Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

CHRISTIAN HAPP
MASTER THESIS

PREPARING THE SWEBLE HUB SOFTWARE FOR THE CLOUD

Eingereicht am 29. September 2016

Betreuer:
Dipl.-Inf. Hannes Dohrn
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 29. September 2016

License

Erlangen, 29. September 2016

Abstract

This master thesis proposes concept and implementation of a microservice-based architecture for the Sweble Hub software. In future, with this microservice-based architecture, Sweble should be deployed in the cloud in order to manage a big workload from many users accessing the wiki. The thesis gives an overview about the microservice architecture pattern and which additional components are necessary because of the distributed setting. A concept is introduced in which way the current architecture of Sweble can be sliced into microservices. From this concept two microservices are implemented. Finally, the concept and implementation of the microservice architecture is evaluated for its suitability. It is shown, that Sweble fits into the microservice pattern. However, microservices are not a silver bullet and with the architecture style some complexities are introduced into the system because of the distributed environment.

Zusammenfassung

Diese Masterarbeit stellt ein Konzept und eine Implementierung einer Microservice Architektur der Sweble Hub-Software vor. In Zukunft soll Sweble mit dieser Microservice Architektur in der Cloud verteilt betrieben werden, um einer großen Belastung stand zuhalten, die von vielen Nutzern, die auf das Wiki zugreifen, verursacht wird. Diese Arbeit gibt einen Überblick über das Microservice Architektur Pattern und welche zusätzlichen Komponenten, die aufgrund der verteilten Umgebung, notwendig sind. Es wird ein Konzept vorgestellt, in welcher Art Sweble in Microservices aufgeteilt werden kann. Zwei der konzipierten Microservices wurden implementiert. Abschließend wird das Konzept und die Implementierung der Microservice Architektur auf ihre Tauglichkeit evaluiert. Es wird gezeigt, dass Sweble in eine Microservice Architektur passt. Allerdings sind Microservices kein Patentrezept und mit dem Architekturstil wird dem System aufgrund der Verteilheit zusätzliche Komplexität hinzugefügt.

Inhaltsverzeichnis

Abkürzungsverzeichnis

BC Bounded Context.

DDD Domain-driven Design.

DI Dependency Injection.

JAR Java Archive.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

JWT JSON Web Token.

REST Representational State Transfer.

UI User Interface.

URI Uniform Resource Identifier.

VM Virtuelle Maschine.

WOM Wiki Object Model.

WRI Wiki Resource Identifier.

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Sweble Wiki Hub ist eine Wiki-ähnliche Web-Anwendung für Wissensmanagement und End-User Programmierung. Das Sweble Wiki Hub kann in mehrere unabhängige Projekte, die an sich ein eigenes Wiki darstellen, unterteilt werden. Ein längerfristig Ziel von Sweble ist, dass es in der Lage ist, mehrere, große Projekte zu hosten. Als Beispielprojekt kann hier die englische Wikipedia genannt werden, die zur Zeit etwa 5240629 Artikel umfasst[1]. Konsequenterweise werden bei der großen Anzahl an Artikel viele Nutzer das Wiki benutzen. Das Wiki soll damit zurechtkommen, dass viele Nutzer auf das Wiki gleichzeitig zugreifen, wie u.a. Artikel abzufragen, neue Artikel zu erstellen, Artikel zu bearbeiten oder nach Artikeln zu suchen. Um diese Last zu stemmen, können Applikationen verteilt in die Cloud deployed werden. In den letzten Jahren, mit dem Aufkommen von Cloud Computing und seinen Vorteilen, erwägen viele Unternehmen ihre Applikationen in die Cloud zu verlagern (Jamshidi, Ahmad & Pahl, 2013). Sweble Wiki Hub weist bis jetzt eine monolithische Architektur auf und wird als Ganzes auf einmal deployed. Es ist eine Single-Tier Applikation, was bedeutet, dass alle Komponenten der Sweble Applikation auf einen einzelnen Server gelegt werden. Dieser monolithische Architekturstil erschwert eine Skalierung in der Cloud (Villamizar u. a., 2015). In den letzten Jahren sind Microservices populär geworden. Viele große Unternehmen, die ihre Anwendung aufgrund tausender Benutzer skalieren müssen, sind auf auf eine Microservice Architektur umgestiegen. Dazu zählen beispielsweise Netflix (Mauro, 2015), Soundcloud (Calcado, 2015), Gilt (Goldberg, 2014) und LinkedIn (Ihde & Parikh, 2015). Eine Microservice Architektur ist einfach in die Cloud portierbar, da diese von Cloud-Umgebungen profitiert, weil nicht-funktionale Eigenschaften, wie Skalierbarkeit, Charakterzüge von Microservices sind (Balalaie, Heydarnoori & Jamshidi, 2015b). Durch die Aufteilung der Anwendung in Microservices kann diese einfach skaliert werden. Bei Lastspitzen, wenn viele Nutzer auf das Wiki zugreifen, können weitere Microservices der Umgebung hinzugefügt werden. Sobald weniger Nutzer auf das Wiki

[1] https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

zugreifen, können Microservices beendet werden, um Ressourcen zu sparen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist ein Konzept vorzustellen, wie das Sweble Wiki Hub in mehrere Microservices aufgeteilt werden kann. Aus diesem Konzept soll ein Teil der Architektur implementiert werden. Anschließend soll das Konzept und die Implementierung evaluiert werden.

1.3 Aufbau der Arbeit

Im zweiten Kapitel wird das Sweble Wiki Hub mit seinen Funktionen vorgestellt. Anschließend werden die Komponenten einer Microservice Umgebung und in welcher Art die Services miteinander kommunizieren können, dargestellt. Das Kapitel endet, wie die einzelnen Services überwachen werden können, damit das Gesamtsystem konsistent bleibt. Das dritte Kapitel widmet sich mit dem Konzept der Microservice Umgebung. Es gibt einen Überblick, welche Möglichkeiten es gibt, wie Microservices an einer zentralen Service Registry registrieren werden können. Danach wird gezeigt, wie Software Projekte modelliert werden können, um aus dem Modell die einzelnen Microservices abzuleiten. Es folgt ein Überblick, auf welche drei Arten Applikationen skaliert werden können. Darauffolgend wird die aktuelle monolithische Architektur des Sweble Wikis aufgezeigt. Aus der aktuellen Architektur wird ein Konzept vorgestellt, wie das Sweble Wiki in Microservices aufgeteilt werden kann. Die Implementierung folgt im vierten Kapitel. Hier wird gezeigt, mit welcher Technologie die Microservices implementiert wurden. Im Anschluss wird erläutert, wie Microservices mit der Service Registry interagieren. Schlussendlich werden die inneren Strukturen der zwei implementierten Microservices aufgezeigt. Eine Evaluation Swebles Microservice Architektur findet im fünften Kapitel statt. Es folgt ein Kapitel, das die Arbeit zusammenfasst. Das letzte Kapitel gibt einen Ausblick, welche nächsten Schritte notwendig sind, um die Microservice Umgebung zu vervollständigen.

2 Grundlagen

Dieses Kapitel geht kurz auf die Hintergründe und Funktionen des Sweble Wikis ein. Folgend werden die Charakterzüge einer Microservice Architektur vorgestellt.

2.1 Sweble Wiki

Sweble ist ein Wiki-System, das in der Programmiersprache Java implementiert wurde. Es ist dem Wiki-System MediaWiki[1] ähnlich. MediaWiki verwendet WikiText als Markup-Sprache um Wikiartikel zu beschreiben. Mit MediaWiki ist es möglich WikiText zu HTML transformieren, eine Repräsentation in eine höhere Ebene wie einen Abstract Syntax Tree bietet es nicht an. Deshalb haben Dohrn und Riehle (2011b) ein eigenes Format (Wiki Object Model (WOM)) zur Beschreibung und Speicherung von Wiki Texten definiert, sowie ein Parser, der WikiText Markup in einen strukturierten, abstrakten Syntaxbaum umwandelt (Dohrn & Riehle, 2011a). Das Sweble Wiki baut auf den Erkenntnissen des Parsers und des WOMs auf.
Sweble Wiki ist im Grunde eine Wiki Software, bietet aber zusätzliche Funktionen ähnlich einer Versionsverwaltungs Software an. Mit Sweble ist es möglich, mehrere Artikel zu bearbeiten und diese anschließend in einen Commit zusammenzufassen. Es ist eine Versionshistorie zur Protokollierung der Änderungen einsehbar. Das Wechseln zwischen verschieden Versionen muss möglich sein. Eine bessere Koordination des gemeinsamen Zugriffs von mehreren Autoren auf dieselbe Datei wird ermöglicht. Dies kann beispielsweise durch Branching umgesetzt werden, damit aus einer Abspaltung einer Version, ein Artikel parallel bearbeitet werden kann. Das Zusammenführen zweier Branches soll folglich auch unterstützt werden. Das ganze Wiki ist in unabhängige Projekte unterteilt. Das Forken von Projekten soll unterstützt werden. D.h. es soll möglich sein, eine Kopie eines Projekts anzulegen um Änderungen vorzunehmen, ohne das ursprüngliche Projekt zu beeinflussen.

[1] https://www.mediawiki.org

Eine zentrale Komponente im Wiki sind Transformationen. Der Kern von Sweble ist so aufgebaut, dass es Ressourcen nur als WOM darstellen kann. Weitere Repräsentationen von Ressourcen sollen durch Plug-ins unterstützt werden. Beispielsweise kann eine Ressource als Wiki-Artikel, Bild, Video, Musik, Metadaten oder Quellcode einer Skriptsprache etc, dargestellt werden. Dies macht es erforderlich, dass eine Ressource zwischen der zentralen Darstellungsform WOM und eine ihrer Representationsausprägung transformiert werden kann. Jedes Plug-in bringt dadurch eine Transformation nach WOM mit. So kann beispielsweise von WOM nach Wiki-Artikeln in HTML transformiert werden.

2.2 Microservices

2.2.1 Eigenschaften

Der Begriff Microservices ist ein Architekturstil, bei dem ein System in kleine, eigenständige Services aufgeteilt wird. Die Services laufen in unabhängigen Prozessen und kommunizieren mit leichtgewichtigen Mechanismen wie Representational State Transfer (REST)-APIs miteinander (Fowler & Lewis, 2014). Ein gegensätzlicher Architekturstil zu Microservices ist ein monolithischer Ansatz, bei der die Anwendungslogik innerhalb einer einzelnen, ausführbaren Einheit ist. Für kleine Systeme kann eine monolithische Architektur eine geeignete Lösung sein. Sobald aber die Größe eines System anfängt zu wachsen, tauchen Probleme wie eine schlechtere Verständlichkeit des Codes, erhöhte Deploymentzeit und eine schwierige Skalierung bei rechenintensiven Arbeitsschritten auf (Richardson, 2014b). Hier schaffen Microservices Abhilfe, die unter anderem folgende Vorteile bieten:

- **Modulierungskonzept**:
  Ein Microservice ist eine für sich abgeschlossene, eigenständige Einheit. Ein Microservice soll sich hierbei auf die Erledigung einer Aufgabe spezialisieren und das soll er gut machen. Um auf Funktionalitäten anderer Services zuzugreifen, muss über eine vorher definierte Schnittstelle zugegriffen werden. Dadurch schleichen sich ungewollte Abhängigkeiten zwischen Services nicht so leicht ein. Der Entwickler muss Schnittstellen und Abhängigkeiten vorher explizit definieren. Dadurch wird eine geringe Kopplung zwischen den Microservices angestrebt. Ohne die strikte Trennung der Microservices über ein Netzwerk kann es leichter passieren, dass auf Klassen anderer Module zugegriffen wird, die architektonisch nicht gewünscht ist.
  Die starke Modularisierung durch Microservices ermöglicht eine leichtere Ersetzbarkeit eines Microservices. Wenn ein Microservice veraltet ist oder der Aufwand zum Abändern zu groß ist, kann ein Microservice schnell neu

geschrieben werden. Hier muss natürlich darauf geachtet werden, dass Microservices möglichst klein gehalten werden sollen. Einen sinnvollen Richtwert im Bezug auf die Größe eines Microservice gibt Eaves (2014): Die Codebasis sollte möglichst kein sein. Ein Service sollte in zwei Wochen neu geschrieben werden können.

