fehlerfrei hinter einem Edge Server (in diesem Fall einem Zuul Proxy) betrieben werden.

2. Webanwendungen im Browser können das Token nicht geheim halten, wodurch ein mögliche Sicherheitslücke entstünde. Stattdessen wird auf einen serverseitigen Proxy gesetzt, welcher das Token in die Anfragen der Oberfläche hinzufügt.

Auf Ebene 2 befinden sich nach Bedarf die Composite Services, welche als OAuth Ressource zu behandeln sind.

In Ebene 3 sind weiterhin die Core Services, welche auch als OAuth Ressource zu behandeln sind.

Die letzte Ebene enthält alle Dienste zur Datenhaltung, welche – wie bereits oben beschrieben – für die einzelnen Core Services zur Verfügung stehen.

Auch bestehen neben diesen Ebenen die allgemein verfügbaren Dienste, wie der Object Storage, Service Discovery oder Mailserver.

Zusätzlich ist zu beachten, dass alle Kundendaten auf einer gemeinsamen Plattform verarbeitet und gespeichert werden. Dies kann mithilfe einer Multi-Tenancy-Architektur sichergestellt werden.

4.2. Kommunikation

Die Kommunikation zwischen den einzelnen Diensten und nach außen findet grundlegend über REST-Schnittstellen statt. Die zu entwickelnden Dienste sollen hierbei auf das HATEOAS Prinzip nach der HAL-Spezifikation setzen. Ausnahmen können z. B. das Advanced Message Queuing Protocol (AMQP) oder Socket basierte Schnittstellen für Echtzeitkommunikation darstellen.

4.3. Zugriff auf Dienste

Eine freie Kommunikation im internen Netzwerk ist jederzeit gegeben, während externe Anfragen durch das API Gateway und den UI Dienst beschränkt sind. Diese verifizieren den anfragenden Client anhand eines OAuth2 Tokens, welches von einer Instanz des Authentifizierungsdienstes abgerufen werden muss. Eine genauere Überprüfung von Berechtigungen findet auf Core und Composite Ebene statt.

4.4. Spring als Backend

Für die Umsetzung der Microservices stehen eine Vielzahl von Frameworks zur Verfügung. Da die einzelnen Dienste unabhängig voneinander agieren und über REST-Schnittstellen kommunizieren, kann in der Theorie jeder Dienst eine andere Implementationssprache und geeignete Bibliotheken und Frameworks wählen. Das System soll sich jedoch zu Beginn auf ein Framework und eine Sprache konzentrieren, um eine bessere Wartbarkeit zu garantieren.

Einige bekannte Frameworks stellen hierbei **Spring (Java)**, **Play (Scala / Java)**, **Laravel** oder **Meteor (Javascript / Node.js)** dar. Wichtige Faktoren bei der Wahl eines Frameworks sind unter anderem der aktuelle Entwicklungsstand des Projektes, die weitere Entwicklung, die Größe der Community, mitgebrachte Funktionalitäten in Bezug auf die Anforderungen – wie SaaS, Microservices, verteilte Systeme –, sowie die Performanz des resultierenden Systems.

Alle genannten Frameworks bieten viele der genannten Punkte, jedoch besitzt das Spring Framework eine Großzahl an existierenden Lösungen für die Anforderungen eines SaaS Systems. Spring bietet hierbei unter anderem Lösungen für den Bereich OAuth2, Datenbanken, REST und HATEOAS (nach HAL). Auch für verteilte Systeme gibt es sowohl passende und in der Praxis erprobte Lösungen von Spring als auch Integrationen von externen Projekten – beispielsweise **Netflix OSS**.

Vom Spring Projekt werden zur Implementation unter anderem die folgenden Projekte genutzt:

Spring Boot Vereinfacht die Erstellung von alleinstehenden Anwendungen, indem viele Teile der Konfiguration, sowie weitere zum Start der Anwendungen benötigten Schritte automatisiert werden.

Spring Security Ermöglicht die Authentifizierung und Autorisierung in einer Anwendung und schützt vor gängigen Attacken, wie Cross Site Request Forgery (CSRF).

Spring OAuth Erweitert Spring Security um OAuth (1a) und den OAuth2-Standard auf Client- und Serverseite.

Spring Data Bietet abstrahierte Arbeit mit Datenbanken z. B. über einheitliche Repositories unabhängig von der Datenbank und objektrelationale Abbildung.

Spring Data JPA Ein Modul zur Erweiterung von Spring Data. Ermöglicht die Erstellung von Entitäten und Repositories basierend auf Java Persistence API (JPA). Nutzt hierbei

standardmäßig als Implementation Hibernate. Dies ermöglicht Objekt-Relational Mapping (ORM), dynamische Anfragen, sowie eine bereits verfügbare Unterstützung der Paginierung.

Spring Data REST Ein Modul zur Erweiterung von Spring Data. Erweitert Repositories automatisiert um eine REST-Schnittstelle, welche dem HAL-Standard folgt.

Spring Cloud Netflix Bietet die Integration von Netflix OSS Projekten in Spring Boot Anwendungen. Unter anderem werden die Projekte Eureka – ein Service Discovery Server vom Netflix OSS – und Zuul – ein Reverse Proxy vom Netflix OSS – genutzt.

4.5. Ember.js als Frontend

Passend zu dem Backend ist auch ein entsprechendes Frontend in Form einer Webanwendung vorgesehen. Diese Oberfläche soll sich flüssig und interaktiv anfühlen, sodass auch hier ein passendes Framework benötigt wird. Zur Lösung existieren verschiedene Javascript Frameworks, welche auf unterschiedlichen Ansätzen basieren und teils unterschiedliche Ziele verfolgen. Einige bekanntere sind hierbei **AngularJS**, **Ember.js**, **Backbone.js** oder auch **React**. Auf einen genaueren Vergleich wird verzichtet, da das Frontend ein Nebenprodukt dieser Arbeit darstellt und nur im API Kontext genauer auf die Anpassungen des verwendeten Frameworks eingegangen wird. Jedoch gilt auch an dieser Stelle, wie bereits bei den Frameworks für das Backend, dass Faktoren wie Aktivität, Community und Funktionalität wichtige Faktoren bei der Wahl des Frameworks sind. Aufgrund der gebotenen Funktionen, der Popularität und weiteren Aspekten fiel die Entscheidung auf *Ember.js*.

Ember.js bietet auf eine einfache Weise die Möglichkeit komplexere Anwendungen mit HTML und Javascript zu realisieren. Diese setzen sich durch einen Router, Route Handlern, Models, Templates und einzelnen Komponenten zusammen. Eine grobe Übersicht über diese Kern Konzepte bietet Grafik 4.3 (S. 18). Weitere Vorteile bieten die Anbindung an REST-Schnittstellen, die Nutzung von Promises zur Handhabung von Asynchronität, sowie viele vorhandene Erweiterungen und Komponenten aus der Community zur Realisierung der Oberfläche.

4.6. Benötigte Dienste

Für die Umsetzung müssen einige allgemein als auch fachlich benötigte Dienste realisiert werden. Im Folgenden werden die wichtigsten Dienste kurz aufgeführt und in ihrer Funktion

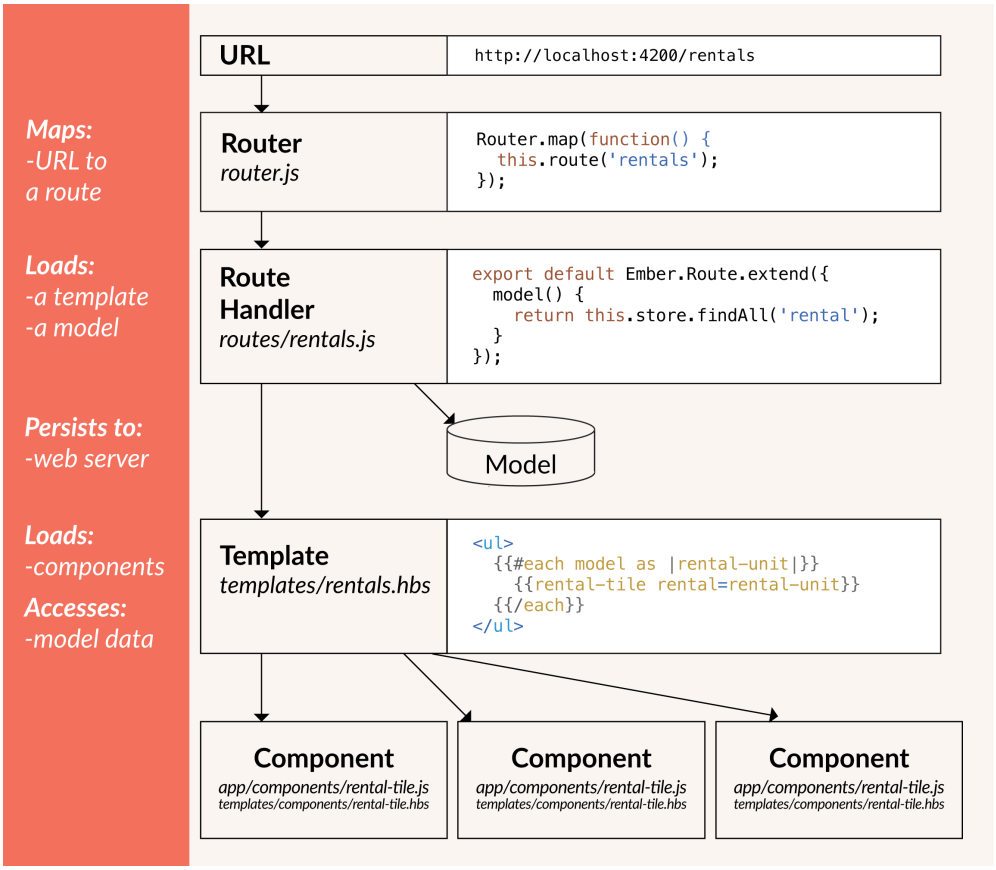

Abbildung 4.3.: Kernkonzepte von Ember.js (Quelle: Ember.js (2016))

beschrieben. Alle selbst zu entwickelnden Dienste haben, wie bereits im Abschnitt 4.2 (S. 15) genauer beschrieben, eine HAL basierte Schnittstelle anzubieten.

Agencies Dienst zur Verwaltung von Agenturen und agenturweiten Einstellungen.

Accounts Dienst zur Verwaltung von Benutzerkonten und benutzerspezifischen Einstellungen.

Models Dienst zur Verwaltung von Models.

Customers Dienst zur Verwaltung von Kunden.

Appointments Dienst zur Verwaltung von Terminen.

Subscriptions & Subscription Check Dienst zur Verwaltung von den Agentur Abonnements, welcher regelmäßig die Gültigkeit der Abonnements überprüft. Wenn ein Abonne-

ment ausgelaufen ist, so werden alle Benutzerkonten der jeweiligen Agentur deaktiviert. Durch eine "Renew" Funktion kann das Abonnement verlängert und die Konten wieder aktiviert werden.

Mailing Dienst zum E-Mail Versand. Eine Queue ermöglicht die zentrale Verwaltung vom Mailversand und ein asynchrones (somit nicht blockendes) versenden von E-Mails. Als Queue soll RabbitMQ dienen und via AMQP sowohl gefüllt als auch abgearbeitet werden. Die E-Mails werden bereits vorgefertigt (Absender, Empfänger, Betreff, Inhalt usw.) in die Queue eingetragen, sodass der Mailingdienst rein für den Versand via SMTP zuständig ist.

Authentication Dienst, welcher Benutzer authentifiziert. Soll eine Single Sign On (SSO) Schnittstelle, OAuth2 und ein Registrierungsformular anbieten. Via SSO und OAuth2 sollen alle Anwendungen die Identität des Benutzers feststellen können und Zugriff auf die jeweiligen API Bereiche erhalten. Das Registrierungsformular legt neue Agenturen inklusive eines Abonnements und eines Benutzerkontos in den jeweiligen Diensten an.

Gateway Dienst mit Schnittstelle nach außen, welche zu den einzelnen internen Diensten weiterleitet (Accounts, Models usw.). Hierzu wird die Identität des Benutzers anhand eines OAuth2 Tokens beim Auth Service überprüft. Nur wenn die Identität bestätigt ist, darf die Anfrage zu den internen Diensten weitergeleitet werden.

Object Storage Stellt verteilten Speicher zur Verfügung, welcher sowohl von den Diensten genutzt werden kann, als auch via HTTP GET Anfrage die öffentlich hinterlegten Dateien ausliefert. Dieser wird nach der **Amazon S3 API** alle Operationen zur Verfügung stellen und ist somit mit allen S3 Clients kompatibel. Zudem können sowohl Amazon S3 Instanzen als auch Lösungen von anderen Anbietern genutzt werden. In diesem Fall soll die Open Source Lösung **Minio** eingesetzt werden, welche sowohl mehrere Clientbibliotheken, als auch ein Docker Image bereitstellt. Dies ermöglicht eine einfache Einrichtung und ist für den Prototypen ausreichend. Für den Produktivbetrieb kann Minio nach Bedarf durch andere S3 kompatible Lösungen ausgetauscht werden.