- **Komfortables Deployment**:
  Wenn in einer monolithischen Anwendung nur eine Codezeile geändert wird und eine neue Version ausgeliefert werden soll, muss die gesamte Anwendung neu deployed werden. Wird allerdings eine Microservice Architektur verwendet, wird ein Microservice unabhängig vom übrigen System deployed. Dadurch kann ein Service schneller deployed und die neue Funktionalität einem Kunden zeitnah zur Verfügung gestellt werden. Falls Probleme im Deployment auftreten, können diese schneller dem jeweiligen Microservice zugeordnet werden.

- **Skalierung**:
  Bei einem monolithischen System ist die Skalierung eher schwierig. Wenn nur ein kleiner Teil des Gesamtsystems unter hoher Last steht und nicht leistungsfähig genug ist, muss das System als Ganzes skaliert werden. Bei Einsatz einer Microservice Architektur kann allerdings jeder Service eine eigene Skalierung vornehmen. Die Leistungsfähigkeit des betroffenen Services kann z. B. durch die Anzahl der Microservices erhöht werden, indem die Last auf diese verteilt wird. Hier kann Commodity Hardware eingesetzt werden, die relativ billig, allgemein verfügbar und gut austauschbar ist.

- **Technolgische Wahlfreiheit**:
  Jeder Microservice kann entsprechend seiner Anforderungen und Problemen seine eigene Technologie einsetzen. Die Microservices können verschiedene Sprachen, Plattformen oder Frameworks verwenden. Für unterschiedliche Anforderung kann jeder Microservice eine andere Persistenztechnologie einsetzen. Dadurch kann für jede Aufgabe das am besten geeignete Werkzeug ausgewählt werden.

- **Belastbarkeit**:
  Ausfälle gibt es immer. Werden eine Vielzahl von Microservices betrieben, sind Ausfälle statistisch gesehen mit hoher Wahrscheinlichkeit zu erwarten. Die Microservices agieren als verteilte Anwendung. Rotem-Gal-Oz (2006) zeigt eine Sammlung von Irrtümer, die Programmierer machen, wenn sie inbesondere das erste Mal eine verteilte Anwendung entwickeln. Zu den fehlerhaften Annahmen zählen beispielsweise, dass das Netzwerk ausfallsicher ist, der Datendurchsatz unendlich ist oder die Latenzzeit gleich null ist. Auch wenn versucht wird, die Ursachen für Ausfälle zu reduzieren, sollte immer davon ausgegangen werden, dass ein Ausfall die Normalität ist. Ein Ausfall eines Microservices sollte den Betrieb anderer Microservices nicht

beeinflussen und somit den Betrieb des Gesamtsystems nicht beeinträchtigen. In großen Unternehmen, wie Netflix werden sogar Störungen und Fehler provoziert, um zu sehen, wie sich das Gesamtsystem bei einem Ausfall verhält und ob es den Belastungen standhält. Um einen Ausfall zu simulieren können ebenfalls Programme wie Chaos Monkey[2] eingesetzt werden, das zu bestimmten Uhrzeiten wahllos Rechner abschaltet oder Latenzen für alle Netzwerkpakete einführt.

2.2.2 Komponenten

In einer Microservice Architektur wird die Businesslogik auf mehrere Systeme verteilt. Deshalb werden weitere Komponenten benötigt, welche die Koordination der Services übernehmen. Nach Balalaie u. a. (2015b) sollten folgende Komponenten vorhanden sein, damit alle Vorteile einer Microservices Architektur genutzt werden können.

- **Configuration Server**:
  Ein Prinzip von Continuous Delivery ist die Trennung zwischen Source Code und Konfiguration. Bei Änderungen der Konfiguration ist ein neues Deployment der Anwendung nicht nötig. Die Microservices können ihre entsprechende Konfiguration von einem Configuration Server abrufen.
- **Service Discovery**:
  Üblicherweise rufen Services andere Services auf. In einer monolithischen Anwendung rufen sich Services durch Prozeduraufrufe auf. In traditionellen verteilten Systemen laufen Services in festgelegten, bekannten Adressen und kommunizieren über HTTP/REST oder mit einem RPC Mechanismus miteinander. Microservice basierende Architekturen werden üblicherweise in einer virtualisierten oder containerisierten Umgebung betrieben. Service-Instanzen werden dynamisch Netzwerkstandorte zugewiesen. Auch ändert sich die Anzahl der Service-Instanzen dynamisch aufgrund von Skalierung der Umgebung durch Starten neuer Server oder durch ungewollte Server-Ausfälle. Folglich muss ein Service Discovery Mechanismus verwendet werden, bei dem sich Services mit IP-Adresse und Port an einer Service Registry registrieren. Andere Services können an der Service Registry den Standort eingetragener Services abfragen. Eine Service Registry stellt somit sicher, dass Microservices sich gegenseitig finden können.
- **Load Balancer**:
  Soll eine Anwendung skalierbar sein, muss die Last auf mehrere Instanzen verteilen werden. Der Load Balancer verteilt Anfragen auf mehrere Services, ist nach außen aber eine einzelne Komponente. Der Aufrufer merkt hierbei

[2] https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

nicht, ob ein Service läuft oder mehrere. Die verfügbaren Services fragt der Load Balancer von der Service Discovery Komponente ab.

- **Edge Server**:
  Ein Edge Server ist eine Implementierung des API Gateway Pattern (Richardson, 2014a) und stellt eine externe API der Öffentlichkeit zur Verfügung. Der gesamte, externe Datenverkehr sollte über den Edge Server zu den internen Microservices geroutet werden. Der Edge Server kapselt interne Strukturen der Anwendung. Dadurch ist der Client durch Veränderungen der internen Struktur des Systems nicht betroffen.

2.2.3 Integration und Kommunikation

In monolithischen Anwendungen rufen sich Komponenten gegenseitig mit Funktionsaufrufen, die die Programmiersprache anbietet auf. Bei einer Microservice Architektur ist die Anwendung ein verteiltes System und die Funktionen sind auf mehrere Server verteilt. Deshalb müssen die Microservices mithilfe eines gemeinsamen Protokolls kommunizieren. Im Design einer Microservice Architektur gibt es mehrere Möglichkeiten welche Technologie für die Integration der Microservices eingesetzt wird. Microservices können synchrone Request/Response-basierende Kommunikationsmechanismen wie beispielsweise REST verwenden. Bei einem synchronen Request/Response basierenden Mechanismus sendet der Client eine Anfrage an einen Service. Der Service verarbeitet die Anfrage und sendet eine Antwort zurück. Beim Client wird der Thread, der die Anfrage sendet, blockiert, während er auf eine Antwort wartet. Ein bekanntes Protokoll ist REST. REST ist ein Architekturstil für verteilte Systeme und ein Begriff für die grundlegenden Ansätze des World Wide Webs. Die fünf Kernprinzipien dieser Architektur sind folgende (Fielding, 2000).

- Das zentrale Konzept einer REST Architektur sind Ressourcen. Jede Ressource wird über eine Uniform Resource Identifier (URI) eindeutige identifiziert. Eine URI ist global eindeutig.
- Eine Ressource kann in mehreren Repräsentationen dargestellt werden. Die Ressource ist selbst nie sichtbar, sondern lediglich eine ihrer Repräsentationen. Dargestellt wird die Ressource in einem bestimmten Format wie beispielsweise HTML oder JavaScript Object Notation (JSON). Durch Content Negotiation kann ein Client festlegen, welche Datenrepräsentation er verarbeiten kann.
- Der Zugriff oder die Manipulation von Ressourcen erfolgt über eine einheitliche Schnittstelle mit Standardmethoden. Bei HTTP zählen zu diesen Methoden beispielsweise u.a. GET zum Abfragen von Ressourcen, POST zum Erstellen neuer Ressourcen und DELETE, um Ressourcen zu löschen.

Zusätzlich sind noch Garantien über das Verhalten der Methoden definiert. Ein GET-Request sollte beispielsweise idempotent sein.

- Beziehung zwischen Ressourcen können durch Links ausgedrückt werden. Eine Ressource kann auf URIs anderer Ressourcen verweisen.
- Der Server sollte zustandslos sein. Ein Request des Clients muss alle relevanten Informationen enthalten, die der Server zum Verständnis und zur Verarbeitung benötigt. Der Server muss keine clientspezifischen Details über mehrere Requests speichern. Die Kopplung zwischen Client und Server wird verringert, da aufeinanderfolgende Anfragen auch von unterschiedlichen Servern bearbeitet werden können. Dadurch wird die Skalierbarkeit des Servers vereinfacht.

Alternativ können Messaging-Systeme für eine asynchrone, nachrichtenbasierende Kommunikation eingesetzt werden. Services kommunizieren durch asynchronen Austausch von Nachrichten. Ein Client sendet eine Nachricht an einen Messaging-Server. Wenn es nötig ist, dass der Server dem Client antworten muss, antwortet der Server mit einer separaten Nachricht. Da die Kommunikation asynchron ist, wird der Client nicht blockiert. Der Client geht davon aus, dass er eine Antwort nicht sofort empfangen wird. Der Message Server verwaltet Queues, die von Clients befüllt werden. Beispielsweise kann mit einem Messaging-System Worker-Queues oder ein Publish-Subscribe Mechanismus implementiert werden. Bei einer Publish-Subscribe Queue gibt es Producer, die die Queue mit Nachrichten befüllen. Beliebige Consumer können sich an der Queue anmelden, um Nachrichten zu erhalten. Jede Nachricht, die von einem Producer geschickt wird, wird an alle angemeldeten Consumer weitergeleitet. Dafür müssen sich Producer und Consumer nicht kennen und sind gegenseitig entkoppelt. Dem Producer muss nicht unbedingt bekannt sein, wie viele Consumer seine Nachricht erhalten.
Ein Messaging-System benötigt einen Messaging-Server, der eine weitere Komponente in der Infrastruktur ist. Der Message-Server muss ausfallsicher betrieben werden, da sonst die gesamte Kommunikation in der Microservice-Umgebung ausfallen kann. Deshalb garantieren Message-Lösungen eine hohe Verfügbarkeit, beispielsweise durch Replikation.
Es gibt eine Vielzahl von Messaging-Systemen. Eine davon ist RabbitMQ[3], welche das Advanced Message Queuing Protocol (AMQP) implementiert.

2.2.4 Monitoring und Logging

Microservices haben nicht nur Vorteile. Die Aufspaltung der Anwendung in mehrere Microservices verkompliziert das Monitoring des Gesamtsystems. Es kann vorkommen, dass ein Problem nicht nur auf eine Komponente beschränkt ist,

[3]https://www.rabbitmq.com

sondern auf mehrere Microservices verteilt ist und dadurch die Fehlersuche erschwert wird. Die Log-Dateien sind auf mehrere Microservices verteilt und können schwer ausgewertet werden. Wolff (2015) empfiehlt daher, einen Log-Server in einer Microservice Architektur einzusetzen. Microservices können eigene Log-Daten an den Log-Server über das Netzwerk verschicken. Hierbei sollte bei den Microservices auf gemeinsame Standards, wie einheitliche Log-Levels zurückgegriffen werden.
Um den Zustand der Microservices zu überwachen, sollten Metriken eingesetzt werden. Wie bei Protokolldateien müssen Metriken zusammengeführt und ausgewertet werden. Metriken sind Kennzahlen, die den Zustand eines Microservices angeben. Wird ein bestimmter Wert unter- bzw überschritten weist dies auf ein Problem hin und es kann darauf reagiert werden. Dies sollte möglichst automatisiert erfolgen, wie beispielsweise das Starten neuer Instanzen bei zu hoher Last. Das Sammeln von Kennzahlen sollte über einen längeren Zeitraum stattfinden, damit sich Muster abzeichnen können, weil das vereinzelnde überschreiten einer Kennzahl nicht unmittelbar ein Problem anzeigen muss. Alle Microservices sollten grundlegende Monitoring Informationen bereitstellen (Wolff, 2015), damit ein Überblick über den Zustand des Gesamtsystems erfasst wird. Newman (2015) empfiehlt, dass ein Microservice mindestens Kennzahlen wie Antwortzeiten und Fehlerquoten zur Verfügung stellt. Nicht nur für den Betrieb des Systems sind Kennzahlen und Log-Daten sinnvoll, um Probleme festzustellen, sondern auch für Entwickler, die verstehen wollen, wie das System funktioniert, wie der Anwender das System verwendet und wie sich dementsprechend das System verhält.