UI Dienst, welcher die Weboberfläche für die Agenturen ausliefert und als SSO Client dient. Der Nutzer wird zum Authentifizierungsdienst geleitet, gibt seine Anmeldedaten ein und wird bei Eingabe gültiger Daten zurück zum UI Dienst geleitet. Hierbei werden Codes ausgetauscht, mit denen sich der UI Dienst in Verbindung mit einer App ID und dem App Secret ein OAuth2 Token vom Authentifizierungsdienst holen kann. Die mit Ember.js erstellte Oberfläche ruft die UI eigenen API Endpunkte auf. Der UI Dienst fungiert hierbei

als Proxy, welcher alle Anfragen an die jeweiligen Services weitergegeben. Hierbei wird jeder Anfrage das OAuth2 Token im Authorization Header hinzugefügt. Dies stellt sicher, dass sowohl die App ID als auch App Secret interne Geheimnisse bleiben und nur von autorisierten Anwendungen zur Authentifizierung genutzt werden können. Aus einer clientseitigen Anwendung im Browser könnten diese Werte einfach ausgelesen und für Anfragen an den Authentifizierungsdienst missbraucht werden.

Datenbanken Sowohl die Entitäten als auch andere Informationen müssen persistent gespeichert werden. Hierfür kommen mehrere Datenbanken zum Einsatz. Da es sich um eine Microservice Architektur handelt, ist es völlig offen welche Datenbank von den Entitäten genutzt wird. Es können je nach Anforderungen zum Beispiel relationale, aber auch NoSQL basierte Datenbanken genutzt werden. Aufgrund der fortgeschrittenen Implementation und bereits vorhandenen Lösungen des Spring Frameworks, soll jedoch hauptsächlich auf MySQL als relationale Datenbank gesetzt werden.
Der Authentication Dienst besitzt neben einer MySQL Instanz zur Verwaltung der Tokens eine Redis Instanz. Diese dient der Verwaltung von Sitzungen und stellt diese allen Instanzen des Dienstes zur Verfügung, was eine Einsparung von Sticky Sessions ermöglicht. Auch der UI Dienst setzt auf Redis für die Sitzungsverwaltung. Als globale Message Queue für verschiedene Anwendungszwecke wird auf RabbitMQ gesetzt.

5. Realisierung

In diesem Kapitel wird die Umsetzung wichtiger Komponenten des zuvor konzipierten SaaS Systems erläutert und auf die dabei aufgetretenen Hürden mit den eingesetzten Technologien eingegangen. Codebeispiele werden in diesem Kapitel entweder Inline erklärt oder befinden sich im Anhang A (S. 37). Alle Java-Beispiele enthalten der Übersichtlichkeit halber keine Getter oder Setter für Felder, auch wenn diese von außen verändert werden dürfen.

5.1. Entitäten Dienste

Die Realisierung der einzelnen Microservices für Entitäten ist einfach gelöst. Mit nur wenig Aufwand stehen eine REST-Schnittstelle nach dem HAL-Standard, eine Zugriffskontrolle über OAuth2 und ein CRUD-Interface für die Entität zur Verfügung. Diese simple Umsetzung wird unter anderem durch die Projekte Spring Boot, Spring Data JPA, Spring Data REST, Spring Security und Spring Security OAuth ermöglicht. Details zu den Projekten sind in Abschnitt 4.4 (S. 16) zu finden.

Zum Beginn wird eine Main-Klasse (in diesem Fall *Application.java*) benötigt, welche wie folgt aufgebaut ist:

```
1 @SpringBootApplication
2 @EnableResourceServer
3 @EnableJpaRepositories
4 public class Application {
5     public static void main(String[] args) {
6         SpringApplication.run(Application.class, args);
7     }
8 }
```

Mit der Annotation `@SpringBootApplication` wird die automatische Konfiguration und die Durchsuchung nach Komponenten durch Spring Boot aktiviert. Dies bezieht sich sowohl auf selbst definierte Komponenten als auch auf weitere verfügbare Komponenten aus Abhängigkeiten. Die Zugriffskontrolle via OAuth2 wird durch `@EnableResourceServer`

aktiviert, welche in der Standardkonfiguration den Zugriff nur noch mit einem gültigen OAuth2 Bearer Token erlaubt. Des Weiteren sind Informationen über den Benutzer verfügbar, welche für den Dienst relevant sein könnten. Als letzte Annotation aktiviert `@EnableJpaRepositories` die Funktion, dass Package weit nach Repositories gesucht wird und diese automatisch verarbeitet werden.

Darauf folgend werden eine Entität, welche ein Plain Old Java Object (Pojo) mit zusätzlichen Annotationen ist, und das Repository benötigt. Eine Entität kann wie folgt implementiert werden:

```
1  @Entity
2  @Table(name = "agencies")
3  public class Agency {
4
5      @Id
6      private Integer id;
7
8      @NotNull
9      private String name;
10
11     public Agency() {}
12 }
```

Die Annotation `@Entity` spezifiziert, dass es sich um eine Entität handelt. Mit `@Table` können optional weitere Informationen über die Tabelle gegeben werden. In diesem Fall, dass die Tabelle den Namen `agencies` trägt. Das ID Feld, welches nicht zwingend den Namen id tragen muss, muss mit `@Id` annotiert werden. Für Felder, die keine Null Werte enthalten dürfen, kann die Annotation `@NotNull` genutzt werden.

Nachdem die Entität deklariert wurde, kann mit nur wenigen Zeilen Code das dazugehörige Repository in Form eines Interfaces erstellt werden:

```
1  @RepositoryRestResource(collectionResourceRel = "agencies", path = "
       agencies")
2  public interface AgenciesRepository extends
       PagingAndSortingRepository<Agency, Integer> {}
```

Grundlegend sind für Spring Boot zwar alle Repositories eine `@RepositoryRestResource`, jedoch können mithilfe dieser Annotation der Name der Ressource und der Pfad gesteuert werden. Die zweite Zeile definiert nur ein Interface, welches das `PagingAndSortingRepository`

von Spring Data erweitert. Hierbei bestimmt der erste Generic Parameter die Entität `Agency` und der zweite den Typen des Identifiers `Integer` von der Entität.

Um Konflikte bei der Vergabe von IDs im verteilten System zu verhindern, können Universally Unique Identifier (UUID) genutzt werden. Das ID-Feld einer Entität sieht dann wie folgt aus:

```
@Id
@GeneratedValue(generator = "uuid2")
@GenericGenerator(name = "uuid2", strategy = "uuid2")
@Column(columnDefinition = "BINARY(16)")
private UUID id;
```

Die Annotationen `@GeneratedValue` und `@GenericGenerator` definieren hierbei, dass und wie automatisch eine UUID generiert werden soll. Während durch `@Column` der Persistenz Schnittstelle übergeben wird, welche Definition die Spalte für dieses Feld besitzen muss.

Einen weiteren Sonderfall stellen `DateTime` Variablen dar. Da die standard `Date` -Klasse von Java nicht sehr umfangreich ist, arbeitet Spring vorwiegend mit der `DateTime` -Klasse von Jadira. Dies ermöglicht vor allem in der weiteren Entwicklung einen angenehmeren Umgang mit Datum und Uhrzeit. Ein korrekt definiertes Feld in einer Entität sieht wie folgt aus:

```
@Type(type = "org.jadira.usertype.dateandtime.joda.
    PersistentDateTime")
private DateTime birthday;
```

Die `@Type` Annotation ermöglicht es JPA – bzw. Hibernate –, dass der `DateTime` Wert korrekt in die Datenbank eingetragen und auch wieder ausgelesen werden kann.

Nachdem die Main-Klasse den Webserver startet, alle Komponenten initialisert und das Repository nach außen eine HAL konforme REST API für die Entität zur Verfügung stellt, fehlt nur noch die Konfiguration der Anwendung. Diese wird in die Dateien *bootstrap.yml* und *application.yml* im *resources* Ordner geschrieben. Eine Konfiguration des Dienstes wird wie folgt aufgebaut – *bootstrap.yml* und darauf die *application.yml*) –:

```
spring:
  application:
    name: models
```

```
security:
  oauth2:
    resource:
      userInfoUri: http://auth:9999/uaa/user
  sessions: stateless

spring:
  datasource:
    url: jdbc:mysql://models-db/models?useSSL=false
    username: models
    password: foo
    test-while-idle: true
    time-between-eviction-runs-millis: 900000
    validation-query: SELECT 1

eureka:
  client:
    service-url:
      defaultZone: http://eureka:8761/eureka/
```

Der Wert `models` in der *bootstrap.yml* wird für Eureka benötigt, um den Namen des Dienstes mitteilen zu können. Die Security Konfiguration setzt den OAuth2 Endpunkt, welcher gegen das Bearer Token die Informationen über den Benutzer zurückgibt. Der Wert für `sessions` wird auf `stateless` gesetzt, um die Zustandslosigkeit des Dienstes zu garantieren, indem niemals eine Session durch Spring erstellt wird. Die `datasource` Konfiguration beinhaltet grundlegend eine normale MySQL Konfiguration. Allerdings wird nach gewisser Inaktivität die Verbindung durch den MySQL Server beendet. Dies kann durch die `test-while-idle` Option verhindert werden. In der Konfiguration wird als Abfrage `SELECT 1` (siehe `validation-query`) definiert, welche im Falle der Inaktivität alle 15 Minuten (siehe `time-between-eviction-runs-millis`) ausgeführt werden soll. Zuletzt muss die Url zu Eureka in `eureka.client.service-url.defaultZone` angegeben werden.

Zusätzlich zu diesen Schritten können noch weitere Anpassungen durchgeführt werden. Kurz zusammengefasst handelt es sich um folgende Schritte:

1. Um die interne Kommunikation zu erleichtern, kann über die Java-Konfiguration mit dem `ResourceServerConfigurerAdapter` jegliche Anfrage anonym erlaubt werden. Da externe Zugriffe nur durch ein Gateway möglich sind, besitzen diese zwingend ein

Token und es kann bei einem anonymen Zugriff von einem internen Dienst ausgegangen werden.

2. Mit dem `RepositoryRestConfigurerAdapter` können Validatoren vor dem Erstellen und Speichern von Entitäten ausgeführt werden. Dies ermöglicht eine ausführliche Prüfung der Benutzereingaben.

3. Zur Unterstützung von der Multi-Tenancy-Architektur, sollte eine eigene Repository Implementation genutzt werden. Dies wird im folgenden Abschnitt genauer erläutert.

5.2. Multi Tenancy

Um mehrere Kunden auf der gleichen Plattform zu unterstützen, müssen die Daten voneinander getrennt werden. In diesem Fall gilt dies meist pro Agentur, teilweise pro Benutzer und nur selten gibt es Entitäten für alle Agenturen. Dieser Abschnitt bezieht sich hierbei bewusst nur auf den ersten Fall, da dieser nicht von Spring Data unterstützt wird, jedoch für Multi Tenancy Architekturen zwingend notwendig ist.

Zu Beginn benötigt jede Entität, auf die Fall 1 zutrifft, ein Feld für die Agentur-ID. Der Variablen Typ ist hierbei irrelevant, in diesem Fall wird eine `UUID` genutzt. Hingegen ist der Name des Feldes wichtig, da dieser über alle gefilterten Entitäten hinweg identisch sein muss. Hier wird im weiteren Verlauf der Name `agency` verwendet.

Folgend wird ein Interface für das eigene Repository definiert:

```
1 @NoRepositoryBean
2 public interface AgencyFilteredRepository<T, ID extends Serializable
      > extends PagingAndSortingRepository<T, ID>,
      JpaSpecificationExecutor<T>
3 {
4 }
```

In Zeile 1 wird mit `@NoRepositoryBean` verhindert, dass Spring Data Rest versucht dieses Repository zu verarbeiten, um einen Endpunkt für die Schnittstelle zu generieren. Daraufhin wird das Interface unseres Repositories definiert (vgl. Zeile 2). Dieses erweitert das für unsere Entitäten grundlegende `PagingAndSortingRepository` und zusätzlich den `JpaSpecificationExecutor` , welcher weitere Anfragemöglichkeiten für das Repository zur Verfügung stellt. Jedes Repository, welches zurzeit `PagingAndSortingRepository`

erweitert und gefiltert werden soll, muss stattdessen das neue `AgencyFilteredRepository` Interface erweitern.

Nachdem die Grundlage geschaffen wurde, folgt die Implementation für das eigene Repository (siehe Anhang A.1 (S. 38)). Eine genaue Beschreibung ist in dem genannten Anhang in Form von Kommentaren zu finden. Diese Implementation modifiziert alle Datenbankanfragen, indem sie zusätzlich um die Agentur filtert. Wenn kein Bearer Token vorhanden ist, so wird davon ausgegangen, dass es sich nicht um einen Benutzer, sondern eine interne Anfrage handelt. In diesem Fall wird nicht nach der Agentur gefiltert. Es besteht somit eine interne Übersicht aller Daten, während Anfragen von außen nur mit gültigem Token möglich sind und anhand des Benutzers gefiltert werden.

Zuletzt wird eine eigene Factory für Repositories benötigt. Diese verweist auf die angepasste Implementation, falls ein Repository `AgencyFilteredRepository` erweitert. Dies geschieht mit einer eigenen Factory Bean (siehe Anhang A.2 (S. 42)) und der Anpassung der zuvor erstellten Main-Klasse, indem `@EnableJpaRepositories` um Folgendes ergänzt wird:

```
1 @EnableJpaRepositories(repositoryFactoryBeanClass =
     AgencyFilteredRepositoryFactoryBean.class)
```

Diese Variante ermöglicht es mit einem Feld in der Entität, sowie der Angabe nur eines Interfaces alle Token enthaltenden Anfragen an das Repository zu filtern. Sollte dieses Verhalten nicht gewünscht sein, können weiterhin die von Spring mitgelieferten Interfaces genutzt werden, anstatt um `AgencyFilteredRepository` zu erweitern.

Es besteht jedoch eine Einschränkung, welche das Definieren von eigenen Queries im Interface vom Repository betrifft. Spring Data bietet die Möglichkeit eigene Methoden zu deklarieren, welche dann anhand des Methodennamens in eine Datenbankabfrage umgewandelt werden können. Als Beispiel diene folgende im Interface deklarierte Methode:

```
1 public List<Thing> findByEmailContainingAsc(String email);
```

Diese würde alle `Thing` zurückgeben, dessen Feld `email` den Wert vom Parameter `email` enthält und diese Liste auf Datenbankebene nachdem diesem Feld aufsteigend sortieren. Hierbei wird jedoch die Repository Implementation umgangen und somit sind diese Anfragen ungefiltert. Im Multi Tenancy Bereich ist diese Filterung wichtig, weshalb eigene Controller implementiert werden müssen und somit ein Vorteil von Spring Data verloren

geht. Dies wirkt sich auch auf die automatisierte HATEOAS Aufbereitung und somit auf die Schnittstellennavigierbarkeit aus.

5.3. OAuth2 Server

Wie bereits im Konzeptions Kapitel erwähnt, findet die Authentifizierung über OAuth2 statt. Hierfür wird sowohl ein eigener OAuth2 Server als auch die Clientimplementation in den Diensten und Edge Servern benötigt. Auch hier bietet Spring sowohl eine Lösung für den Server als auch für die Dienste und SSO-Clients an.

Zu Beginn muss der OAuth2 Server inklusive Spring Security Login nach dem OAuth2 Developers Guide umgesetzt werden. Hinzu kommen Anpassungen bezüglich der Benutzerdaten. Diese stammen nicht aus einer lokalen Datenbank, sondern befinden sich beim Accounts Service. Es müssen angepasste Implementation vom `AuthenticationProvider` und dem `UserDetailsService` entwickelt werden, welche via REST-Schnittstelle die Daten erhalten. Eine Beispielimplementation für den `AuthenticationProvider` ist im Anhang A.3 (S. 44), für den `UserDetailsService` in Anhang A.4 (S. 47) zu finden. Beide enthalten genauere Erläuterungen in der Form von Kommentaren.

5.4. OAuth2 Gateway

Daraufhin wird für den Zugriff von außen das Gateway benötigt, welches ein gültiges OAuth2 Token voraussetzt. Dieses ist mit dem Zuul Proxy vom Netflix Opens Source Projekt und dem Spring OAuth Projekt schnell realisiert. In einer Spring Boot Applikation müssen die Annotationen `@EnableZuulProxy` und `@EnableResourceServer` gesetzt werden. Zusätzlich sollten Zuul und der Server wie in Anhang A.5 (S. 50) beschrieben konfiguriert werden. Hierbei fließen auch Erfahrungswerte von weiteren Szenarien mit rein, welche während der Entwicklung auftraten. Genauere Informationen zu den einzelnen Konfigurationsvariablen sind als Kommentar im Quelltext zu finden. Nach diesen Schritten ist das Gateway einsatzbereit und nimmt Anfragen mit einem gültigen OAuth2 Token im Authorization Header (`Authorization: Bearer $TOKEN`) entgegen.

5.5. OAuth2 SSO

Ähnlich wie schon das Gateway wird auch ein SSO Client umgesetzt. Anstatt mit `@EnableResourceServer` ein OAuth2 Token zu verlangen, kann mit `@EnableOAuth2Sso` das SSO Verfahren aktiviert werden. In Verbindung mit dem Zuul Proxy wird jeder Anfrage das vom OAuth2 Server erhaltene Token hinzugefügt. Grundlegend ist hierfür die Konfiguration identisch mit dem Gateway. Eine Beschreibung der SSO spezifischen Anpassungen an der Konfiguration ist in Anhang A.6 (S. 51) zu finden.

5.6. Object Storage

Zur Umsetzung des Object Storage wird das Minio Docker Image genutzt. Eine Amazon S3 API kompatible Bibliothek zur Anbindung des Object Storage an einen Microservice steht von Minio[1] zur Verfügung. Das Code Beispiel in Anhang A.7 (S. 52) zeigt auf, wie ein Bucket erstellt, die Policy verändert und eine Datei hochgeladen werden kann. Zudem wird auch der Vorgang zum Erstellen eines temporären Download Links beschrieben.

5.7. Ember.js Anpassungen

Wie eine Ember.js Applikation erstellt wird, kann in dem Ember.js Guide nachgelesen werden. In diesem Abschnitt wird erläutert, wie Ember.js mit der HAL-Schnittstelle und Spring kompatibel gemacht werden kann. Standardmäßig setzt Ember.js auf die Ember Data Bibliothek zur Persistenz. Diese besitzt bereits allgemeine REST und JSON-API Adapter und Serializer. Während der Entwicklung sind ein funktionierender Adapter und Serializer zur Anbindung an eine HAL-Schnittstelle entstanden, welche jedoch noch nicht komplett dem Standard entsprechen und auf der JSON-API Implementation basieren. Beim Serializer handelt es sich um eine angepasste Version des Ember-data-hal-9000[2] Projekts. Dieser ist in Anhang A.10 (S. 57) zu finden, sowie der Adapter in Anhang A.11 (S. 71). Beide enthalten eine genauere Erläuterung in der Form von Kommentaren.

5.8. Docker

Docker bietet durch die Containerisierung eine Plattform für den Betrieb der einzelnen Dienste. Dies ermöglicht es, dass Docker als Entwicklungsumgebung genutzt werden kann und eine

[1] https://github.com/minio/minio-java
[2] https://github.com/201-created/ember-data-hal-9000

Umgebung für den Betrieb im Test- und Produktivsystem bietet. In den folgenden zwei Abschnitten wird der Betrieb beider Anwendungsfälle grundlegend anhand von verschiedenen Docker Werkzeugen erläutert.

5.8.1. Entwicklungsumgebung

Vor dem Betrieb in einem Test- oder Produktivsystem steht die Entwicklung. Neben dem reinen Betrieb der Dienste kann Docker auch als Build System eingesetzt werden. Dies ermöglicht somit nicht nur eine Konsistenz im Betrieb, sondern auch beim Bauen von Software auf unterschiedlichen Geräten. An dieser Stelle wird nur der Betrieb der Dienste erläutert. Die Erweiterung um ein Build System in Docker sollte durch die grundlegenden Prinzipien selbsterklärend sein.

Zur simplen Lösung bietet sich das bereits im Abschnitt 2.4 (S. 7) erwähnte Tool Docker Compose an. Genauere Erläuterungen zu den kommenden Arbeitsschritten sind in der Dokumentation[3] von Docker Compose zu finden. Als Beispiel wird der Betrieb eines Dienstes und der dazugehörigen Datenbank betrachtet.

Für den Dienst wird ein Dockerfile (Anhang A.8 (S. 56)) benötigt, welches Instruktionen für den Bau des Images bietet. Als Datenbank soll ein MySQL Server zum Einsatz kommen, für diesen existiert bereits ein einsatzfähiges Docker Image[4]. Zuletzt wird noch eine Konfigurationsdatei für Docker Compose benötigt (Siehe **docker-compose.yml** in Anhang A.9 (S. 56)). Die Konfigurationsdatei sollte sich im Root Verzeichnis des Projektes befinden, das Dockerfile im jeweiligen Unterverzeichnis des Dienstes.

Nachdem die Konfiguration abgeschlossen ist, kann beispielsweise mit Gradle `gradlew accounts:installDist` die benötigte Jar generiert werden. Es wird von einer kompatiblen Build Konfiguration ausgegangen, welche auch alle Abhängigkeiten in ein Verzeichnis mit dem Namen *lib* kopiert. Nach dem Build der Jar, können mit dem Befehl `docker-compose up -d --build` alle benötigten Container gestartet werden. Die Option `-d` lässt alle Container im Hintergrund starten (detached), während die Option `--build` den Build der Images erzwingt. Sollte es Änderungen am Build vom Dienst geben oder noch kein Image existieren, so wird das Image neu erstellt bzw. aktualisiert. Ohne die Option `--build` würden nur fehlende Images erzeugt werden.

[3] https://docs.docker.com/compose/
[4] https://hub.docker.com/_/mysql/

Um das von Compose erstellte System zu löschen, kann der Befehl `docker-compose down` genutzt werden. Dieser löscht alle Container, Netzwerke und weitere von Compose erstellten, zum System gehörigen, Elemente. Falls auch Volumes genutzt werden, so können diese mit dem Schalter `-v` explizit gelöscht werden. Um einen Datenverlust zu vermeiden, werden die Volumes standardmäßig nicht mit entfernt.

Einzelne Dienste können mit `docker-compose up -d [DIENST] [...]` gestartet werden. Der Schalter `--build` würde wieder den Build der Images veranlassen. So können nach einer Änderung – **Hierbei sei zu beachten, dass zuvor ein externer Build ausgeführt werden muss!** – einzelne Dienste oder auch das gesamte System über `docker-compose up -d --build` aktualisiert werden. Container werden hierbei mit neueren Versionen gestartet, soweit Änderungen vorhanden sind. Alle anderen Container bleiben unberührt, dies gilt auch, wenn `docker-compose up` ohne Angabe von Diensten durchgeführt wird. Die im Compose File verknüpften Dienste (wie Datenbanken) werden von Docker Compose automatisch unter Berücksichtigung der Abhängigkeiten gestartet. Dies gilt sowohl beim Start aller als auch von einzelnen Diensten.

5.8.2. Test- und Produktivsystem

Nachdem mit Docker Compose im Entwicklungssystem die Software betrieben und getestet worden ist, soll die Software auch in einem Test- oder Produktivsystem, bestehend aus mehreren Servern, in Betrieb genommen werden. Docker Compose ermöglicht die einfache Verwaltung mehrerer Container auf einem einzelnen System, jedoch nicht verteilt auf mehrere Docker Hosts. Hierfür bietet Docker den Swarm Mode an, welcher bereits im Abschnitt 2.4 (S. 7) eingeführt wurde. Inzwischen besteht zusätzlich die Möglichkeit aus Docker Compose Systemen sogenannte Bundles zu generieren und diese als Stacks zu betreiben. In diesem Abschnitt wird jedoch die ältere und manuelle Variante direkt über die Swarm Befehle erläutert.

Swarm kann zu Beginn mit nur einem Server genutzt werden. Es besteht grundlegend keinen Unterschied, ob der Swarm aus nur einem oder aus mehreren Knoten besteht. Genauere Erläuterungen zu dem Betrieb mit mehreren Nodes und mehreren Managern, sowie zu den genutzten Befehlen sind auch hier in der Dokumentation von Docker[5] zu finden.

Zu Beginn muss ein Swarm initialisiert werden. Dafür wird über die Konsole vom Docker

[5] https://docs.docker.com/engine/swarm/swarm-tutorial/

Host der Befehl `docker swarm init` ausgeführt. Hierbei wird ein Befehl ausgegeben, welcher zum Hinzufügen von weiteren Nodes genutzt werden kann. Dieser wird nur zum Einrichten eines Swarms mit mehreren Servern benötigt. Der aktuelle Zustand von Swarm kann über `docker info` abgerufen werden, grundlegende Informationen, wie Hostname, Verfügbarkeit oder Manager Status, von allen Nodes können via `docker node ls` ausgegeben werden.

Nachdem der Swarm bereit ist, müssen die Dienste (im Weiteren als Services bzw. Service bezeichnet) erstellt werden. Hierfür werden wie bei Docker Compose die Images benötigt. Diese kommen entweder aus einer Docker Registry – dem Docker Hub von Docker oder selbst gehostet – oder müssen lokal vorliegen. Dafür kann entweder die Build Funktion von Docker Compose genutzt werden `docker-compose build [DIENST]` , welche das Image unter anderem direkt benennt oder der standard Docker Befehl `docker build` [6] mit den entsprechenden Optionen. Im Produktiveinsatz sollte ein Repository für die Images genutzt werden, sodass jede Node sich die fertigen Images aus dem Repository herunterladen kann. Alternativ müsste auf jedem Docker Host der Quellcode verfügbar sein und ein Build aller eigenen Images manuell durchgeführt werden.

Sind die Images verfügbar, so wird noch ein gemeinsames Netzwerk benötigt. Dieses kann mit `docker network create --driver overlay my-network` erstellt werden, wobei der Wert `overlay` für von der Option `--driver` wichtig ist, da Swarm nur mit Overlay Netzwerken arbeitet. Daraufhin kann z. B. für die MySQL Datenbank ein Service mit dem folgenden Befehl erstellt werden:

```
1 docker service create --name accounts-db --network my-network -e
    MYSQL_ROOT_PASSWORD=bar -e MYSQL_DATABASE=models -e MYSQL_USER=
    models -e MYSQL_PASSWORD=foo mysql:5.7
```

Die Option `--name` benennt den Service, `--network` teilt dem Service das zuvor erstellte Netzwerk zu und die `-e` Optionen legen Umgebungsvariablen (Environment) für den MySQL Server fest. Zuletzt wird angegeben, welches Image der Service verwenden soll.

Dies kann analog für den Accounts Service durchgeführt werden:

```
1 docker service create --name accounts --network my-network --publish
    80:8080 accounts
```

[6] https://docs.docker.com/engine/reference/commandline/build/

Hier werden keine Umgebungsvariablen benötigt, jedoch kommt die Option `--publish` hinzu, welche einen Service Port *8080* auf einen externen Zielport *80* mappt.

Eine Übersicht der Services kann über `docker service ls` abgerufen werden, genauere Informationen über einzelne Service sind verfügbar via:

```
1 docker service inspect --pretty [SERVICE]
```

Diese Informationen enthalten – beispielsweise beim Accounts Service – die Information, dass der externe Port 80 auf den Service Port 8080 gemappt ist. Änderungen an der Service Konfiguration können mit dem Befehl `docker service update [OPTIONS] [SERVICE]` durchgeführt werden. So können Port Mappings und andere Optionen auch im Nachhinein hinzugefügt oder gelöscht werden.