3 Architektur und Design

3.1 Architekturentscheidung

Beim Einsatz von Microservices sind mehrere Architekturentscheidungen zu treffen, die alle ihre Vor- und Nachteile haben. Aufgrund von bestehenden Anforderungen muss abgewägt werden, welche Entscheidung getroffen wird.

3.1.1 Service Discovery

Ein wichtiger Bestandteil einer Microservice Architektur ist die Service Discovery. Das Auffinden eines Microservices kann in zwei Schritte unterteilt werden. Als erstes muss ein Microservice seinen eigenen Standort einer zentralen Service Registry mitteilen. Es wird in der Regel der Host und der Port in der Registry eingetragen. Als zweiter Schritt fragt der Client die Registry unter welchem Standort der gewünschte Microservice erreichbar ist. Beim Aufbau einer Service Registry sind folgende Fragestellungen zu beachten:

- **Monitoring:**
  Zum Anmelden bzw. Abmelden an der Service Registry gibt es unterschiedliche Möglichkeiten. Eine Option ist das Self-Registration Pattern, bei dem Microservices selbst verantwortlich sind, sich an der Registry anzumelden (Abb. 3.1). Jeder Microservice teilt der Service Registry mit, unter welcher IP-Adresse und welchem Port er erreichbar ist. Zusätzlich muss jeder Microservice einen Heartbeat Mechanismus implementieren. Es ist abzuwägen, ob Microservices der Registry nur eine Art Heartbeat-Ping senden oder ob detailliertere Monitoring Informationen mitgeschickt werden, wie beispielsweise die Festplattenauslastung oder ob der Webserver einen erfolgreichen Statuscode zurückliefert. Durch den Heartbeat Mechanismus wird der Registry mitgeteilt, dass der Microservice erreichbar und einsatzbereit ist und es wird verhindert, dass der Microservice automatisch aus der Registry gelöscht wird. Wird ein Microservice beabsichtigt heruntergefahren, meldet er

sich bei der Registry ab, damit er aus der Liste der verfügbaren Microservices entfernt wird.
Ein Nachteil dieses Ansatzes ist, dass sich Microservices an die Service Registry koppeln und dass der Registrierungscode in jeder eingesetzter Programmiersprache implementiert werden muss.

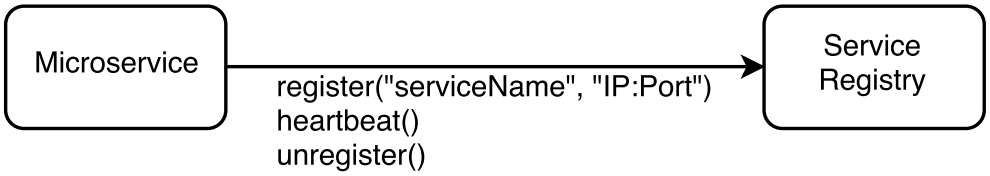

Abbildung 3.1: Self-Registration Pattern.

Ein anderer Ansatz, bei dem sich Microservices nicht mehr selbständig registrieren müssen, ist das Third-Party Registration Pattern (Abb. 3.2). Hier erledigt eine zusätzliche Komponente, das Service Registrar, die Anmeldung. Das Service Registrar muss Änderungen an der Anzahl der laufenden Instanzen durch Überwachen der Deployment Umgebung oder durch Anmeldung zu bestimmten Events verfolgen. Sobald das Registrar eine neue Instanz feststellt, meldet sie diese bei der Service Registry an. Anschließend muss das Registrar die Verfügbarkeit der Microservices mit Healthchecks überprüfen. Das Registrar ist auch für das Abmelden beendeter Instanzen verantwortlich. Ein Nachteil des Ansatzes ist, dass eine weitere Komponente der Umgebung hinzugefügt wird, die eingerichtet und hochverfügbar bereitgestellt werden muss.

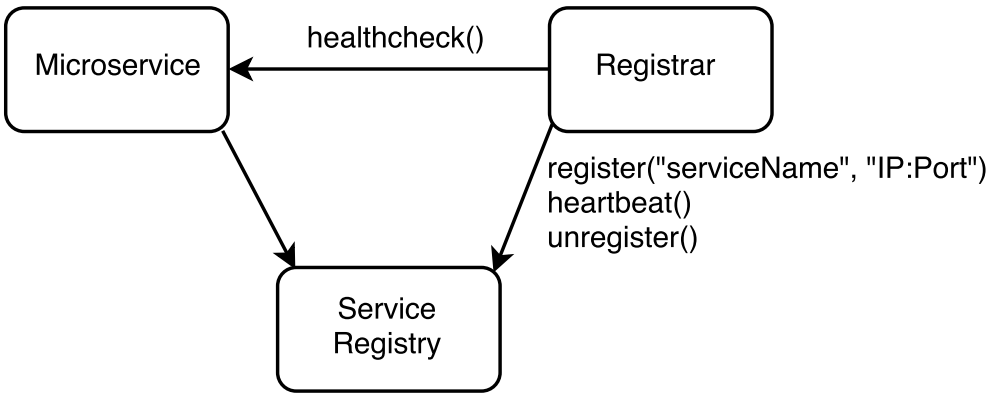

Abbildung 3.2: Third-Party Registration Pattern.

- **Load Balancing:**
  Ein Vorteil einer Microservice Architektur ist die unabhängige Skalierbarkeit einzelner Services. Wenn mehrere Microservice aktiv sind, kann die Last zwischen den Microservices aufgeteilt werden. Service Discovery und

Load Balancing hängen in einer Microservice Architektur stark zusammen. Die Service Registry kennt alle Microservices, der Load Balancer muss zwischen den Microservices die Last verteilen. Deshalb ist das Zusammenspiel der beiden Komponenten wichtig. Dafür gibt es mehrere Möglichkeiten.

1. Ein zusätzlicher Load Balancer verteilt die Last auf die einzelnen Microservices (Abb. 3.3). Der Load Balancer fragt die Service Registry, welche Microservices aktiv sind und verteilt die Last auf diese. Dieser Ansatz hat aber auch Nachteile. Der Load Balancer ist eine weitere Komponente in der Microservice Umgebung und muss konfiguriert und installiert werden. Des Weiteren kann der Load Balancer zum Flaschenhals werden, da alle Requests über den Load Balancer laufen. Deshalb ist ein zentraler Load Balancer für alle Microservices nicht zu empfehlen. Wolff (2015) empfiehlt pro Microservice ein Load Balancer einzusetzen. Ein weiterer Nachteil des Ansatzes ist, dass ein Request an einen Microservice ein zusätzlichen Netzwerk Hop durchlaufen muss und dadurch zusätzliche Latenzzeit zur Bearbeitung des Requests hinzugefügt wird.

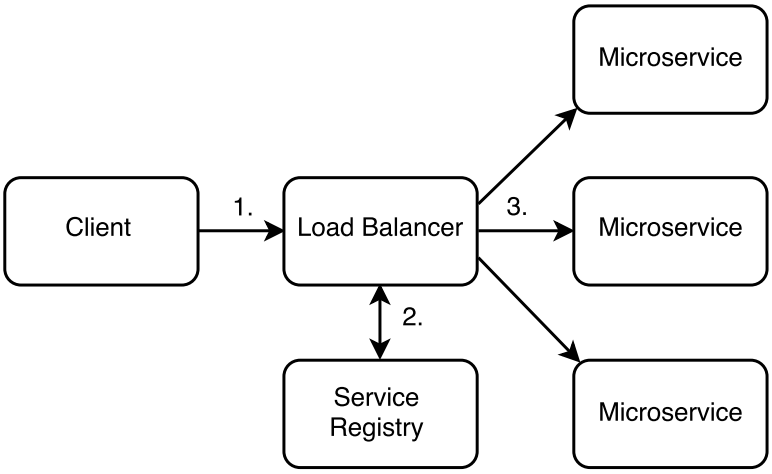

Abbildung 3.3: Zentraler Load Balancer.

2. Eine weitere Möglichkeit ist, dass die Service Registry das Load Balancing übernimmt (Abb. 3.4). Die Service Registry kennt alle aktiven Microservices. Wenn nun der Standort eines Microservice an der Service Registry abgefragt wird, gibt die Service Registry bei jeder Anfrage eine andere Instanz zurück. Die Service Registry muss Algorithmen implementieren die zwischen den Instanzen auswählt. Denkbar wäre eine zufällige Auswahl einer Instanz oder ein Round-Robin Ansatz. Bei letzterem Ansatz wird bei jeder Anfrage die nächste Instanz in der Liste der aktiven Instanzen ausgewählt. Wird das Ende der Lis-

te erreicht, wird am Anfang der Liste fortgesetzt. Im Client bleibt die Instanz aber so lange gültig, bis eine neue Anfrage an die Service Registry gestellt wird. Dadurch ist eine fein granulare Lastverteilung nicht möglich. Es dauert einige Zeit bis ein neu gestarteter Microservice Last abbekommt und bei Ausfall eines Microservices ist die Last schwer zu korrigieren. Die Vorteile dieses Ansatzes sind, dass er mit der vorhandene Service Registry einfach umsetzbar ist und dass keine zusätzliche Software Komponente der Architektur hinzugefügt werden muss.

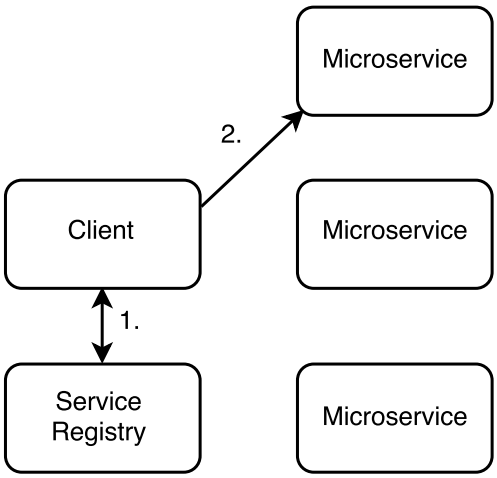

Abbildung 3.4: Load Balancing mit der Service Registry.

3. Die Logik des Load Balancers kann auch im Client Code integriert werden (Abb. 3.5). Der Client holt sich von der Service Registry alle aktiven Microservices und speichert diese Liste lokal ab. Die Liste kann nach einiger Zeit veraltet sein und nicht mehr aktuell sein. Es kann vorkommen, dass der Client Microservices nicht mehr erreichen kann, da diese mittlerweile heruntergefahren wurden. Der Client muss deshalb die Liste alle Microservices innerhalb einer sinnvollen Zeitspanne aktualisieren. Aus der Liste kann der Client anschließend ein Microservice anhand unterschiedlicher Algorithmen, wie beim vorherigen Ansatz, auswählen. Dieser Ansatz hat die selben Vorteile wie der vorherige Ansatz. Allerdings wird der Client an die Service Registry gekoppelt und eine Änderung am Load Balancer muss an alle Clients weitergegeben werden. Wenn Microservices in unterschiedlichen Programmiersprachen implementiert sind, muss für jede Sprache ein eigener Client implementiert werden.

- **Verfügbarkeit der Service Registry:**
  Die Service Registry ist eine zentrale Komponente der Service Discovery

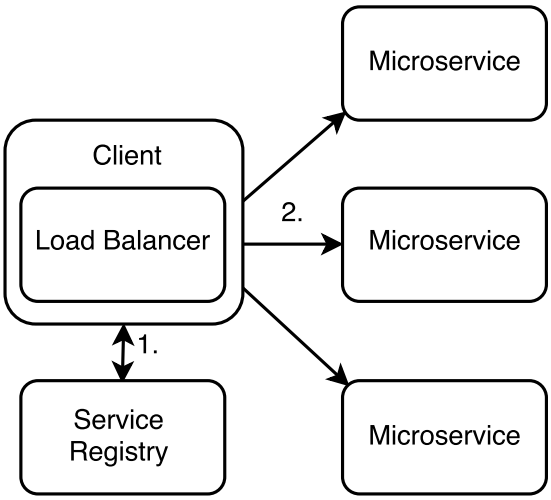

Abbildung 3.5: Client-seitiges Load Balancing.

und muss deshalb hoch verfügbar und auf dem neuesten Stand sein. Folglich besteht die Service Registry aus einem Cluster von mehreren Servern, die ein Replikationsprotokoll verwenden, um Verfügbarkeit und Konsistenz zu bewahren.

Die Verfügbarkeit der Service Registry leitet zum CAP-Theorem über. Nach Gilbert und Lynch (2002) ist es in einem verteilten System unmöglich, die drei Garantien **C**onsistency, **A**vailability und **P**artition Toleranz gleichzeitig zu gewährleisten. Bei einer Störung können nur zwei dieser drei Garantien gleichzeitig erfüllt sein. Es muss abgewägt werden, welche Garantie vernachlässigt werden kann. Consistency besagt, dass alle Replikate die selbe Antwort auf eine Anfrage liefern müssen. Availability bedeutet, dass Anfragen an das System immer beantwortet werden. Partition Toleranz sagt aus, dass das System weiterhin funktionstüchtig ist, auch wenn die Kommunikation zwischen Teilen des Systems nicht mehr möglich ist. Es gibt eine Vielzahl von Service Discovery Lösungen, die entweder als AP- oder CP-System konzipiert sind. Ein CA-System macht in einem verteilten System kein Sinn, da beim Aufgeben der Partition Toleranz keine Kommunikation zwischen Teilen des System möglich ist und es somit nicht über ein Netzwerk betrieben werden kann.

Service Discovery ist ein schwieriges Problem, das allgegenwärtig im Gebiet von verteilten Systemen ist. Nichts veranschaulicht dies besser, als die schiere Anzahl an Projekten, welche versuchen das Problem zu lösen. Es folgt eine Auswahl von vorhanden Lösungen mit ihren jeweiligen Eigenschaften, die als Service Discovery in einer Microservice Umgebung eingesetzt werden können.

- **Eureka**[1] kommt aus dem Netflix-Stack und bietet Service Discovery und

[1] https://github.com/Netflix/eureka

Load Balancing an. Es gibt eine Server Komponente, die eine REST-Schnittstelle zur Kommunikation anbietet. Dazu gibt es eine Java-Bibliothek. Idealerweise sind Microservices auch in Java oder einer Java Virtual Machine (JVM)-kompatiblen Sprache geschrieben. Alternativ kann ein eigener Eureka Client geschrieben werden, der auf die REST-Endpunkte zugreift. Eureka wurde ursprünglich für Amazon Web Services (AWS) von Netflix entwickelt und wird hauptsächlich in Amazons Cloud benutzt. Um den Zustand des Eurekas Servers zu replizieren, wird empfohlen, ein Server pro *Availability Zone* in der Amazon Cloud zu betreiben. Die Server replizieren ihren Zustand durch ein asynchrones Modell, was bedeutet, dass die Instanzen eine leicht unterschiedliche Liste von verfügbaren Instanzen haben (AP-System). Service Registrierung übernimmt jeder Microservice über den Eureka Client selbst. Registrierte Microservices senden alle 30 Sekunden eine "Heartbeat" Nachricht, um den Lease zu erneuern. Wenn ein Microservice den Lease für ein paar mal nicht erneuern kann, wird er aus der Service Registry genommen. Die Registrierungsinformationen werden auf alle Replikate im Cluster verteilt. Die Service Discovery erfolgt auch durch den Client. Er ruft die aktuellen Registrierung vom Server und cached diese lokal. In regelmäßigen Abständen (30 Sekunden) wird die Liste aktualisiert. Der Client unterstützt Round-Robin Load Balancing. Eureka wurde entwickelt, um auch gegen Ausfälle sehr robust zu sein. Es zieht Verfügbarkeit vor starker Konsistenz vor. Wenn es Partitionierung in einem Cluster gibt, geht es zu einem Erhaltungszustand und ermöglicht es weitere Registrierung vor zu nehmen. Wenn die Partition aufgehoben wird, werden die Zuständen der Server zusammengeführt.

- **Consul**[2] ist aus mehreren Komponenten aufgebaut. Es bietet mehrere Key-Features wie Service Discovery, Health Checks, einen Key/Value Store und Multi-Datacenter Unterstützung an. Jeder Knoten, der einen Service anbietet, betreibt einen Consult Agent. Der Agent ist nicht notwendig, um andere Services zu entdecken oder um den Key/Value Store zu erreichen. Er ist verantwortlich für Health Checks der Services, die auf dem Knoten laufen, sowie den Knoten selbst zu überprüfen. Der Agent kommuniziert mit ein oder mehreren Consul Servern. Auf den Consul Servern werden die Daten gespeichert und repliziert. Die Server wählen selbständig einen Leader aus. Die Discovery wird über eine DNS- oder HTTP-Schnittstelle durchgeführt. Anstatt das Microservices aktive Verbindungen mit keep Alive Nachrichten an die Consul Server senden, verwendet Consul eine andere Architektur für Health Checks. Es wird ein Consul Client auf jedem Node im Cluster ausgeführt. Alle Clients sind Teil eines Gossip-Pool[3], das ein verteiltes Health Checking anbietet. Das Gossip Protokoll implementiert

[2] https://www.consul.io
[3] https://www.consul.io/docs/internals/gossip.html

eine Fehlererkennung, die auf Cluster jeder Größe skalieren kann und nicht auf eine ausgewählte Gruppe von Servern konzentriert. Consul ermöglicht anstatt einen primitiven Heartbeat Check, einen umfangreiche Auswahl an Health Checks. Es kann überprüft werden, ob der Webserver einen erfolgreichen Status Code zurückliefert oder ob noch genügend Festplattenkapazitäten vorhanden sind. Auch wenn Consul relativ jung ist und komplexe Algorithmen verwendet, wird es u.a. von Newman (2015) empfohlen.

- **Registrator**[4] setzt das Third-Pary Registration Pattern um (siehe Kapitel 3.1.1). Registrator fungiert als Service Registrar, als Service Registry kann beispielsweise Consul oder Apache ZooKeeper herangezogen werden. Registrator registriert und deregistriert automatisch Services, die als Docker Container deployed werden. Wenn es ein neuen Container gefunden hat, wird dies der Service Registry mitgeteilt. Es ist keine Anpassung des Services innerhalb des Containers nötig. Wenn ein Docker Container mit "published Ports" gestartet wird, erfährt dies Registrator. Der Service Name wird aus Namen des Dockerimage abgeleitet.

- **Apache ZooKeeper**[5] ist ein fehlertoleranter Koordinierungdienst für verteilte Anwendungen. Er bevorzugt Konsistenz vor Verfügbarkeit (CP). Durch mehrere replizierte ZooKeeper Knoten ist Hochverfügbarkeit gewährleistet. Eine typische Größe für eine ZooKeeper Zelle sind drei, fünf oder sieben Knoten. ZooKeeper verwendet das Zab Protokoll um Zustandsänderung im Cluster zu koordinieren. Es gibt ein Leader Replikat und die restlichen Knoten sind Follower-Replikate. Ein Client kann eine neue Session mit jedem beliebigen ZooKeeper Replikat aufbauen. Dadurch wird der Kommunikationsaufwand auf alle Replikate verteilt. Bei zustandändernden Anfragen wird eine Anfrage, an das mit dem Client verbundene Replikat gesendet. Das Replikat sendet die Anfrage an das Leader-Replikat weiter. Der Leader bearbeitet die Anfrage und führt die Zustandsänderung durch. Anschließend wird die Zustandsänderung auf allen Replikaten verteilt und das Replikat sendet eine Antwort zurück an den Client. Lesende Anfragen kann das mit dem Client verbundene Replikat direkt ausführen, da keine Koordination zwischen den ZooKeeper Replikaten notwendig ist.
  ZooKeeper verwaltet die zu speichernden Daten in einem hierarchischen Namensraum. Es werden Knoten in einer Baumdatenstruktur abgelegt. Jeder Knoten im Baum wird in ZooKeeper als ZNode bezeichnet. ZNodes sind eindeutig identifizierbar und können Nutzdaten aufnehmen. ZNodes können verschiedene Eigenschaften besitzen. Es wird zwischen persistente und flüchtigen, auch ephemeral-Nodes genannt, unterschieden. Für jeden Microservice wird ein ephemeral-Node unter dem Namensraum des

[4] http://gliderlabs.com/registrator
[5] https://zookeeper.apache.org

Microservice-Typs erzeugt. Gelöscht werden flüchtige Knoten explizit durch den Client oder automatisch durch ZooKeeper, sobald die Verbindung zum Client beendet oder unterbrochen wird.

3.1.2 Domain-driven Design und Bounded Context

Domain-driven Design (DDD) ist ein Ansatz zur Modellierung komplexer Software. Als erstes wurde DDD von Eric Evans in seinem gleichnamigen Buch eingeführt (Evans, 2004). Das Ziel ist ein Domänenmodell zu erstellen, welches auf der zugrunde liegenden Problemdomäne beruht. Das Verständnis aus den erstellten Modellen soll im Code und in den Softwareartefakten ausgedrückt werden. DDD ist für die Architektur von Microservices wichtig, da es bei der Strukturierung größerer Systeme hilft (Wolff, 2015). Jeder Microservice soll seine eigene fachliche Einheit bilden und lose gekoppelt sein. Eine Änderung in einem Microservice soll also keine Änderung an einem anderen Microservice nach sich ziehen. DDD beschreibt außerdem das Prinzip des Bounded Context (BC). Jede Domäne besteht aus mehreren BCs. Ein BC kapselt Konzepte oder Begriffe, die nur in dem Fachgebiet einen Sinn machen oder in anderen Kontexten eine andere Bedeutung haben. Ein BC stellt aber auch Dinge zur Verfügung, die mit anderen BCs geteilt werden müssen. Hierfür werden klare Schnittstellen definiert, die auflisten, welche Modelle anderen Kontexten zur Verfügung gestellt werden. Die Modellierung eines Softwaresystems mit BCs ist die Basis für fachliche Schnitte für die Microservices. Jeder BC kann in ein oder mehrere Microservices aufgeteilt werden. Dass ein Microservice mehrere BCs enthält, soll vermieden werden, da die Implementierung mehrerer neuer Features in einem Microservice durchgeführt wird, was dem Ziel widerspricht, dass Features unabhängig entwickelt werden sollen (Wolff, 2015). Zur besseren Übersichtlichkeit können Abhängigkeiten und das Zusammenspiel der BCs in einer Context Map visualisiert werden. Evans (2004) definiert mehrere Kollaborationen, die zwischen BCs bestehen können, wie beispielsweise den Shared Kernel, Open Host Service, Customer/Supplier, Conformist, Anticorruption Layer. Bei einem Shared Kernel teilen sich die Domänenmodelle gemeinsame Elemente und in anderen Bereichen unterscheiden sie sich. Bei einem Open Host Service legt ein BC ein Protokoll fest, damit jeder andere BC seine Dienste nutzen kann. Dies ist besonders wichtig, wenn eine Integration mit vielen anderen BCs notwendig ist.