Unter anderem enthalten Service Informationen auch die Anzahl der Replika. Diese beschreibt die Skalierung eines Services. Ein Service kann mit dem Befehl

```
1 docker service scale [SERVICE]=[AMOUNT]
```

skaliert werden. Wird der Wert von Accounts auf 2 gesetzt, so kann beobachtet werden, dass unter `docker service ls` sich der Wert für Replika verändert und ein weiterer Container für den Service ausgeführt wird. Hierbei sollte jedoch beachtet werden, dass beispielsweise ein MySQL Server nicht ohne Cluster Konfiguration mit mehr als einer Instanz ausgeführt werden darf. Diese würden sonst unabhängig voneinader arbeiten und unterschiedliche Datenbestände führen. Durch den internen Load Balancer könnten Anfragen an die Datenbank somit zu unterschiedlichen Ergebnissen und somit auch zu Fehlern führen.

Nachdem alle Dienste gestartet und erreichbar sind, wird die Durchführung von rollenden Updates betrachtet. Das Verhalten der Updates kann pro Service während der Erstellung oder auch im Nachhinein mit `docker service update` konfiguriert werden. Docker hat als Standard folgende Werte gesetzt:

`--update-delay`	0	Zeit zwischen der Aktualisierung einzelner Replikas
`--update-failure-action`	pause	Aktion falls Update fehlschlägt
`--update-max-failure-ratio`	0	Tolerierte Fehlerrate während eines Updates
`--update-monitor`	0	Zeit der Fehlerüberwachung nach jeder Aktualisierung eines Replikas
`--update-parallelism`	1	Wie viele Replikas gleichzeitig aktualisiert werden

Mit dem Befehl `docker service update --image accounts:1.1 accounts` wird Swam mitgeteilt, dass alle Replika von `accounts` als Image die Version 1.1 vom accounts Image nutzen sollen. Ist dies nicht der Fall, so wird ein Update ausgeführt. Jedes Replika wird nacheinander beendet, mit dem neuen Image gestartet und im Fehlerfall auf das zuvor konfigurierte Image zurückgesetzt. Dies ermöglicht, soweit mindestens zwei Replika existieren, einen ausfallfreien Übergang zwischen verschiedenen Versionen und sichert im Fehlerfall den weiteren Betrieb ab.

Sollen Services gelöscht werden, so kann dies mit `docker service rm [SERVICE]` durchgeführt werden. Ein Netzwerk kann analog hierzu mit `docker network rm [NETWORK]` gelöscht werden.

Diese Grundlagen sollten eine Basis für den Betrieb im Test- oder Produktivsystem bieten, jedoch sollten zusätzliche Sicherheitsvorkehrungen – wie Firewalls – in Betracht gezogen werden.

6. Evaluierung

Nachdem eine grundlegende Architektur geschaffen wurde, die ersten Teile des SaaS-Systems implementiert worden sind und der Betrieb mit Docker erläutert wurde, kann ein Fazit über die Entwicklung von SaaS mit Spring und Docker gezogen werden.

6.1. Fazit

Die Vor- und Nachteile von SaaS-Systemen und Microservices wurden bereits in den Grundlagen (Kapitel 2 (S. 3)) erläutert, jedoch sind weitere Punkte während der Entwicklung aufgefallen. Dies bezieht sich auch auf die zuvor getroffenen Entscheidungen, wie die Nutzung von Spring oder auch Docker.

Die Entwicklung mit Microservices kann schnell komplex werden und die Gefahr zirkulare Abhängigkeiten einzuführen ist groß. Dies ist in diesem Fall unter anderem HATEOAS und dem HAL-Standard zuzuschreiben. Nach dieser Spezifikation enthalten Ressourcen immer Links zu ihren Relationen, welche wiederum zu ihren Relationen verlinken. Im nächsten Schritt kann diese Eigenschaft dazu führen, dass bereits Aufrufe zurück zur ersten Ressource stattfinden. In Verbindung mit der Traverson Bibliothek führte dieses Verhalten schnell zu zirkularen Abhängigkeiten. Zwar konnten mit Traverson nach dem HATEOAS Prinzip Links "generiert" bzw. ermittelt werden, jedoch muss für die Relationen einer Ressource der gleiche Schritt mit Traverson durchgeführt werden. Ein simples Beispiel kann schon eine 1:n Beziehung darstellen. Ein Benutzer X gehört zu Agentur Y. Die Ressource von Y enthält den Link zur Ressource von Y. Y wiederum den Link zur Liste aller Benutzer, die zu Y gehören. Da Traverson diesen Link einmal aufruft, wird für alle Benutzer die Relation zu Y aufgelöst. Schon befindet sich das System in einem zirkularen Verhältnis. Als vorübergehende Lösung kann hier beispielsweise die manuelle Zusammenstellung des Links ohne Aufruf des anderen Dienstes genutzt werden.

Abgesehen von diesem Problem können HATEOAS und HAL als eine saubere und gute Lösung für Schnittstellen angesehen werden, da diese sowohl von Mensch als auch von Maschine einfach zu verarbeiten sind.

Auch die Spring Projekte brachten viele Vorteile in den verschiedensten Bereichen mit sich und führten schnell zu Ergebnissen. Mit den Repositories kann innerhalb kürzester Zeit ein Microservice für eine Ressource gebaut werden. Dieser ist sofort dem HAL-Standard entsprechend und kann mit nur einer Annotation via OAuth geschützt werden. Sobald jedoch von der Standard Konfiguration abgewichen wird und es für diese keine Funktion innerhalb oder zwischen den Spring kompatiblen Projekten gibt, kann es schnell umständlich werden. Viele Klassen (auch bereits vorhandene) müssen in diesem Moment selbst neu implementiert werden, obwohl beispielsweise für die Multi-Tenancy-Architektur nur ein allgemeingültiger Filter für das Repository nötig wäre. Hierbei handelt es sich jedoch um Meckern auf hohem Niveau. Dank des Open Source Ansatzes, kann solch eine Funktionalität jederzeit vom Entwickler selbst ergänzt und somit in das Spring Projekt übernommen werden.

Zu guter Letzt soll Docker nicht unerwähnt bleiben, welches sowohl die lokale Entwicklung, als auch den späteren Einsatz im Live System vereinfacht. Sowohl die Performance, als auch die einfache Orchestration und Skalierung durch den Swarm Mode, machen es zu einer guten Alternative zu klassischen Root Servern und Virtuellen Maschinen. Jedoch hat auch dieses Prinzip seine Schwächen.

In Bezug auf Spring bzw. Java, zeigen sich diese in der Java Virtual Machine (JVM). Normalerweise nutzen Java Programme eine gemeinsame JVM, wodurch eine effiziente Speicherverwaltung gewährleistet ist. In einer Docker Umgebung hingegen, sind die Dienste unabhängig voneinander und besitzen jeweils ihre eigene JVM. Dies erzeugt einen erhöhten Speicher Verbrauch und bedeutet nach einer gewissen Laufzeit pro Dienst, welcher in einer JVM läuft, einen Verbrauch von etwa 1GB an Arbeitsspeicher.

Ein weiterer Kritikpunkt stellt hierbei die Persistenz von Daten dar. Docker kennt grundlegend keine Persistenz und die Daten eines Containers existieren nur solange der Container existiert. Eine Lösung stellen die Volumes dar. Diese funktionieren zwar einwandfrei auf einem einzelnen Docker Host, jedoch noch nicht Node übergreifend im Swarm Mode. Hierfür wird zusätzliche Software benötigt, sodass beispielsweise Datenbanken über ein verteiltes Volume, unabhängig von der Node auf dem eine Instanz ausgeführt wird, ihren Zustand stets beibehalten. Eine vorübergehende Lösung stellen Constraints, welche die Services an bestimmte Nodes binden können, oder dedizierte Datenbank Server dar.

Zusammengefasst stellen diese Komponenten aber eine solide Basis für ein SaaS-System dar. Als Sprache für die Entwicklung von den Diensten sollte eher von Java abgesehen werden, da viele Vorteile der JVM verloren gehen. Hier erweisen sich Sprachen wie Node.js (Javascript) oder Go voraussichtlich performanter und speichereffizienter. Auch die Prinzipien SaaS und Microservices haben sich während der Entwicklung als sinnvoll erwiesen. Sie erzeugten einen viel flexibleren Eindruck, als es ein Monolith bieten könnte, und erleichtern die Kontrolle und den Support gegenüber dem Kunden.

6.2. Ausblick

Abschließend folgt noch ein kurzer Ausblick auf weitere Möglichkeiten, die das System bietet. Dank der einheitlichen API und der OAuth2 Authentifizierung ist die Anbindung an eine App ohne serverseitige Anpassungen möglich. Schon während der Entwicklung konnte parallel eine prototypische App entwickelt werden. Neben weiteren Diensten können auch externe Systeme, wie Online Banking (gesamte Buchhaltung innerhalb der Anwendung) oder auch PayPal (automatisierte Abrechnung der Abonnements) entsprechend eingebunden und durch eigene Dienste flexibel entwickelt werden.

A. Anhang

A.1. Multi Tenancy: AgencyFilteredRepositoryImpl.java

```
1  package utilities.repositories;
2
3  import org.springframework.data.domain.Pageable;
4  import org.springframework.data.domain.Sort;
5  import org.springframework.data.jpa.domain.Specification;
6  import org.springframework.data.jpa.domain.Specifications;
7  import org.springframework.data.jpa.repository.support.
       JpaEntityInformation;
8  import org.springframework.data.jpa.repository.support.
       SimpleJpaRepository;
9  import utilities.security.AuthedUser;
10
11 import javax.persistence.EntityManager;
12 import javax.persistence.TypedQuery;
13 import javax.persistence.criteria.CriteriaBuilder;
14 import javax.persistence.criteria.CriteriaQuery;
15 import javax.persistence.criteria.Predicate;
16 import javax.persistence.criteria.Root;
17 import java.io.Serializable;
18 import java.util.UUID;
19
20 /**
21   * Die eigene Implementation erweitert das standardmäßige
22   * "SimpleJpaRepository" und implementiert entsprechend das zuvor
         erstellte Interface.
23   */
24 public class AgencyFilteredRepositoryImpl<T, ID extends Serializable
       > extends SimpleJpaRepository<T, ID> implements
       AgencyFilteredRepository<T, ID> {
25     // Konstruktor, welcher die Parameter nur an die Elternklasse "
           SimpleJpaRepository" weitergibt
26     public AgencyFilteredRepositoryImpl(JpaEntityInformation<T, ?>
           entityInformation,
27                                         EntityManager entityManager)
                                               {
28         super(entityInformation, entityManager);
29     }
30
```

```
31      /**
32       * Creates a new {@link TypedQuery} from the given {@link
             Specification} and adds a filter by the User's agency.
33       *
34       * @param spec can be {@literal null}.
35       * @param pageable can be {@literal null}.
36       * @return
37       */
38      @Override
39      protected TypedQuery<T> getQuery(Specification<T> spec, Pageable
             pageable) {
40          Specifications<T> newSpec = null;
41
42          // Keine Spezifikation gegeben
43          if(spec == null) {
44              // erstelle Spezifikation, welche ein Where auf die
                     Agentur enthält
45              newSpec = Specifications.where(agencyFilter());
46          } else {
47              // Die vorhandene Spezifikation wird mit einer Where
                     Abfrage auf die Agentur erweitert
48              newSpec = Specifications.where(agencyFilter()).and(spec)
                     ;
49          }
50
51          /**
52            * Die angepasste Spezifikation wird mit dem weiteren
53            * Parameter "pageable" an die "getQuery" Methode der
54            * Elternklasse übergeben. Sollte sich die Funktionsweise
55            * von "getQuery" in der Elternklasse ändern, so ist diese
56            * Implementation weiterhin einsatzfähig.
57            */
58          return super.getQuery(newSpec, pageable);
59      }
60
61      /**
62       * Creates a {@link TypedQuery} for the given {@link
             Specification} and {@link Sort} and adds a filter by the
             User's agency.
63       *
```

```
64      * @param spec can be {@literal null}.
65      * @param sort can be {@literal null}.
66      * @return
67      */
68     @Override
69     protected TypedQuery<T> getQuery(Specification<T> spec, Sort
           sort) {
70         // identisch zur vorherigen Methode, nur "sort" Parameter
               statt "pageable"
71         Specifications<T> newSpec = null;
72
73         if(spec == null) {
74             newSpec = Specifications.where(agencyFilter());
75         } else {
76             newSpec = Specifications.where(agencyFilter()).and(spec)
                   ;
77         }
78
79         return super.getQuery(newSpec, sort);
80     }
81
82     /**
83      * Creates a new count query for the given {@link Specification}
             and adds a filter by the User's agency.
84      *
85      * @param spec can be {@literal null}.
86      * @return
87      */
88     @Override
89     protected TypedQuery<Long> getCountQuery(Specification<T> spec)
           {
90         // identisch zur vorherigen Methode, jedoch keine
               Paginierung oder Sortierung, stattdessen für Abfragen,
               welche eine Zählung ausführen
91         Specifications<T> newSpec = null;
92
93         if(spec == null) {
94             newSpec = Specifications.where(agencyFilter());
95         } else {
```

```
96              newSpec = Specifications.where(agencyFilter()).and(spec)
                    ;
97          }
98
99          return super.getCountQuery(newSpec);
100     }
101
102     /**
103      * Creates a new {@link Specification} which filters by the
             agency field.
104      *
105      * @return
106      */
107     protected Specification<T> agencyFilter() {
108         // Es wird eine Spezifikations Instanz erstellt
109         return new Specification<T>() {
110             /**
111               * Diese enthält eine "toPredicate" Methode, welche die
112               * Spezifikation in ein Predikat zur Generierung einer
113               * Datenbankabfrage umwandelt.
114               */
115             @Override
116             public Predicate toPredicate(Root<T> root, CriteriaQuery
                    <?> query, CriteriaBuilder cb) {
117                 // UUID der Agentur, falls es sich um einen Nutzer
                        handelt. Sonst "null"
118                 UUID agency = agency();
119
120                 // Nur wenn Agentur gesetzt ist, dann muss auch
                        gefiltert werden.
121                 if (agency != null) {
122                     /**
123                       * Mit dem CriteriaBuilder wird eine
                            Gleichheitsprüfung
124                       * zwischen dem Wert des Feldes "agency" und
                            dem Wert der
125                       * Variable "agency" definiert.
126                       */
127                     return cb.equal(root.get("agency"), agency);
128                 }
```

```
129
130                 // Führe keine Filterung durch
131                 return null;
132             }
133         };
134     }
135
136     protected UUID agency() {
137         // Statische Methode einer Hilfsklasse, welche die Agentur
                aus den OAuth Daten ausliest
138         return AuthedUser.getAgency();
139     }
140 }
```

A.2. Multi Tenancy: AgencyFilteredRepositoryFactoryBean.java

```
1  package utilities.repositories;
2
3  import org.springframework.data.jpa.provider.PersistenceProvider;
4  import org.springframework.data.jpa.provider.QueryExtractor;
5  import org.springframework.data.jpa.repository.support.
       JpaRepositoryFactory;
6  import org.springframework.data.jpa.repository.support.
       JpaRepositoryFactoryBean;
7  import org.springframework.data.repository.core.RepositoryMetadata;
8  import org.springframework.data.repository.core.support.
       RepositoryFactorySupport;
9
10 import javax.persistence.EntityManager;
11
12 /**
13  * Injects a custom repository factory, which returns the {@link
        AgencyFilteredRepositoryImpl} class, if the repository
14  * interface is based on {@link AgencyFilteredRepository}. Else it
        will use the result of {@link JpaRepositoryFactory#
        getRepositoryBaseClass(RepositoryMetadata)}.
15  *
```

```
16  * @author Daniel Gehn <daniel.gehn@haw-hamburg.de>, <dg@awesom-
        media.de>
17  */
18 // Es wird um die standardmäßig genutzte "JpaRepositoryFactoryBean"
       erweitert
19 public class AgencyFilteredRepositoryFactoryBean extends
       JpaRepositoryFactoryBean {
20
21     /**
22       * Konstruktor, welcher die Parameter nur an die Elternklasse
23       * übergibt
24       */
25     @Override
26     protected RepositoryFactorySupport createRepositoryFactory(
           EntityManager entityManager) {
27         return new AgencyFilteredRepositoryFactory(entityManager);
28     }
29
30     // Die eigentliche Factory Klasse. Diese erweitert auch die
           standard JPA Factory
31     private class AgencyFilteredRepositoryFactory extends
           JpaRepositoryFactory {
32
33         // Konstruktor, welcher erneut nur Parameter an die
               Elternklasse übergibt
34         public AgencyFilteredRepositoryFactory(EntityManager
               entityManager) {
35             super(entityManager);
36         }
37
38         /**
39           * Methode entscheidet, welche Klasse die Basis bzw. die
40           * Implementation für das angegebene Repository (ein
                Interface) bilden soll.
41           */
42
43         @Override
44         protected Class<?> getRepositoryBaseClass(RepositoryMetadata
               metadata) {
```

```
45              // Überprüfung, ob das Repository das angepasste
                    Repository Interface erweitert
46              if(AgencyFilteredRepository.class.isAssignableFrom(
                    metadata.getRepositoryInterface())) {
47                  // Ist dies der Fall, so soll die eigene
                        Implementation genutzt werden
48                  return AgencyFilteredRepositoryImpl.class;
49              }
50
51              // Ansonsten wird wieder der standard JPA Factory ü
                    berlassen, welche Implementation genutzt werden soll.
52              return super.getRepositoryBaseClass(metadata);
53          }
54      }
55  }
```

A.3. Auth Service: CustomAuthenticationProvider.java

```
1  package service.auth;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.cloud.client.ServiceInstance;
5  import org.springframework.cloud.client.loadbalancer.
       LoadBalancerClient;
6  import org.springframework.security.authentication.
       AuthenticationProvider;
7  import org.springframework.security.authentication.DisabledException
       ;
8  import org.springframework.security.authentication.LockedException;
9  import org.springframework.security.authentication.
       UsernamePasswordAuthenticationToken;
10 import org.springframework.security.core.Authentication;
11 import org.springframework.security.core.AuthenticationException;
12 import org.springframework.security.crypto.password.PasswordEncoder;
13 import org.springframework.stereotype.Service;
14 import org.springframework.web.client.HttpClientErrorException;
15 import org.springframework.web.client.HttpServerErrorException;
16 import org.springframework.web.client.RestTemplate;
17 import service.auth.models.Account;
18
```

```
19 import java.net.URI;
20
21 /**
22  * Restful implementation of the Spring Security
        AuthenticationProvider.
23  *
24  * @author Daniel Gehn <daniel.gehn@haw-hamburg.de>, <dg@awesom-
        media.de>
25  */
26 @Service
27 public class CustomAuthenticationProvider implements
        AuthenticationProvider {
28
29     // Es wird ein Passwort Encoder benötigt, um mit dem
           gespeicherten Passwort vergleichen zu können.
30     @Autowired
31     private PasswordEncoder passwordEncoder;
32
33     // Load Balancer Client, um Instanz zum Service zu erhalten
34     @Autowired
35     private LoadBalancerClient loadBalancer;
36
37     @Override
38     public Authentication authenticate(Authentication authentication
           ) throws AuthenticationException {
39         // Benutzername und Passwort aus den Authentifizierungsdaten
40         String username = authentication.getName();
41         String password = authentication.getCredentials().toString()
               ;
42
43         URI accountsUri = null;
44
45         // Versuch eine Instanz vom accounts Dienst zu wählen
46         try {
47             ServiceInstance instance = loadBalancer.choose("accounts
                   ");
48             if (instance != null) {
49                 accountsUri = instance.getUri();
50             }
51         } catch (RuntimeException e) {
```

```
52              // Eureka not available
53          }
54
55          // Sollte keine Instanz verfügbar sein, kein Login
56          if(accountsUri == null) {
57              return null;
58          }
59
60          // Rest Template für REST Operationen
61          RestTemplate restTemplate = new RestTemplate();
62
63          try {
64              /**
65                * GET Anfrage an den accounts Dienst, mit Suche
                      anhand vom Benutzernamen (username).
66                * Wandelt die Antwort in ein Account Objekt um,
                      welches nur die für
67                * den Auth Dienst relevanten Informationen enthält,
                      unabhängig davon,
68                * ob es weitere Informationen gibt.
69                */
70              Account account = restTemplate.getForObject(accountsUri
                    + "/accounts/search/username?username=" + username,
                    Account.class);
71
72              // Wenn der Accounts Dienst mit 404 (Not Found)
                    antwortet, so besitzt account den Wert null
73              if (account == null) {
74                  return null;
75              }
76          } catch (HttpServerErrorException | HttpClientErrorException
                    ex) {
77              return null;
78          }
79
80          // Solange ein Konto nicht bestätigt ist, darf ein Login
                nicht stattfinden.
81          if(!account.getConfirmed()) {
82              throw new DisabledException("Account_is_not_confirmed.")
                    ;
```

```
83          }
84
85          // Solange ein Konto gesperrt ist, darf ein Login nicht
                stattfinden.
86          if(account.getLocked()) {
87              throw new LockedException("Account is locked.");
88          }
89
90          // Überprüfung des Passworts auf Korrektheit mit dem
                Passwort Encoder
91          if (passwordEncoder.matches(password, account.getPassword())
                ) {
92              // Erstellung eines Tokens, welches die User
                    Authentifizierung enthält
93              return new UsernamePasswordAuthenticationToken(account.
                    getUsername(), account.getPassword(), account.
                    getAuthorities());
94          }
95
96          // Passwörter stimmen nicht überein, Login fehlgeschlagen.
97          return null;
98      }
99
100     // Dieser Authentication Provider unterstützt Authentifzierung f
            ür das UsernamePasswordAuthenticationToken Verfahren
101     @Override
102     public boolean supports(Class<?> authentication) {
103         return authentication.equals(
                UsernamePasswordAuthenticationToken.class);
104     }
105
106 }
```

A.4. Auth Service: CustomUserDetailsService.java

```
1 package service.auth;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.cloud.client.ServiceInstance;
```

```
5  import org.springframework.cloud.client.loadbalancer.
       LoadBalancerClient;
6  import org.springframework.security.core.userdetails.User;
7  import org.springframework.security.core.userdetails.UserDetails;
8  import org.springframework.security.core.userdetails.
       UserDetailsService;
9  import org.springframework.security.core.userdetails.
       UsernameNotFoundException;
10 import org.springframework.stereotype.Service;
11 import org.springframework.web.client.HttpClientErrorException;
12 import org.springframework.web.client.HttpServerErrorException;
13 import org.springframework.web.client.RestTemplate;
14 import service.auth.models.Account;
15
16 import java.net.URI;
17
18 /**
19  * Restful implementation of the Spring Security UserDetailsService.
20  *
21  * @author Daniel Gehn <daniel.gehn@haw-hamburg.de>, <dg@awesom-
        media.de>
22  */
23 @Service
24 public class CustomUserDetailsService implements UserDetailsService
       {
25
26     // Load Balancer Client, um Instanz zum Service zu erhalten
27     @Autowired
28     private LoadBalancerClient loadBalancer;
29
30     @Override
31     public UserDetails loadUserByUsername(String username) throws
           UsernameNotFoundException {
32         URI accountsUri = null;
33
34         // Versuch eine Instanz vom accounts Dienst zu wählen
35         try {
36             ServiceInstance instance = loadBalancer.choose("accounts
                   ");
37             if(instance != null) {
```

```
38                  accountsUri = instance.getUri();
39              }
40          } catch (RuntimeException e) {
41              // Eureka not available
42          }
43
44          // Sollte keine Instanz verfügbar sein, so wird eine
                entsprechende Exception geworfen.
45          if(accountsUri == null) {
46              throw new UsernameNotFoundException("Account service is
                    currently unavailable.");
47          }
48
49          // Rest Template für REST Operationen
50          RestTemplate restTemplate = new RestTemplate();
51
52          try {
53              /**
54                * GET Anfrage an den accounts Dienst, mit Suche
                      anhand vom Benutzernamen (username).
55                * Wandelt die Antwort in ein Account Objekt um,
                      welches nur die für
56                * den Auth Dienst relevanten Informationen enthält,
                      unabhängig davon,
57                * ob es weitere Informationen gibt.
58                */
59              Account account = restTemplate.getForObject(accountsUri
                    + "/accounts/search/username?username=" + username,
                    Account.class);
60
61              // Wenn der Accounts Dienst mit 404 (Not Found)
                    antwortet, so besitzt account den Wert null
62              if (account == null) {
63                  throw new UsernameNotFoundException("Unknown user");
64              }
65
66              // Konto wurde gefunden und es wird ein Spring Security
                    User zurückgegeben mit den relevanten Informationen
                    des Benutzers
```

```
67              return new User(account.getUsername(), account.
                    getPassword(), account.getConfirmed(), true, true, !
                    account.getLocked(), account.getAuthorities());
68          } catch (HttpServerErrorException | HttpClientErrorException
                ex) {
69              throw new UsernameNotFoundException("Unknown user");
70          }
71      }
72
73  }
```

A.5. Gateway: application.yml

```
1  # Konfiguration vom Zuul Proxy
2  zuul:
3    # Wenn Eureka (s.u.) konfiguriert ist, werden standardmäßig alle
       Dienste geproxied. Es sollen jedoch alle Dienste ignoriert
       werden, damit volle Kontrolle über Routen besteht.
4    ignored-services: "*"
5    # Ob Anfragen erneut gesendet werden dürfen bei
       Verbindungsproblemen
6    retryable: true
7    # Standard ist "Cookie,Set-Cookie,Authorization", aber das Bearer
       Token (im Authorization Header) muss an die einzelnen Dienste
       weitergegeben werden
8    sensitive-headers: Cookie,Set-Cookie
9    # Deklarationen der Routen. In diesem Fall die Dienste.
10   routes:
11     # Name des Dienstes
12     accounts:
13       # Pfad des Dienstes "http://gateway/accounts/**", wobei "**"
           keine oder eine beliebige Zeichenkette darstellt.
14       path: /accounts/**
15       # Name des Dienstes bei Eureka, um eine verfügbare Instanz zu
           finden.
16       serviceId: accounts
17       # Standard ist "true", dies erzeugt jedoch sich doppelnde URIs
             "/accounts/accounts/**". Der Wert "false" sendet das
           Prefix an den Dienst, welcher dies berücksichtigen kann.
18       strip-prefix: false
```

```
    agencies:
      path: /agencies/**
      serviceId: agencies
      strip-prefix: false
# Und alle weiteren Dienste, welche nach außen verfügbar sein sollen.

# Größere Dateiuploads ermöglichen, in diesem Fall darf eine Datei bis zu 100Mb und die Anfrage insgesamt 200Mb groß sein.
spring:
  http:
    multipart:
      max-file-size: 100Mb
      max-request-size: 200Mb

# Sicherstellen, dass das Timeout vom Proxy später einsetzt, falls ein Dienst länger zum Antworten braucht.
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000

# Standard Konfiguration von OAuth2 und Eureka, wie auch in normalen Diensten.
security:
  oauth2:
    resource:
      userInfoUri: http://auth:9999/uaa/user

eureka:
  client:
    service-url:
      defaultZone: http://eureka:8761/eureka/
```