3.1.3 Skalierung

Abbott und Fisher (2009) beschreiben Skalierung mit Hilfe eines dreidimensionalen Modells (Abb. 3.6). Der sogenannte Scale Cube besteht aus einer x-, y- und z-Achse. Skalierung in Richtung der x-Achse steht für eine typische horizontale

Skalierung (scale out). Ein Load Balancer teilt die Last auf alle Microservice Instanzen auf. Laufen N Instanzen, verarbeitet jeder Microservice 1/N der Last. Die Microservices sollten hierbei zustandslos sein. Dies ist ein einfacher und häufig eingesetzter Ansatz um Anwendungen zu skalieren. Zusätzlich kann noch eine automatische Skalierung vorgenommen werden, bei dem sich das System dynamisch an die steigende/ fallende Last anpasst. Wird eine Lastspitze erreicht, sollten zusätzliche Microservice Instanzen gestartet werden, um die Last auszugleichen. Das Starten oder Stoppen von Instanzen sollte möglichst automatisiert ablaufen, da manuelle Eingriffe zu aufwendig wären (Wolff, 2015). Automatische Skalierung kann mithilfe von Metriken umgesetzt werden. Microservices liefern die Anzahl oder die Antwortzeiten von Anfragen einer zentralen Monitoringschnittstelle mit, die entscheidet, ob neue Microservices gestartet bzw. gestoppt werden.
Skalierung in Richtung der y-Achse beruht auf dem üblichen Ansatz von Microservices. Anstatt alle Funktionalitäten einer Anwendung in eine einzelne Applikation zu packen, wird die Anwendung in mehrere Anwendungen zerlegt, die möglichst unabhängig voneinander sind. Die Zerlegung der Anwendung sollte durch vertikale Schnitte erfolgen; d.h. Jeder dieser "Vertikalen" gehört zu einem Team, hat sein eigenes Frontend und Backend. Gemeinsamer Code zwischen den Verticals sollte nicht erlaubt sein. Durch diese Decomposition ist eine einzelne Anwendung nur für eine oder mehrere verwandte Funktionen verantwortlich und die Anwendung kann somit gut skaliert werden.
Richtung z-Achse wird Sharding genannt. Beim Sharding wird die zu verwaltende Datenmenge aufgeteilt und jeder Microservice ist nur für eine Teilmenge des Datensatzes zuständig. Üblicherweise wird diese Methode bei der Skalierung von Datenbanken eingesetzt, kann aber auch für Microservices eingesetzt werden.

3.2 Bisherige Architektur des Sweble Wikis

Ist geplant, eine System-Architektur eines Monolithen auf eine Microservice Architektur umzustellen, sollte zu Beginn die aktuelle System-Architektur anhand von visuellen oder textuellen Beschreibungen verstanden werden (Balalaie, Heydarnoori & Jamshidi, 2015a). Um die innere Struktur eines System zu verstehen, sind die wichtigsten Komponenten und angebotenen Services des Systems zu erfassen. Dies gibt einen Hinweis, welche Komponenten Services anbieten und welche Komponenten auf diese Services zugreifen. Es werden die dynamischen Strukturen der einzelnen Komponenten und dadurch auch die gesamte Businesslogik der Anwendung besser verstanden. Ebenfalls sollte der Technologie Stack der Anwendung klar sein, um einen Überblick über eingesetzte Datenbanktechnologie, Middleware Technologien und third-party Libraries, die im Projekt verwendet werden, zu erhalten. Das folgende Kapitel gibt diesen Überblick über die Architektur von Sweble.

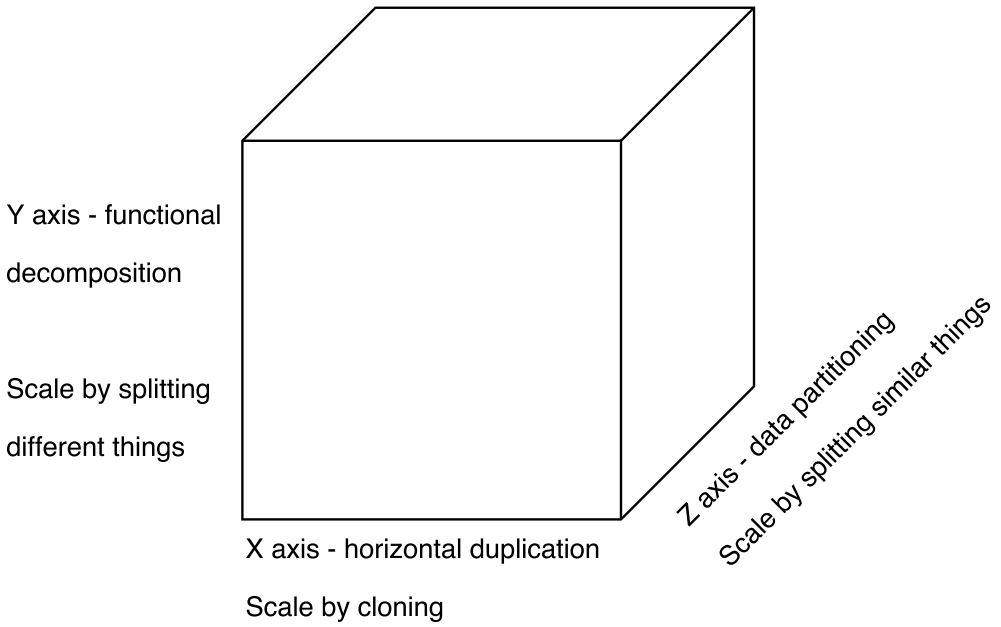

Abbildung 3.6: Skalierung in drei Dimensionen nach (Abbott & Fisher, 2009).

Maven bietet die Möglichkeit an, ein größeres Projekt in mehrere Module aufzuteilen. Sweble Wiki ist anhand von mehreren Maven Modulen hierarchisch unterteilt. Ein Maven Modul kann eine Library oder ein Sweble Module sein. Abbildung 3.7 zeigt die Maven Module von Sweble und ihre jeweiligen Abhängigkeiten. Sweble ist in eine 3-Schichtenarchitektur (Data Access, Domain Logic und Presentation Layer) unterteilt. Die Data Access Layer kapselt Mechanismen, die zur Datenspeicherung und Abfrage notwendig sind. In dieser Schicht ist unter anderem die Library persistence-common, welche Hilfsklassen enthält, die den Zugriff auf die Datenbank erleichtern und konfigurieren. Die Library wird von allen Modulen importiert, die Daten in der Datenbank abspeichern. Als Datenbankmanagementsystem werden die In-Memory Datenbank H2 und PostgreSQL unterstützt. Jedes Sweble Modul das auf die Datenbank zugreift, hat sein Schemata und somit seinen eigenen Namensraum. Folgende Sweble Module werden der Data Access Layer zugeordnet:

- *system-persistence:* Persistenzschnittstelle zum Abspeichern und Abfragen von Projekten, Usern, Rollen und Authentifikationstokens.
- *wom-new-persistence:* Persistenzschnittelle zum Commiten von Änderungsständen, Erstellen von Commitreferenzen, Abragen von Commithistorien und Ressourcen und zum Forken von Projekten.
- *simple-search-backend:* Baut Suchindizes auf der Datenbank auf, damit das Wiki durchsucht werden kann.

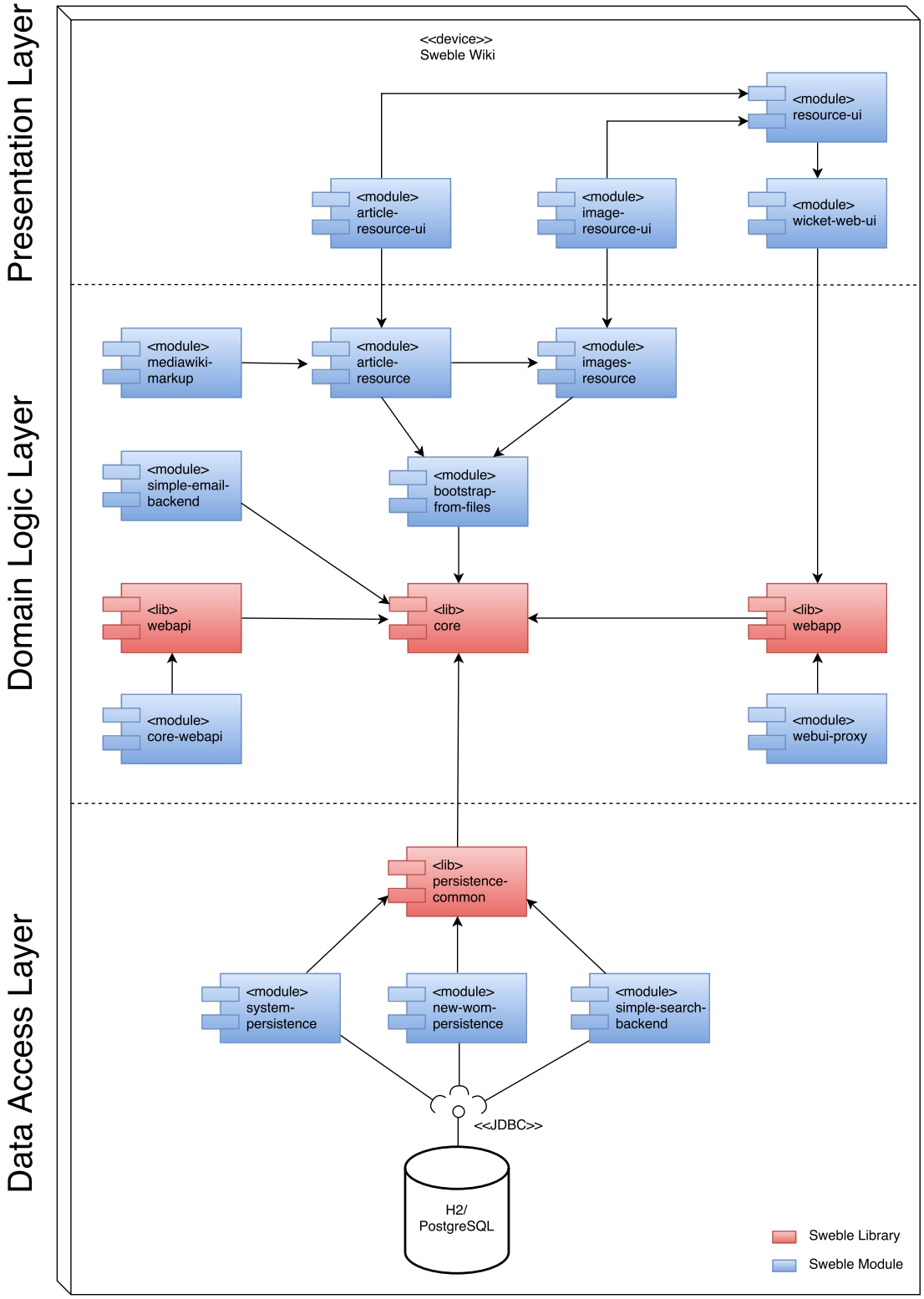

Abbildung 3.7: Abhängigkeiten der Maven Module des Sweble Wikis.

Zusätzlich wird zur Versionierung der Datenbank die Liquibase[6] Bibliothek eingesetzt. Liquibase ist ein datenbankunabhängiges Database Change Management Tool. Es können Änderungen am Datenbankschema verwaltet und durchgeführt werden. Alle Änderungen, die am Schema durchgeführt werden sollen, werden in Textdateien, sogenannte Changelogs, gespeichert. Die Changelogs definieren ein oder mehrere Changesets, welche eine geschlossene Datenbankänderungen festlegen. Die Changesets enthalten entweder datenbankunabhängige, liquibase-spezifische Syntax oder datenbankspezifische SQL-Blöcke. Diese Changelogs lassen sich bei Bedarf rückgängig machen. Sweble verwendet als weitere Library Java Object Oriented Querying[7] (jOOQ). jOOQ ist eine eine "database-mapping" Library, mit dem Ziel das Schreiben von SQL-Abfragen für den Programmierer einfacher zu gestalten.