A.6. SSOClient: application.yml

```
# Konfiguration vom Zuul Proxy
zuul:
```

```
  # Allgemeines Prefix für die Routen (ergibt dann Pfade wie "/api/
    accounts/**")
  prefix: /api

# Während ein Resource Server sich nur Informationen besorgt, muss
   ein SSO Client auch Token abfragen und den Benutzer zum Login
   weiterleiten
security:
  oauth2:
    client:
      accessTokenUri: http://auth:9999/uaa/oauth/token
      userAuthorizationUri: http://auth.appname.dev/uaa/oauth/
        authorize
      clientId: acme
      clientSecret: acmesecret
```

A.7. ObjectStorage.java

```
package service.attachments;

import io.minio.MinioClient;
import io.minio.errors.*;
import io.minio.policy.PolicyType;
import org.xmlpull.v1.XmlPullParserException;

import java.io.File;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

public class ObjectStorage {

    // URL zum S3 Server
    private String server;

    // Access Key für den S3 Server
    private String accessKey;

    // Secret Key für den S3 Server
    private String secretKey;
```

```
23
24      // Instanz vom MinioClient
25      private MinioClient client;
26
27      public MinioTools(String server, String accessKey, String
            secretKey) throws InvalidPortException,
            InvalidEndpointException {
28          this.server = server;
29          this.accessKey = accessKey;
30          this.secretKey = secretKey;
31
32          this.client = new MinioClient(server, accessKey, secretKey);
33      }
34
35      /**
36       *
37       * @param file The file to upload
38       * @param prefix A prefix to apply individual policies
39       * @param bucket The bucket where the file will reside
40       * @param deleteFileAfterUpload Delete the uploaded file after
             it's uploaded
41       * @param policy The policy to apply to the prefix
42       * @return The object name
43       * @throws IOException
44       * @throws InvalidKeyException
45       * @throws NoSuchAlgorithmException
46       * @throws InsufficientDataException
47       * @throws InternalException
48       * @throws NoResponseException
49       * @throws InvalidBucketNameException
50       * @throws XmlPullParserException
51       * @throws ErrorResponseException
52       * @throws InvalidArgumentException
53       * @throws InvalidObjectPrefixException
54       */
55      public String upload(File file, String prefix, String bucket,
            Boolean deleteFileAfterUpload, PolicyType policy) throws
            IOException, InvalidKeyException, NoSuchAlgorithmException,
            InsufficientDataException, InternalException,
            NoResponseException, InvalidBucketNameException,
```

```
        XmlPullParserException, ErrorResponseException,
        InvalidArgumentException, InvalidObjectPrefixException {
56      // Erstellung des Buckets, falls dieser noch nicht existiert
57      if (!client.bucketExists(bucket)) {
58          client.makeBucket(bucket);
59      }
60
61      // Sollten eine Policy und ein Prefix gegeben sein, so soll
            diese für den Bucket und ein bestimmtes Prefix gesetzt
            werden
62      if(policy != null && prefix != null) {
63          client.setBucketPolicy(bucket, prefix, policy);
64      }
65
66      String objectName = file.getName();
67
68      /**
69       *  Wenn Prefix vorhanden, dann wird dieses vor den
             Dateinamen gesetzt.
70       *  Dies kann zum Erzeugen einer virtuellen Hierarchie
             genutzt werden.
71       *
72       *  Beispiel mit prefix = "attachments/":
73       *  Es erhalten alle Dateien dieses Prefix, somit können
             Bucket Policies für einzelne Prefixe gesetzt werden.
74       */
75      if(prefix != null) {
76          objectName = prefix + objectName;
77      }
78
79      // Datei hochladen (Bucket, Name im Object Storage, lokaler
            Pfad)
80      client.putObject(bucket, objectName, file.getCanonicalPath()
            );
81
82      // Wenn original Datei nach Upload gelöscht werden soll
83      if(deleteFileAfterUpload) {
84          file.delete();
85      }
86
```

```
87          return objectName;
88      }
89
90      /**
91       * Returns a url to the given object in the given bucket or null
             if it's not valid.
92       * The Link expires after the given expiration time (in seconds)
              .
93       *
94       * @param bucket
95       * @param objectName
96       * @param expiration
97       * @return
98       */
99      public String downloadLink(String bucket, String objectName,
             Integer expiration) {
100          /**
101           *  Versuch eine vorsignierte Url zum Objekt zu erhalten
                 unter Angabe des Buckets und des Objekt Namens.
102           *  expiration gibt an wie lange der Link gültig ist (in
                 Sekunden)
103           */
104          try {
105              return minioTools.getClient().presignedGetObject(bucket,
                     objectName, expiration);
106          } catch (InvalidBucketNameException
107                  | NoSuchAlgorithmException
108                  | IOException
109                  | InsufficientDataException
110                  | XmlPullParserException
111                  | ErrorResponseException
112                  | InternalException
113                  | InvalidExpiresRangeException
114                  | InvalidKeyException
115                  | NoResponseException e) {
116              e.printStackTrace();
117          }
118
119          return null;
120      }
```

```
}
```

A.8. Microservice: Dockerfile

```
# Image auf dem das zu bauende basieren soll
# In diesem Fall ein Linux Image, welches Java in Version 8 vorinstalliert hat.
FROM java:8

# Kopieren der gebauten Jar Datei und der Libs
# Ist der Pfad ein Ordner, so kopiert COPY alle Inhalte des Ordners rekursiv.
COPY ./build/install/accounts /usr/local/bin/accounts

# Arbeitsverzeichnis wechseln in unser neu erstelltes Verzeichnis
WORKDIR /usr/local/bin/accounts

# CMD beschreibt, welcher Befehl beim Start des Containers
# ausgeführt werden soll.
# In diesem Fall wird die JAR des Dienstes ausgeführt.
# Zusätzlich wird "/dev/./urandom" als Quelle für Zufallszahlen
# gesetzt, da dies die Ausführung von Java beschleunigt, wenn
# Zufallszahlen genutzt werden.
# Spring nutzt diese an mehreren Stellen und würde ohne
# "urandom" nach kurzer Zeit bei der parallelen Ausführung von
# mehreren Diensten blockieren.
CMD ["/usr/bin/java", "-jar", "-Djava.security.egd=file:/dev/./urandom", "accounts-1.0.0-SNAPSHOT.jar"]
```

A.9. docker-compose.yml

```
# Version des Docker Compose Konfigurationsformats
version: '2'
services:
  # Wir definieren einen Service mit dem Namen "accounts"
  accounts:
    # Dieser soll ein Build im Verzeichnis "accounts" ausführen (das Dockerfile)
    build: "accounts"
    # Mapping von Ports des Containers auf Ports des Host Systems
```

```
    # Ein Aufruf von http://host:80 würde die Anfrage auf den
    # internen Hostnamen http://accounts:8080 leiten, welcher
    # über den internen Load Balancer aufgelöst wird.
    ports:
      - "80:8080"
    # Benötigt die Account Datenbank, diese muss somit zuvor
       gestartet werden
    depends_on:
      - accounts-db
    # Netzwerk Verbindung zur Account Datenbank muss gegeben sein
    links:
      - accounts-db

  # Definierung der Account Datenbank
  accounts-db:
    # Basiert auf dem offiziellen MySQL Image mit MySQL Version 5.7
    image: mysql:5.7
    # Gewünscht ist ein Volume, sodass kein Datenverlust entsteht.
    # Dieses wird von Docker frei bestimmt und mountet auf den
    # angegebenen Pfad.
    volumes:
      - /var/lib/mysql
    # Environment Variablen, um MySQL zu konfigurieren
    environment:
      MYSQL_ROOT_PASSWORD: bar
      MYSQL_DATABASE: accounts
      MYSQL_USER: accounts
      MYSQL_PASSWORD: foo
```