Im Business Layer sind die Hauptfunktionalitäten des Wikis implementiert. Eine zentrale Library ist die core-Library, die von jedem Sweble Modul direkt oder indirekt importiert wird. In der Library befinden sich unter anderem Domänenklassen, Schnittstellenbeschreibungen, ein Domain Manager, um User, Project, Rollen, Permissions und Authentication-Token zu erzeugen und abzufragen, und ein Security Guard, der beispielsweise überprüft, ob ein User eine Ressource verändern oder abspeichern darf. Auch befinden sich in der Library eine Transformer- und ResourceTypeRegistry. Im Wiki sollen mehrere Ressourcen unterstützt werden. Dies kann ein "klassischer" Wiki-Artikel sein, aber auch Bilder, Videos, Audio, Meta-Daten, Software Snippets etc. Für jeden Ressourcentyp muss es eine eigene Transformation geben, die von und zu WOM transformiert. Sweble ist so aufgebaut, dass es Plugin-fähig ist, d.h. jeder kann durch eine Implementierung einer eigenen Transformation weitere Ressourcentypen hinzufügen. Um diese Transformationen im Wiki verfügbar zu machen, verwaltet die Klasse WomTransformerRegistry die verfügbaren Transformationen. Bei einer Transformation wird von einem Ressourcentyp zu einem anderen Ressourcentyp transformiert. In Sweble wird der Typ einer Ressource mit der Klasse ResourceType abgebildet. Ein ResourceType ist in Sweble als Interface modelliert und hat mehrere Implementierungen. Die Vererbungshierarchie eines ResourceType ist in Abbildung 3.8 zu sehen. Jeder ResourceType hat ein Attribut vom Typ *com.google.common.net.MediaType*. Die Klasse repräsentiert ein Internet Media Type[8] (auch bekannt als MIME-Type oder Content Type), welches ein Identifier für Datenformate ist. Zusätzlich besitzt jede ResourceType rekursiv einen Verweis auf ein anderes ResourceType in Form eines Attributs vom Typ ResourceType, welches das Eltern-ResourceType repräsentiert. ResourceTypes fügen somit eine Hierarchie zwischen Media Types ein und eine Baumstruktur von ResourceTypes wird erzeugt. Das Interface ResourceType deklariert Methoden,

[6] http://www.liquibase.org/
[7] http://www.jooq.org
[8] https://en.wikipedia.org/wiki/Media_type

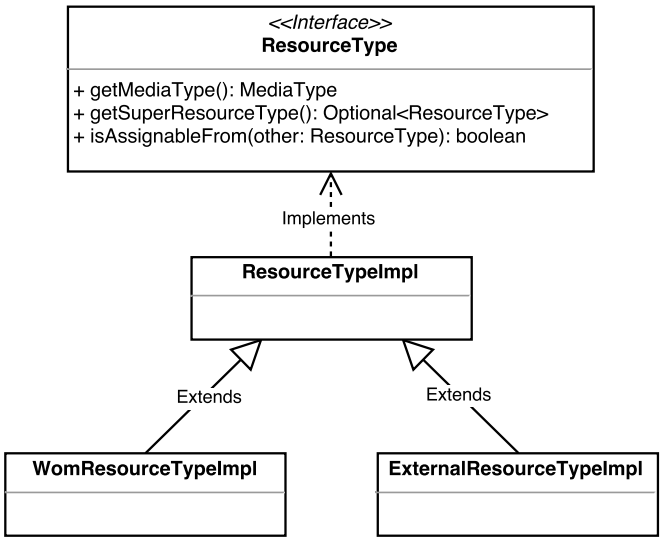

Abbildung 3.8: Klassendiagramm von ResourceType.

um den Media Type und das Eltern-ResourceType abzufragen. Das Interface wird von der Klasse ResourceTypeImpl implementiert. Zusätzlich erweitern die Klassen WomResourceTypeImpl und ExternalResourceType die Klasse ResourceTypeImpl. ResourceTypeImpl ist eine Domänenklasse für allgemeine Ressource Typen, WomResourceTypeImpl eine Klasse im Bezug auf Ressourcen, die als WOM dargestellt werden und ExternalResourceType kapselt externe Ressourcen. Jeder ResourceType der im Wiki unterstützt wird, wird an der ResourceTypeRegistry registriert. Die Hierarchie von ResourceTypes ist für die Transformer wichtig. Wenn beispielsweise ein Transformer angibt, dass er eine Ressource vom Typ "X" als Source transformieren kann, dann kann der Transformer auch Ressourcen transformieren, die ein Untertyp von "X" sind. Mithilfe der ResourceTypeRegistry kann der Transformer somit ResourceTypes für einen Media Type abrufen und anschließend mit der Methode *isAssignableFrom()*, welche die Hierarchie der ResourceTypes überprüft, validieren, ob er die Ressourcen transformieren kann.

Abbildung 3.9 zeigt einen Ausschnitt der Klassen in Form eines UML-Klassendiagramms, die bei einer Transformation benötigt werden. Die zentrale Klasse ist die WomTransformerRegistryImpl, welche alle verfügbaren Transformationen verwaltet und bietet Methoden an, um neue Transformationen zu registrieren oder bestehende Transformationen abzufragen. Wenn eine Transformation durchgeführt wird, wird immer von einer Source in ein Target transformiert. In Sweble existieren dafür die Interfaces WomTransformerSource und WomTransformerTarget. In Sweble gibt es mehrere Sources und Targets die transformiert werden können und folglich auch mehrere Implementierungen der beiden Interfaces, die hier aber nicht näher betrachtet werden. Jede Source und Target besitzt ein Attribut vom Typ Media Type, welches ein Media Type einer ResourceType reprä-

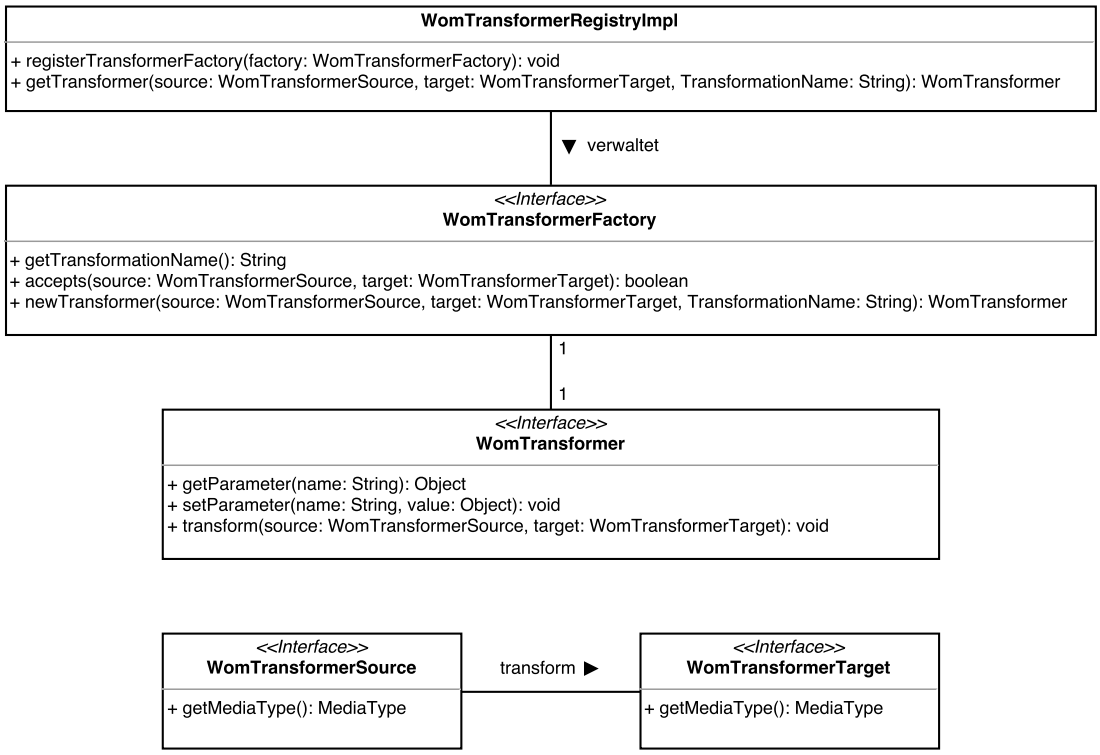

Abbildung 3.9: UML-Klassendiagramm der WomTransformer Domäne.

sentiert. Für jede Transformation im Wiki werden neue Implementierungen des Interfaces WomTransformerFactory und WomTransformer erstellt. Das Interface WomTransformer deklariert Methoden, um Parameter für die Transformation zu setzen und abzufragen. Mithilfe der Parameter können bestimmte Eigenschaften der Transformation verändert werden und das Target der Transformation leicht beeinflusst werden. Zwischen den Implementierungen einer WomTransformerFactory und eines WomTransformer besteht eine eins zu eins Beziehung. Wenn es im Wiki beispielsweise eine Transformation von WOM nach JSON geben soll, wird ein WomToJsonTransformer und eine WomToJsonTransformerFactory implementiert. Der WomToJsonTransformer beinhaltet die eigentliche Logik, wie eine WOM-Resource nach JSON transformiert wird. Die Factory legt den Namen der Transformation fest. Ebenfalls wird in der Factory eine Validierung (Implementierung der *accepts*-Methode) durchgeführt, die überprüft, ob die Transformation das übergebene Source und Target transformieren kann. Hier werden die Media Types von Source und Target mit den Media Types, welche der Transformation zugeordnet verglichen. Soll eine neue Transformation durchgeführt werden, wird eine Anfrage an die Methode *getTransformer()* der WomTransformer-Registry mit dem Name der Transformation, der Source und des Target gestellt. Die Registry ruft anschließend die *accept*-Methode der Factory auf und validiert Source und Target. Wenn die Registry ein Factory enthält, bei der die Validie-

rung erfolgreich ist, wird die Methode *newTransformer()* der Factory aufgerufen, welche eine Klasse vom Typ WomTransformer zurücklieft. Mit einem WomTransformer Objekt kann letztendlich die Transformation durchgeführt werden.

Im Business Layer sind folgende Module enthalten:

- *mediawiki-markup:* Bietet Unterstützung für die Wiki-Markup Sprache Wikitext an. Wikitext ist die Markup Sprache die von MediaWiki verwendet wird, um Beiträge zu formatieren. Das Modul bietet Transformationen von WikiText nach WOM und umgekehrt an.
- *article-resource:* Ergänzt das Wiki um mit "article" Ressourcen umzugehen. Dazu zählen Pre Render und Pre Save Transformationen, aber auch Transformationen von WOM nach HTML, um einen Artikel als HTML zu rendern.
- *image-resource:* Ergänzt das Wiki um mit "image" Ressourcen umzugehen. Dazu zählen Transformationen von Raster-/Vektorgraphiken zu WOM und umgekehrt.
- *simple-email-backend:* Bietet die Möglichkeit an, Emails an einen Email Server zu senden.
- *core-webapi:* Stellt eine REST API zur Verfügung, damit externen Applikationen mit dem Sweble Hub interagieren können.
- *webui-proxy:* Verantwortlich für das Starten des Wikis. Beim Deployment von Sweble wird die gesamte Applikation in ein Web Application Archive (WAR-File) gepackt. Anschließend wird der Java Servlet Container Jetty gestartet und das WAR-File auf diesen deployed.

Der Presentation Layer ist die Schnittstelle zum Benutzer und verantwortlich für Benutzereingaben und die Representation der Daten. Swebles User Interface (UI) basiert auf dem Web-Framework Apache Wicket[9]. Die Presentation Layer gliedert sich in folgende Module auf:

- *wicket-web-ui:* Stellt das Standard Design des Wikis zur Verfügung. Es enthält UI-Elemente wie das Menüleiste, Header- und Footerpanel.
- *resource-ui:* Stellt das "default" User Interface für Ressourcen bereit, die im Wiki unterstützt werden. Module, die eine bestimmte Art von Ressource hinzufügen möchten, müssen bestimmte UI-Komponenten registrieren. Zusätzlich bietet es noch HTML-Seiten für die Suche von Ressourcen, eine "Welcome Page" und Editoren zum Bearbeiten von Ressourcen an.
- *article-resource-ui:* UI Komponente zum *article-resource*-Modul.