A.10. Ember.js: serializers/application.js

```
/**
 * Custom Serializer to support our HAL based API
 *
 * Based on https://github.com/201-created/ember-data-hal-9000
 */

import DS from "ember-data";
import Ember from 'ember';
import { assert, deprecate, warn } from 'ember-data/-private/debug';
```

```
10 import isEnabled from 'ember-data/-private/features';
11
12 let {JSONAPISerializer} = DS;
13
14 // Reserved keys, per the HAL spec
15 let halReservedKeys = ['_embedded', '_links'],
16   reservedKeys = halReservedKeys.concat(['meta']),
17   keys = Object.keys;
18
19 const COLLECTION_PAYLOAD_REQUEST_TYPES = [
20   'findHasMany',
21   'findMany',
22   'query',
23   'findAll'
24 ];
25
26 /**
27  * Converts a HAL Link (an item out of `_links`) to a JSON-API
      compatible link.
28  * If the link contains more keys than `href`, those are saved as
      meta.
29  *
30  * @param link
31  * @returns {*}
32  */
33 function halToJSONAPILink(link) {
34   let converted,
35     linkKeys = keys(link);
36
37   if (linkKeys.length === 1) {
38     converted = link.href;
39   } else {
40     converted = {href: link.href, meta: {}};
41     linkKeys.forEach(key => {
42       if (key !== 'href') {
43         converted.meta[key] = link[key];
44       }
45     });
46   }
47
```

```
48    return converted;
49  }
50
51  /**
52   * Flattens an array
53   * @param array
54   * @returns {*}
55   */
56  function arrayFlatten(array) {
57    let flattened = [];
58    return flattened.concat.apply(flattened, array);
59  }
60
61  /**
62   * Custom HAL compatible serializer based on the JSON API serializer
         .
63   * @type {*}
64   */
65  const HALAPISerializer = JSONAPISerializer.extend({
66    keyForRelationship(relationshipKey/*, relationshipMeta */) {
67      return relationshipKey;
68    },
69    keyForAttribute(attributeName/*, attributeMeta */) {
70      return attributeName;
71    },
72    keyForLink(relationshipKey/*, relationshipMeta */) {
73      return relationshipKey;
74    },
75
76    /**
77     * Checks based on the requestTyp, if the payload contains a
           single object or multiple objects.
78     *
79     * @param payload
80     * @param requestType
81     * @returns {boolean}
82     */
83    isSinglePayload(payload, requestType) {
84      return COLLECTION_PAYLOAD_REQUEST_TYPES.indexOf(requestType) ===
            -1;
```

```
85  },
86
87  extractLink(link) {
88    return link.href;
89  },
90
91  /**
92   * Extracts the entities id out of the resource link.
93   * Ember Data uses ids, while HAL doesn't have public ids.
94   *
95   * @param modelClass
96   * @param resourceHash
97   * @returns {string}
98   */
99  extractId (modelClass, resourceHash) {
100   let href = resourceHash._links.self.href;
101   return href.substring(href.lastIndexOf("/") + 1);
102 },
103
104 /**
105  * Extracts additional metadata from the payload, like pagination
        information.
106  *
107  * @param store
108  * @param requestType
109  * @param payload
110  * @param primaryModelClass
111  * @returns {meta|{}|*}
112  */
113 extractMeta (store, requestType, payload, primaryModelClass) {
114   const meta = payload.meta || {},
115     isSingle = this.isSinglePayload(payload, requestType);
116
117   if (!isSingle) {
118     keys(payload).forEach(key => {
119       if (reservedKeys.indexOf(key) > -1) {
120         return;
121       }
122
123       meta[key] = payload[key];
```

```
124         delete payload[key];
125       });
126
127       if (payload._links) {
128         meta.links = this.extractLinks(primaryModelClass, payload);
129       }
130     }
131
132     return meta;
133   },
134
135   /**
136    * Normalizes the response building a document hash, so Ember Data
           can work with it.
137    *
138    * @param store
139    * @param primaryModelClass
140    * @param payload
141    * @param id
142    * @param requestType
143    * @returns {{}}
144    */
145   normalizeResponse (store, primaryModelClass, payload, id,
         requestType) {
146     const isSingle = this.isSinglePayload(payload, requestType),
147       documentHash = {},
148       meta = this.extractMeta(store, requestType, payload,
             primaryModelClass),
149       included = [];
150
151     if (meta) {
152       documentHash.meta = meta;
153     }
154
155     if (isSingle) {
156       documentHash.data = this.normalize(primaryModelClass, payload,
             included);
157     } else {
158       documentHash.data = [];
159       payload._embedded = payload._embedded || {};
```

```
160
161       const normalizedEmbedded = Object.keys(payload._embedded).map(
              embeddedKey =>
162         payload._embedded[embeddedKey].map(embeddedPayload =>
163           this.normalize(primaryModelClass, embeddedPayload,
                  included)
164         )
165       );
166
167       documentHash.data = arrayFlatten(normalizedEmbedded);
168     }
169
170     documentHash.included = included;
171     return documentHash;
172   },
173
174   /**
175    * Normalizes a single entity from the payload.
176    *
177    * @param primaryModelClass
178    * @param payload
179    * @param included
180    * @returns {*}
181    */
182   normalize(primaryModelClass, payload, included) {
183     let data;
184
185     if (payload) {
186       const attributes = this.extractAttributes(primaryModelClass,
              payload),
187         relationships = this.extractRelationships(primaryModelClass,
                payload, included);
188
189       data = {
190         id: this.extractId(primaryModelClass, payload),
191         type: primaryModelClass.modelName
192       };
193
194       if (Object.keys(attributes).length > 0) {
195         data.attributes = attributes;
```

```
196       }
197
198       if (Object.keys(relationships).length > 0) {
199         data.relationships = relationships;
200       }
201
202       if (data.attributes) {
203         this.applyTransforms(primaryModelClass, data.attributes);
204       }
205     }
206
207     return data;
208   },
209
210   /**
211    * Extracts links from the payload and transforms them to a JSON
         API compatible format.
212    *
213    * @param primaryModelClass
214    * @param payload
215    * @returns {*}
216    */
217   extractLinks(primaryModelClass, payload) {
218     let links;
219
220     if (payload._links) {
221       links = {};
222       Object.keys(payload._links).forEach(link => {
223         links[link] = halToJSONAPILink(payload._links[link]);
224       });
225     }
226
227     return links;
228   },
229
230   /**
231    * Extracts the known model attributes and removes them from the
         payload itself.
232    * Also adds a ‘links‘ attribute (if links are available), which
         contains a normalized links object.
```

```
233     *
234     * @param primaryModelClass
235     * @param payload
236     * @returns {{}}
237     */
238    extractAttributes(primaryModelClass, payload) {
239      let payloadKey,
240        attributes = {};
241
242      primaryModelClass.eachAttribute((attributeName, attributeMeta)=>
           {
243        payloadKey = this.keyForAttribute(attributeName, attributeMeta
             );
244
245        if (!payload.hasOwnProperty(payloadKey)) {
246          return;
247        }
248
249        attributes[attributeName] = payload[payloadKey];
250        delete payload[payloadKey];
251      });
252
253      if(payload._links) {
254        attributes.links = this.extractLinks(primaryModelClass,
             payload);
255      }
256
257      return attributes;
258    },
259
260    /**
261     * Extracts an included relationship. If it's an object it's
           embedded (normalized and pushed into included).
262     *
263     * @param relationshipModelClass
264     * @param payload
265     * @param included
266     * @returns {*}
267     */
268    extractRelationship(relationshipModelClass, payload, included) {
```

```
269     if (Ember.isNone(payload)) {
270       return undefined;
271     }
272
273     let relationshipModelName = relationshipModelClass.modelName,
274       relationship;
275
276     if (Ember.typeOf(payload) === 'object') {
277       relationship = {
278         id: this.extractId({}, payload)
279       };
280
281       if (relationshipModelName) {
282         relationship.type = this.modelNameFromPayloadKey(
              relationshipModelName);
283         included.push(this.normalize(relationshipModelClass, payload
              , included));
284       }
285     } else {
286       relationship = {
287         id: payload,
288         type: relationshipModelName
289       };
290     }
291
292     return relationship;
293   },
294
295   /**
296    * Extracts all embedded and linked known relationships of a model
         .
297    *
298    * @param primaryModelClass
299    * @param payload
300    * @param included
301    * @returns {{}}
302    */
303   extractRelationships(primaryModelClass, payload, included) {
304     let relationships = {},
305       embedded = payload._embedded,
```

```
306       keyForRelationship = this.keyForRelationship,
307       keyForLink = this.keyForLink,
308       extractLink = this.extractLink,
309       links = payload._links;
310
311     if (embedded || links) {
312       primaryModelClass.eachRelationship((key, relationshipMeta) =>
              {
313         let relationship,
314           relationshipKey = keyForRelationship(key, relationshipMeta
                ),
315           linkKey = keyForLink(key, relationshipMeta);
316
317         if (embedded && embedded.hasOwnProperty(relationshipKey)) {
318           let data,
319             relationModelClass = this.store.modelFor(
                  relationshipMeta.type);
320
321           if (relationshipMeta.kind === 'belongsTo') {
322             data = this.extractRelationship(relationModelClass,
                    embedded[relationshipKey], included);
323           } else if (relationshipMeta.kind === 'hasMany') {
324             data = embedded[relationshipKey].map(item => {
325               return this.extractRelationship(relationModelClass,
                      item, included);
326             });
327           }
328
329           relationship = {data};
330         }
331
332         if (links && links.hasOwnProperty(linkKey)) {
333           relationship = relationship || {};
334
335           const link = links[linkKey],
336             useRelated = !relationship.data;
337
338           relationship.links = {
339             [useRelated ? 'related' : 'self']: extractLink(link)
340           };
```

```
341         }
342
343         if (relationship) {
344           relationships[key] = relationship;
345         }
346       }, this);
347     }
348
349     return relationships;
350   },
351
352   /**
353    * Serializes an attribute if it's considered as changed or the
         record itself is new.
354    *
355    * @param snapshot
356    * @param json
357    * @param key
358    * @param attributes
359    * @returns {*}
360    */
361   serializeAttribute(snapshot, json, key, attributes) {
362     if (snapshot.changedAttributes()[key] || snapshot.record.get('
         isNew'))  {
363       this._super(snapshot, json, key, attributes);
364     }
365   },
366
367   /**
368    * Serializes a 'belongsTo' relationship.
369    * The relationship will be embedded in the attributes as id.
370    *
371    * @param snapshot
372    * @param json
373    * @param relationship
374    * @returns {*}
375    */
376   serializeBelongsTo(snapshot, json, relationship) {
377     this._super(snapshot, json, relationship);
378
```

```
379     let key = relationship.key;
380
381     if(typeof json.relationships !== "undefined" && typeof json.
            relationships[key] !== "undefined") {
382       if(typeof json.attributes === "undefined") {
383         json.attributes = {};
384       }
385
386       if(json.relationships[key].data === null) {
387         json.attributes[key] = null;
388       } else {
389         json.attributes[key] = json.relationships[key].data.id;
390       }
391     }
392   },
393
394   /**
395    * Serializes a 'hasMany' relationship.
396    * The relationship will be embedded in the attributes as relative
           path to the related entity.
397    *
398    * @param snapshot
399    * @param json
400    * @param relationship
401    * @returns {*}
402    */
403   serializeHasMany(snapshot, json, relationship) {
404     this._super(snapshot, json, relationship);
405
406     let key = relationship.key;
407
408     if(typeof json.relationships !== "undefined" && typeof json.
            relationships[key] !== "undefined") {
409       if(typeof json.attributes === "undefined") {
410         json.attributes = {};
411       }
412
413       let camelized = Ember.String.camelize(relationship.type);
414       let pluralized = Ember.String.pluralize(camelized);
415
```

```
416       json.attributes[key] = json.relationships[key].data.map(
              function(element) {
417         return "/" + pluralized + "/" + element.id;
418       });
419     }
420   },
421
422   /**
423    * Normalizes a documentHash.
424    * Adaptations had to be made, so the JSON-API Serializer knows
          the type of the data, because HAL doesn't include it
425    * in a type attribute.
426    *
427    * @param documentHash
428    * @returns {*}
429    * @private
430    */
431   _normalizeDocumentHelper(documentHash) {
432     if (Ember.typeOf(documentHash.data) === 'object') {
433       documentHash.data = this._normalizeResourceHelper(documentHash
              );
434     } else if (Array.isArray(documentHash.data)) {
435       let ret = new Array(documentHash.data.length);
436
437       for (let i = 0; i < documentHash.data.length; i++) {
438         let data = {
439           data: documentHash.data[i],
440           type: documentHash.type
441         };
442
443         ret[i] = this._normalizeResourceHelper(data);
444       }
445
446       documentHash.data = ret;
447     }
448
449     if (Array.isArray(documentHash.included)) {
450       let ret = new Array(documentHash.included.length);
451
452       for (let i = 0; i < documentHash.included.length; i++) {
```

```
453         let included = {
454           data: documentHash.included[i],
455           type: documentHash.type
456         };
457         ret[i] = this._normalizeResourceHelper(included);
458       }
459
460       documentHash.included = ret;
461     }
462
463     return documentHash;
464   },
465
466   /**
467    * Normalizes a resourceHash.
468    * Adaptation to the original helper of the JSON-API Serializer.
469    *
470    * @param resourceHash
471    * @returns {*}
472    * @private
473    */
474   _normalizeResourceHelper(resourceHash) {
475     assert(this.warnMessageForUndefinedType(), !Ember.isNone(
          resourceHash.type), {
476       id: 'ds.serializer.type-is-undefined'
477     });
478
479     let modelName, usedLookup;
480
481     if (isEnabled("ds-payload-type-hooks")) {
482       modelName = this.modelNameFromPayloadType(resourceHash.type);
483       let deprecatedModelNameLookup = this.modelNameFromPayloadKey(
            resourceHash.type);
484
485       usedLookup = 'modelNameFromPayloadType';
486
487       if (modelName !== deprecatedModelNameLookup && this.
            _hasCustomModelNameFromPayloadKey()) {
488         deprecate("You are using modelNameFromPayloadKey to
              normalize the type for a resource. This has been
```

```
            deprecated_in_favor_of_modelNameFromPayloadType", false,
            {
489           id: 'ds.json-api-serializer.deprecated-model-name-for-
                resource',
490           until: '3.0.0'
491         });
492
493         modelName = deprecatedModelNameLookup;
494         usedLookup = 'modelNameFromPayloadKey';
495       }
496     } else {
497       modelName = this.modelNameFromPayloadKey(resourceHash.type);
498       usedLookup = 'modelNameFromPayloadKey';
499     }
500
501     if (!this.store._hasModelFor(modelName)) {
502       warn(this.warnMessageNoModelForType(modelName, resourceHash.
             type, usedLookup), false, {
503         id: 'ds.serializer.model-for-type-missing'
504       });
505       return null;
506     }
507
508     let modelClass = this.store.modelFor(modelName);
509     let serializer = this.store.serializerFor(modelName);
510
511     return serializer.normalize(modelClass, resourceHash);
512   },
513 });
514
515 export default HALAPISerializer;
```

A.11. Ember.js: adapters/application.js

```
1 import DS from "ember-data";
2 import Ember from "ember";
3 import config from "appname/config/environment";
4
5 // HAL has search endpoints by specification under the `[RESOURCE]/
    search` prefix.
```

```
6  const SEARCH_PREFIX = '/search';
7
8  /**
9   * Custom implementation of the default RESTAdapter of Ember-Data to
        support HAL.
10  */
11 export default DS.RESTAdapter.extend({
12   namespace: config.apiNamespace,
13   // requests and responses should contain HAL JSON
14   headers: {
15     'Content-Type': 'application/hal+json'
16   },
17
18   /**
19    * Generates the path to a model. Names are camelized and
          pluralized.
20    *
21    * @param modelName
22    * @returns {*}
23    */
24   pathForType: function(modelName) {
25     let camelized = Ember.String.camelize(modelName);
26
27     return Ember.String.pluralize(camelized);
28   },
29
30   /**
31    * Does a POST to create a record.
32    *
33    * By default a json object containing the model type etc. is
          generated and sent.
34    * HAL only needs the attributes of the model.
35    *
36    * @param store
37    * @param type
38    * @param snapshot
39    * @returns {*}
40    */
41   createRecord(store, type, snapshot) {
42     let serializer = store.serializerFor(type.modelName);
```

```
43      let url = this.buildURL(type.modelName, null, snapshot, '
            createRecord');
44      let data = serializer.serialize(snapshot);
45
46      return this.ajax(url, "POST", { data: data.data.attributes });
47    },
48
49    /**
50     * Updates a record.
51     *
52     * By default a json object containing the model type etc. is
            generated and sent.
53     * HAL only needs the attributes of the model.
54     * Also the PATCH verb should be used, to only update changed
            attributes.
55     *
56     * @param store
57     * @param type
58     * @param snapshot
59     * @returns {*}
60     */
61    updateRecord(store, type, snapshot) {
62      let serializer = store.serializerFor(type.modelName);
63
64      let data = serializer.serialize(snapshot);
65
66      let id = snapshot.id;
67      let url = this.buildURL(type.modelName, id, snapshot, '
            updateRecord');
68
69      // Don't execute a request if no attributes have changed
70      if (Ember.isEmpty(data.data.attributes)) {
71        return null;
72      }
73
74      return this.ajax(url, 'PATCH', { data: data.data.attributes });
75    },
76
77    /**
78     * Generates an url for `store.queryRecord(type, query)` call.
```

```
79    *
80    * @param query
81    * @param modelName
82    * @returns {String} url
83    */
84   urlForQueryRecord(query, modelName) {
85     return this._urlForQuery(query, modelName);
86   },
87
88   /**
89    * Generates an url for `store.query(type, query)` call.
90    *
91    * @param query
92    * @param modelName
93    * @returns {String} url
94    */
95   urlForQuery(query, modelName) {
96     return this._urlForQuery(query, modelName);
97   },
98
99   /**
100   * Helper to build url for query call.
101   * Either returns the default url or a customized one if the query
        contains a `searchEndpoint` property.
102   *
103   * @param query
104   * @param modelName
105   * @returns {String} url
106   * @private
107   */
108  _urlForQuery(query, modelName) {
109    let url = this._buildURL(modelName);
110
111    if(!query.hasOwnProperty("searchEndpoint")) {
112      return url;
113    }
114
115    let searchEndpoint = query.searchEndpoint;
116
```

```
117     // Property has to be removed from query, so the request won't
            contain it.
118     delete query.searchEndpoint;
119
120     return url + SEARCH_PREFIX + "/" + searchEndpoint;
121   }
122 });
```

Literaturverzeichnis

[Bitkom 2016] Bitkom: Nutzung von Cloud Computing in Unternehmen in Deutschland in den Jahren 2011 bis 2015, Statista. (2016), May. – URL https://de.statista.com/statistik/daten/studie/177484/umfrage/einsatz-von-cloud-computing-in-deutschen-unternehmen-2011/. – Zugriffsdatum: 2016-10-11

[Docker a] Docker: *Attach services to an overlay network.* – URL https://docs.docker.com/engine/swarm/networking/. – Zugriffsdatum: 2016-11-07

[Docker b] Docker: *Overview of Docker Compose.* – URL https://docs.docker.com/compose/overview/. – Zugriffsdatum: 2016-11-07

[Docker c] Docker: *Swarm mode key concepts.* – URL https://docs.docker.com/engine/swarm/key-concepts/. – Zugriffsdatum: 2016-11-07

[Docker d] Docker: *Swarm mode overview.* – URL https://docs.docker.com/engine/swarm/. – Zugriffsdatum: 2016-11-07

[Docker e] Docker: *Understand images, containers, and storage drivers.* – URL https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/. – Zugriffsdatum: 2016-11-07

[Docker 2016] Docker: *What is Docker?* July 2016. – URL https://www.docker.com/what-docker. – Zugriffsdatum: 2016-10-12

[Ember.js 2016] Ember.js: *Ember.js - Core Concepts.* August 2016. – URL https://guides.emberjs.com/v2.7.0/getting-started/core-concepts/. – Zugriffsdatum: 2016-10-12

[Eurostat 2014] Eurostat: Gründe, welche einer vermehrten Nutzung von Cloud Computing Diensten von Unternehmen in der EU28 entgegenstanden im Jahr 2014, Statista. (2014), December. – URL https://de.statista.com/statistik/daten/studie/

274143/umfrage/umfrage-zu-den-gruenden-gegen-die-nutzung-von-cloud-computing/. – Zugriffsdatum: 2016-10-11

[Experton 2016] Experton: B2B-Marktvolumen von Cloud-Computing-Services (SaaS, PaaS, IaaS) in Deutschland von 2011 bis 2015 und Prognose für 2016 (in Millionen Euro), Statista. (2016), January. – URL https://de.statista.com/statistik/daten/studie/165458/umfrage/prognostiziertes-marktvolumen-fuer-cloud-computing-in-deutschland/. – Zugriffsdatum: 2016-10-11

[Fowler] Fowler, Martin: *What are Microservices?*. – URL https://martinfowler.com/microservices/#what. – Zugriffsdatum: 2016-09-17

[Fowler 2015] Fowler, Martin: *Microservice Trade-Offs*. July 2015. – URL https://martinfowler.com/articles/microservice-trade-offs.html. – Zugriffsdatum: 2016-09-17

[Gartner 2016] Gartner: Umsatz mit Software-as-a-Service (SaaS) weltweit von 2010 bis 2016 (in Milliarden US-Dollar), Statista. (2016), January. – URL https://de.statista.com/statistik/daten/studie/194117/umfrage/umsatz-mit-software-as-a-service-weltweit-seit-2010/. – Zugriffsdatum: 2016-10-11

[IAO 2010a] IAO, Fraunhofer: Haben Sie ein eigenes Software as a Service-Angebot in Planung oder bereits in Umsetzung?, Statista. (2010). – URL https://de.statista.com/statistik/daten/studie/163190/umfrage/umsetzungsplaene-deutscher-it-anbieter-fuer-software-as-a-service-angebote/. – Zugriffsdatum: 2016-10-11

[IAO 2010b] IAO, Fraunhofer: Wie schätzen Sie die Bedeutung der folgenden geschäftsstrategischen Vorteile zu SaaS-Angeboten aus der Sicht eines Anbieters ein?, Statista. (2010). – URL https://de.statista.com/statistik/daten/studie/163274/umfrage/einschaetzung-zu-den-vorteilen-von-saas-angeboten-durch-it-anbieter/. – Zugriffsdatum: 2016-10-11

[Koch 2009] Koch, Florian: Leitfaden für SaaS-Anbieter. (2009). – URL https://www.bitkom.org/noindex/Publikationen/2009/Leitfaden/Leitfaden-fuer-SaaS-Anbieter/Leitfaden-SaaS-20090310-web-neu1.pdf. – Zugriffsdatum: 2016-10-11

[Larsson 2015] Larsson, Magnus: *An operations model for Microservices*. 2015. – URL http://callistaenterprise.se/blogg/teknik/2015/03/25/an-operations-model-for-microservices/. – Zugriffsdatum: 2016-09-04

[Spoiala u. a. 2016] Spoiala, C. C. ; Calinciuc, A. ; Turcu, C. O. ; Filote, C.: Performance comparison of a WebRTC server on Docker versus virtual machine. In: *2016 International Conference on Development and Application Systems (DAS)*, URL http://ieeexplore.ieee.org/document/7492590/?arnumber=7492590&queryText=docker&newsearch=true. – Zugriffsdatum: 2016-10-24, May 2016, S. 295–298

[Villamizar u. a. 2015] Villamizar, M. ; Garcés, O. ; Castro, H. ; Verano, M. ; Salamanca, L. ; Casallas, R. ; Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: *2015 10th Computing Colombian Conference (10CCC)*, URL http://ieeexplore.ieee.org/document/7333476. – Zugriffsdatum: 2016-09-10, Sept 2015, S. 583–590

Glossar

A | B | C | E | H | J | L | M | N | O | P | R | S | T | U | V

A

Advanced Message Queuing Protocol

Netzwerkprotokoll, welches zur Kommunikation von Message-Brokern und den entsprechenden Clients genutzt wird. 15, 79

Amazon S3

Amazon Simple Storage Service (S3) bietet Speicher über typische Web Schnittstellen. Neben Amazon selbst existieren mittlerweile auch andere Anbieter, welche S3 kompatible Schnittstellen anbieten. 79

AMQP

Advanced Message Queuing Protocol. 15, 19, 83

API

Application Programming Interface. 13–15, 17, 19, 23, 28, 36, 81, 84

B

Book

Ein Book stellt eine Zusammenstellung von Bildern dar. Im digitalen Kontext können diese auch Videos enthalten. 11

Booker

Ein Booker ist Zuständig für die Belange der Models und Kunden. 10, 11

Bucket

Stellt einen Container für Objekte in Amazon S3 dar. 28, 83

Bundle

Format zum Bündeln von Docker Compose Systemen, um diese dann in einem Swarm System als Stack ausführen zu können. Bundles verhalten sich zu Stacks, wie Images zu Containern. Ein Bundle enthält unter anderem Informationen über die Images, Netzwerke und Volumes des Compose Systems. Beim Ausführen des Stacks werden die Elemente des Bundles über den Swarm verteilt ausgeführt bzw. erstellt. 30

C

Circuit Breaker

Überprüft die Verfügbarkeit eines Dienstes. Sollte der Dienst nicht verfügbar sein, so wird keine Anfrage des ausführenden Dienstes an den nicht verfügbaren Dienst durchgeführt sondern eine alternative Operation ausgeführt. Dies ermöglicht schnellere Antworten, ohne dass beispielsweise auf ein Timeout gewartet werden muss. 13

Cross Site Request Forgery

Als Cross Site Request Forgery Lücke wird eine Anfrage auf einen Server bezeichnet, welche durch eine böswillige Seite dem Besucher untergeschoben wird. Durch den Cookie vom User wird diese Anfrage beim Ziel Server als gültig akzeptiert und es wird ohne entsprechenden Schutz eine Aktion ausgeführt. 16, 80

CRUD

Create, Read, Update, Delete. 10, 21

CSRF

Cross Site Request Forgery. 16

E

Eventual consistency

Die "schließliche Konsistenz" besagt, dass ein Datensatz in einem verteilten System nach einer unbekannten Zeit im gesamten System konsistent werden kann, insofern in diesem Zeitraum keine Änderungen durchgeführt werden. 5

H

HAL

HAL (lang **Hypertext Application Language**) ist eine Spezifikation nach dem HATEOAS Konzept. Es beschreibt wie die Antworten einer Schnittstelle aufgebaut sein müssen, sodass ein HAL kompatibler Client mit jeglicher HAL API ohne viel Anpassung arbeiten und durch die Schnittstelle navigieren kann. 15–18, 21, 23, 28, 34, 35

HATEOAS

HATEOAS steht für **Hypertext As The Engine Of Application State** und ist ein Konzept für REST Schnittstellen. Es verlangt, dass der Weg durch eine API alleine über den Hypertext möglich ist. 15, 16, 27, 34, 81

Hibernate

ORM Framework für Java, welches das JPA Interface implementiert. 17, 23

J

Java Database Connectivity

Einheitliche Schnittstelle zu Datenbanken für Java. 9, 81

Java Persistence API

Spezifikation für Java, welche Java Objekte in Form von persistenz Entitäten objektrelational abbilden und abfragen kann. Die Entitäten stellen hierbei einfache Java Objekte dar, welche auf eine Tabelle in der Datenbank abgebildet werden. 16, 81

JDBC

Java Database Connectivity. 9

JPA

Java Persistence API. 16, 21, 23, 81

JSON-API

JSON-API ist Spezifikation zur Erstellung von APIs in JSON. 28

JVM

Java Virtual Machine. 35, 36

L

Load Balancer

Verteilt eine Anfrage auf mehrere Server, um die Last gleichmäßig zu verteilen.. 13, 32, 84

M

Multi-Tenancy-Architektur

Die **Multi-Tenancy-Architektur** stellt eine Architektur dar, welche entgegen einer **Single-Tenancy-Architektur** alle Nutzer auf der selben Plattform arbeiten lässt. Es findet hierbei eine virtuelle Trennung der Nutzerdaten statt, anstatt dass jeder Nutzer sein eigenes System besitzt. 15, 25, 35

MySQL

Relationale Datenbank, die aktuell von Oracle entwickelt wird. 20, 24, 29, 31, 32

N

NoSQL

Kurzform von "Not only SQL". NoSQL Datenbanken verfolgen einen nicht relationalen Ansatz und benötigen meist kein festes Schema. In der Regel sind diese horizontal Skalierbar, während relationale Datenbanken eher vertikal skalieren. 20, 83

O

OAuth

Offenes Protokoll zur Schnittstellen-Autorisierung für Anwendungen. Ein Benutzer kann über das OAuth Protokoll einem Client den Zugriff auf eine andere Anwendung (Server) erteilen. Dies geschieht, ohne dass der Client die Zugangsdaten des Benutzers erhält, da die Authentifizierung auf dem Server stattfindet. Die Identifikation des Nutzers findet nur noch über ein Access Token statt, welches in der Regel ohne Benutzer Interaktion vom Client durch ein Refresh Token erneuert werden kann. 13–16, 21, 27, 35, 82

OAuth2

Version 2 vom OAuth Protokoll. 14–16, 19–22, 24, 27, 28, 36

Objekt-Relational Mapping

Objektrelationale Abbildung. 17, 83

Option

Stellt eine Art Reservierung des Models dar. Sollte es weitere interessenten im gleichen Zeitraum erhalten, so wird mit aufsteigendem Zähler eine weitere Option erteilt. Fällt die erste Option weg, so rücken die weiteren Optionen um einen Platz auf. 11

ORM

Objekt-Relational Mapping. 17, 81

P

Pojo

Plain Old Java Object. 22

Policy

Regelt die Zugriffsberechtigung auf Buckets entweder global oder für bestimmte Prefixe. 28

R

RabbitMQ

Open-Source Message Broker, welcher das AMQP Protokoll implementiert. 19, 20

Redis

Open-Source NoSQL In-Memory-Datenbank, die eine Key-Value Datenstruktur nutzt. 20

REST

REST steht für **Representational State Transfer** und beschreibt eine Architektur für Schnittstellen, welche unter anderem zustandslos und einheitlich ist. 15–17, 21, 23, 27, 28, 81

Reverse Proxy

Liefert Daten aus einem internen Netzwerk an Clients aus. Ist somit zentraler Eingangspunkt in das Netzwerk der Server. 17

S

SaaS

Software as a Service. 3–5, 12, 16, 21, 34, 36

Service Discovery

Automatische Erkennung von anderen Diensten über ein festgelegtes Protokoll. Kann sowohl dezentral als auch zentral mit einem Discovery Server sein. 15, 17

Single Sign On

Der Benutzer authentifiziert sich mit seinem Client an nur einer Stelle und kann so Zugriff auf verschiedene Portale erlangen. Als Beispiel können Dienste wie Google oder GitHub genannt werden, welche auf vielen Webseiten zur Authentifizierung genutzt werden können. 19, 84

SSO

Single Sign On. 19, 27, 28

Sticky Session

Wird in Verbindung mit Load Balancern genutzt, um eine Bindung zwischen einem Benutzer und dem Ziel Server zu schaffen. Der Nutzer kann über seine HTTP Session oder die IP Adresse immer zum gleichen Server geleitet werden. 20

T

Traverson

Traverson ist eine Client Komponente des Spring HATEOAS Projekts, welche von der gleichnamigen JavaScript Bibliothek[1] inspiriert wurde. Es dient der vereinfachten Navigation durch Hypermedia APIs. 34

U

Universally Unique Identifier

Allgemein besteht eine UUID aus verschiedenen Bits eines Zeitstempels, einem Versionsbezeichner, Bits einer Clocksequenz und einer eindeutigen Node Identifikationsnummer. Aufgrund dieser Zusammensetzung ist eine Kollision in einem verteilten System trotz fehlender zentraler Koordination sehr unwahrscheinlich. 23, 84

UUID

Universally Unique Identifier. 23

[1] https://github.com/basti1302/traverson

V

Volume

In Bezug auf Docker, ist ein Volume ein persistenter Datenspeicher, welcher den Erhalt von Daten auch nach dem Löschen eines Containers oder dem Neustart von Docker sicherstellt. 30, 35

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. Februar 2017 Daniel Gehn