[9] http://wicket.apache.org

- *image-resource-ui:* UI Komponente zum *image-resource*-Modul.

In Zukunft soll Wicket-Framework durch das Webframework Express[10] ersetzt werden, welches auf der Node.js Plattform aufbaut.

Sweble setzt als Dependency Injection (DI)-Framework Google Guice ein. DI ist ein Design Pattern mit dem die Abhängigkeiten eines Objektes zur Laufzeit injiziert werden können. Hat ein Objekt eine Abhängigkeit zu einem anderen Objekt, wird dies durch das DI-Framwork erzeugt und nicht von dem Objekt selber. In Guice werden mithilfe von Annotations Java Objekte mit ihren Abhängigkeiten konfigurieren. Es erlaubt Implementierungen an ein Interface zu binden, welche anschließend in Konstruktoren, Methoden und Attribute einer Klasse mithilfe einer *@Inject* Annotation injiziert werden können. Der Vorteil bei der Nutzung eines Dependency Injection Framework ist unter anderem, dass Abhängigkeiten zwischen Objekten einfacher verwaltet werden können Code einfacher zu lesen und zu testen ist.

3.3 Aufspalten des Sweble Monolithen

In Kapitel 3.1.2 wurde beschrieben, wie mit BCs ein Software System strukturiert werden kann. Eine Aufteilung von Sweble in BCs ist in Abbildung 3.10 zu sehen. In der Ressourcendomäne befinden sich die BCs Ressourcenmanagement und die Ressourcensuche. Das Ressourcenmanagement verwaltet Ressourcen und ist als Open Host Service (OHS) konzipiert, da viele weitere BCs auf diesen zugreifen müssen. Bei der Schnittstelle können Ressourcen abgefragt oder hinterlegt werden. Der BC Ressourcensuche ist abhängig vom Ressourcenmanagement, da er auf Ressourcen zugreifen muss, um diese in seinem Suchindex aufzunehmen. Eine weitere zentrale Komponente ist der BC Identitymanagement, welcher u.a. User und Projekte verwaltet. Hier findet eine Zuordnung statt, welche Ressourcen zu welchen Projekten zugeteilt sind. Auch verwaltet der BC existierende User im Wiki und ist für Authentifizierung und Autorisierung im Wiki verantwortlich. Zu diesem Zweck bietet das Identitymanagement ebenfalls eine Schnittstelle als OHS an. Schließlich gibt es noch die Transformationsdomäne. Jede Transformation stellt einen BC dar. Die Transformationen greifen auf den Resourcenmanagement-BC zu, um Ressourcen zu laden, um diese anschließend transformieren zu können. Die Transformationsdomäne ist erweiterbar. Es ist möglich, weitere Transformationen dem Wiki hinzuzufügen. Die Anzahl der Transformationen kann somit beliebig hoch sein.

Aus der Context Map mit ihren BCs ist eine Unterteilung des Sweble Wikis in Microservices hergeleitet worden (siehe Abb. 3.11). Ebenfalls ist das Prinzip

[10] expressjs.com

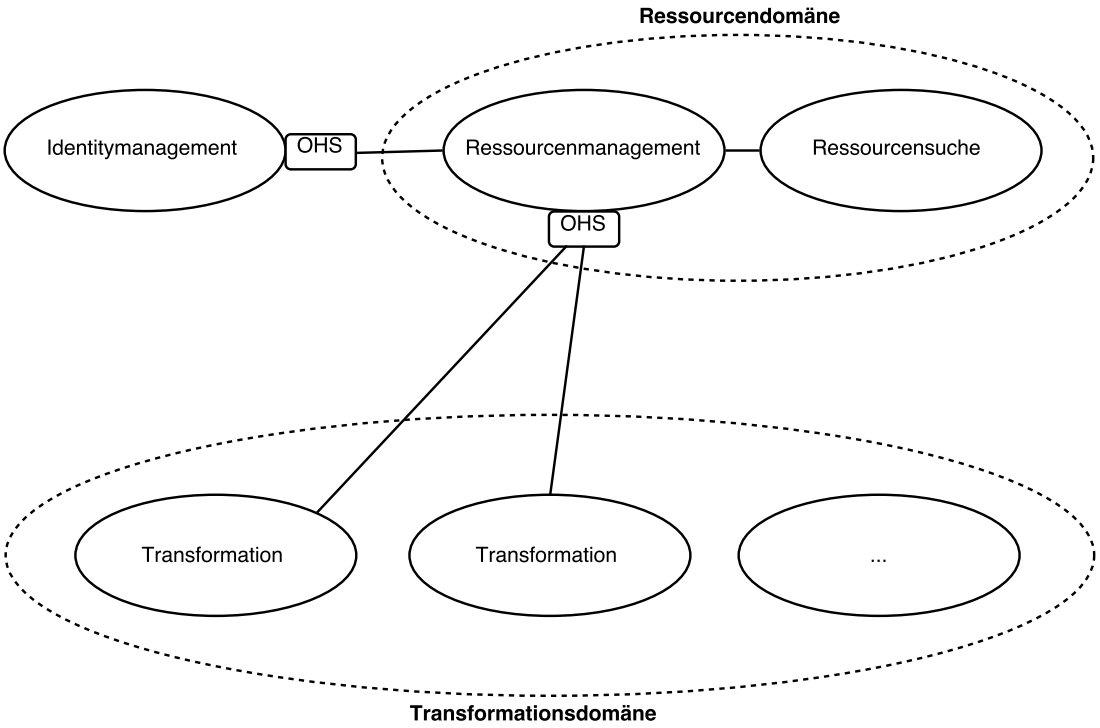

Abbildung 3.10: Context Map des Sweble Wikis.

der vertikalen Decomposition angewendet worden (siehe Kapitel 3.1.3), damit einzelne Microservices einfach skaliert werden können. Jeder Quader stellt einen Microservice dar. Aus dem BC Identitymanagement ist der Identity Microservice abgeleitet worden. In diesem Microservice befinden sich Implementierungen des Domain Managers und der Security Guards, der aus der Core Library herausgezogen wurde. Der Domain Manager besitzt Abhängigkeiten zu Implementierungen von Domänen Objekten wie User, Projekten, Rollen, Permissions und Authentication Tokens. Um diese persistent abzuspeichern, nutzt der Identity Microservice das system-persistence-Modul, welches die Objekte in einer Datenbank abspeichert. Der Identity Microservice besitzt somit seine eigene Datenbank. Ebenfalls befindet sich der Security Guard im Identity Microservice. Dieser wird beispielsweise vom anderen Microservice aufgerufen, um zu validieren, ob ein Benutzer eine Ressource erstellen bzw. editieren darf. Diese genannten Klassen sind in einen Microservice zusammengefasst worden, weil die Systemdomäne des Wikis ausmachen und stark miteinander gekoppelt sind.

Sweble ist so aufgebaut, dass es Plugin fähig ist. Es soll möglich sein, dass jeder eine weitere Ressourcenrepräsentation (und dadurch auch eine weitere Transformation) dem Wiki hinzufügen kann. Für jedes Plugin wird somit ein eigener Transformation-Microservice der Architektur hinzugefügt. Kein Plugin ist an andere Module gekoppelt, was den Vorteil hat, dass ein unabhängiges Deployment

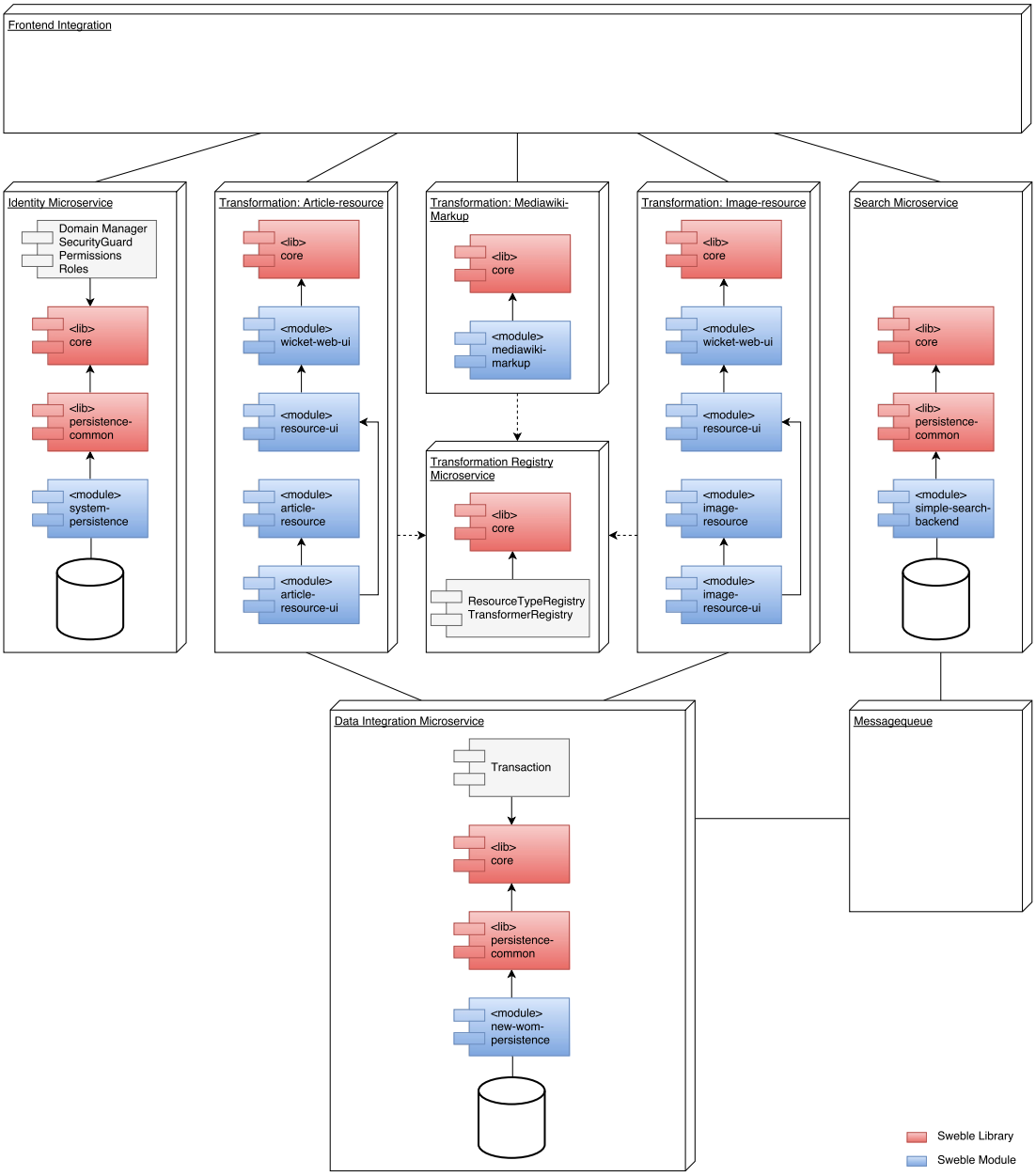

Abbildung 3.11: Unterteilung des Sweble Wikis in Microservices.

des Microservices ermöglicht wird. Bei Fehlern oder Bugs im Plugin, ist somit nur der Microservice des Plugins betroffen und nicht die ganze Applikation. Die Microservices Article-resource, Image-Resource und MediaWiki Markup können als solche Plugins angesehen werden und erweitern das Wiki um weitere Ressourcen. Sie sind die Transformation-Microservices in der Architektur. Article-resource und Image-resource erweitern das Wiki mit Artikel- bzw. Imageresource. Weitere Ressource können durch weitere Transformation-Microservices dem Wiki hinzugefügt werden. Jeder Transformation-Microservice stellt eine eigene Transformation der Ressource von und zu WOM bereit. Aus dem Monolithen sind Module aus der Domain Logic- und Presentation-Layer extrahiert worden. Jeder Transformation-Microservice enthält aus der Domain Logic-Layer das **-resource*-Module, welches die eigentliche Transformation enthält und aus der Presentation-Layer das Module **-resource-ui*, welches die zugehörige UI enthält. Die **-resource-ui*-Module sind abhängig vom *resource-ui* und *wicket-web-ui*-Modulen, die Klassen für das Default Layout von Sweble bereitstellen. Dadurch kann jeder Transformation-Microservice seine transformierte Ressource darstellen, beispielsweise in einer HTML-Page. Die Transformation-Microservices enthalten rechenintensiven Code. Sie bauen nicht aufeinander auf und können unabhängig voneinander betrieben werden. Dadurch können sie gut skaliert werden. Hier bietet sich wie in 3.1.3 vorgestellt, eine horizontale Skalierung an. Jeder Transformation-Microservice kann einzeln durch Hinzufügen oder Entfernen eines Microservices unabhängig skaliert werden. Alle Transformationen werden an einer zentralen Registry registriert. In der monolithischen Architektur war die Klasse WomTransformerRegistry aus der Core Library für die Registrierung von Transformationen verantwortlich (siehe Kapitel 3.2). Da die Transformationen in der Microservice Architektur verteilt und nicht mehr in einem Monolithen gebündelt sind, wird ein zusätzlicher Microservice benötigt, der die Aufgabe der Registrierung übernimmt. Dazu bietet der Transformation Microservice eine Registrierung per REST-Schnittstelle an. Jeder Transformation-Microservice kann sich somit per REST-Call an der Transformation Microservice anmelden. Die in Kapitel 3.2 vorgestellten Klassen für die Transformations Registry müssen angepasst werden. Beim Registrieren einer Transformation, wird der Registry mitgeteilt, unter welcher Location (IP-Adresse und Port) der Microservice erreichbar ist. Bei der Registry kann somit abgefragt werden, wo bestimmte Transformationen erreichbar sind und es wird die Location des angefragten Microservice zurückgeliefert. Genauer wird dies im Kapitel 4.3 betrachtet.

Zum Speichern oder Laden von Ressourcen greifen die Microservices auf den Data Integration Microservice zu. Dieser bietet eine Schnittstelle an, um neue Änderungsstände zu speichern. Die Schnittstelle ist dem Transaction Interface ähnlich, die im Monolithen verwendet wurde. Hier soll es möglich sein, Ressourcen aus der Datenbank abzufragen. Auch soll ein Änderungsstand der Commithistorie hinzugefügt werden können. Ein Änderungsstand besteht aus geänderte Ressourcen,

und/oder Ressourcen, die neu hinzugefügt oder gelöscht werden sollen. Auch soll es an der Schnittstelle möglich sein, zwei Commits zu einem zusammenzuführen.

Wenn in Monolithen ein neuer Änderungsstand abgespeichert wird (d.h. es werden neue Ressourcen im Module *new-wom-persistence* in der Datenbank gespeichert), werden dem *simple-search-backend*-Modul die modifizierten Ressourcen mitgeteilt, damit dieser den Suchindex aktualisieren kann. Dies wird mit einem *PostPersisteHook* realisiert. Der Hook wird dabei für jede Ressource aufgerufen, die hinzugefügt, geändert, oder gelöscht wird. Das *simple-search-backend*-Modul hat sein eigenes Datenbankschema und speichert dort Zusammenfassung von Ressourcen ab. Das *simple-search-backend*-Module wurde in der Microservice Architektur in einen eigenen Microservice ausgegliedert. Ein lokaler Aufruf des Hooks ist nicht mehr möglich. Anstatt des Hooks kann ein Messaging-System eingesetzt werden. Das Messaging-System kann dafür eine Publish/Subscribe Warteschlange anbieten. Der Data Integration Microservice kann als Producer fungieren, der pro neuer Ressource eine Nachricht an das Messaging-System sendet. Die Nachricht beinhaltet beispielsweise eine eindeutige Identifikation der Ressource. Die Identifikation von Ressourcen kann durch den Wiki Resource Identifier (WRI) vorgenommen werden. Durch das asynchrone Nachrichtenprotokoll wird der Data Integration Microservice dabei nicht blockiert und kann weitere Anfragen abarbeiten. Der Search Microservice meldet sich an der Warteschlange an und fungiert als Consumer des Messaging-System. Für jede Nachricht kann der Search Microservice anhand der WRI die Ressource laden und in seinem Suchindex hinzufügen. Das Messaging-System macht ebenfalls eine Skalierung des Search Microservice einfach möglich. Weitere Search Microservices können sich als Consumer an der Warteschlange anmelden.

Wolff (2015) empfiehlt, dass Microservices ihr eigenes UI mitbringen sollen. Dies hat den Hintergrund, dass alle Funktionalitäten in einem Microservice umgesetzt werden. Änderungen an der UI betreffen nur den einen Microservice. Die UIs müssen für das Gesamtsystem integriert werden. Tilkov (2014) gibt einen Überblick über die verschiedenen Möglichkeiten der UI-Integration. In der vorgestellten Architektur haben die Microservices Identity, Article-resource, Image-resource und der Search Microservice ihre eigenen UIs. Die Microservices können komplette HTML-Seiten oder HTML-Snippets ausliefern. Ein Front-End Server integriert die einzelnen HTML-Seiten der Microservices zu einer gemeinsamen UI. Damit alle HTML-Seiten einheitliche Oberflächen unterstützen, kann der Front-End Server als sogenannter Asset-Server fungieren. Er enthält CSS und JavaScript Bibliotheken für die HTML-Seiten der Microservices, damit diese weitgehend identisch aussehen. Auch können Design-Richtlinien definiert werden, um die Einheitlichkeit der HTML-Seiten zu gewährleisten.

Im Wiki existieren unterschiedliche Berechtigungen für unterschiedliche Benutzer. Ein normaler Benutzer hat weniger Rechte als der Admin des Wikis. Die

einzelnen Microservices müssen bei einer ankommenden Anfrage nun auch überprüfen, ob die Anfrage des Benutzers ausgeführt werden darf oder nicht. Hierbei spielen Authentifizierung und Autorisierung eine Rolle. Bei der Authentifizierung wird die Identität des Benutzers überprüft. Dies geschieht bei menschlichen Benutzern meist durch Eingabe eines Benutzernamen und eines Passworts. Bei der Autorisierung wird überprüft, ob ein bestimmter Benutzer eine bestimmte Aktion ausführen darf. Die Authentifizierung und Autorisierung wurden beide in der monolithischen Anwendung durchgeführt. Beide Vorgänge sind unabhängig und können prinzipiell getrennt voneinander durchgeführt werden. In einer Microservice Architektur ist es ohnehin nicht sinnvoll, die Authentifizierung und Autorisierung in einem Microservice durchzufurchen. Die Authentifizierung sollte ein zentrale Microservice übernehmen, da sonst jeder Microservice Benutzernamen und Passwörter speichern müsste. In der Sweble Microservice Architektur bietet sich der Identity Microservice für Authentifizierung an. Dafür kann das OAuth2-Protokoll[11] verwendet werden. OAuth ist ein offener Standard für Token-basierte Authentifizierung und Autorisierung. Der Identity Microservice fungiert dabei als Authorization Server. Ein Nutzer kann sich zunächst durch OAuth2-Ansatz beim Identity Microservice authentifizieren. Dazu wird bei einem neuen Request der Benutzername und das Passwort an den Identity Microservice gesendet. Dieser validiert, ob die Zugangsdaten korrekt sind und antwortet bei positiver Validierung mit einem Access Token. Dafür kann ein JSON Web Token (JWT)[12] verwendet werden. JWT ist ein offener Standard um Token zu generieren, welche bestimmte Informationen enthalten. Im Token werden Informationen, wie der Aussteller des Tokens und Gültigkeitszeitraum definiert. Auch enthält das Token welcher Zugriff auf welche Microservices erlaubt ist. Beispielsweise können Benutzer bestimmte Rollen zugeordnet werden. Jeder Microservice kann somit selber die Autorisierung anhand der Rollen, die im Token gespeichert sind, vornehmen. Würde der Identity Microservice die Autorisierung durchführen, müsste bei jeder Änderung der Zugriffsrechten der Identity Microservice angepasst werden. Das JWT kann zusätzlich verschlüsselt und eine digitale Unterschrift des Ausstellers enthalten, um zusätzliche Sicherheit zu gewährleisten. Das Access Token wird bei Aufruf eines anderen Microservices im HTTP-Header weitergereicht. Jeder Microservice entschlüsselt das Token, prüft die Unterschrift des Ausstellers und die Gültigkeit. Schließlich kann der Microservice anhand der Rolle überprüfen, ob der Benutzer die Aktion ausführen darf oder nicht.

In der Architektur gibt es drei Microservices, die eine Datenbank enthalten. Zu diesen Microservices zählt der der Identity, Search und Data Integration Microservice. Jeder dieser Datenbanken hat unterschiedliche Anforderungen. Beispielsweise sind beim Data Integration Microservice lediglich Create und Read Anfragen notwendig. Es werden entweder neue Commitstände gelesen oder neu gespeichert.

[11] https://oauth.net/2
[12] https://jwt.io

Eine Aktualisierung oder ein Löschen eines Commiteintrages ist nicht vorgesehen. Die Datenbank des Identity Microservice muss die Operationen Create, Read, Update, und Delete (CRUD) unterstützten. Wenn die Datenbank skaliert werden soll, muss diese stark konsistent sein, da beispielsweise bei Abfragen von Usern oder Authentication-Token widerspruchsfreie Daten zurückgeliefert werden müssen. Bei der Datenbank des Search Microservices wäre hingegen "eventual consistency" völlig ausreichend. Mit dem Microservice Ansatz kann gegebenenfalls jeder Microservice eine andere Datenbank Technologie einsetzen, da jeder Service unabhängig voneinander betrieben wird.

4 Implementierung

Aus dem Konzept wurde der Transformation Microservice und der Search Microservice implementiert und aus dem Monolithen ausgegliedert. Der Monolith wird in der aktuellen Umgebung weiterhin benötigt, da der Search Microservice von diesem eine Transformation zum Zusammenfassen von Ressourcen benötigt. Der Transformation Microservice wurde implementiert, weil er eine zentrale Komponente in der Architektur ist und ohne die Registry die Microservice Umgebung nicht betrieben werden kann. Die Microservices, die Ressourcen dem Wiki hinzufügen, wie beispielsweise der Article-resource oder Image-resource Microservice, wurden vorerst nicht implementiert. Dies liegt daran, dass diese Microservices eine eigene UI-anbieten sollen. Da aber auf die neue Express-UI umgestiegen wird, und ein Großteil der Klassen noch nicht implementiert sind, macht eine Implementierung dieser Microservices keinen Sinn. Ferner wurde der Search Microservice implementiert, da er die Transformation Registry benötigt und für diese ein Anwendungsbeispiel darstellt. Übrig bleiben noch Identity- und Data Integration Microservice. Diese wurden noch nicht implementiert und der Code befindet sich noch im Monolithen.

4.1 Dropwizard

Für die Entwicklung der Microservices wird das Dropwizard-Framework verwendet. Dropwizard baut auf Java-Standard-Bibliotheken wie Jetty (Servlet-Container), Jersey (JAX-RS-Referenz Implementierung) und Jackson (JSON-Bibliothek) auf. Dropwizard stellt Funktionalitäten wie Packaging, Deployment, Logging und Monitoring einer Anwendung zur Verfügung. Mit Dropwizard kann ein eigenständiger RESTful-Webservice eingerichtet und entwickelt werden und bietet sich deshalb für die Entwicklung von Microservices an. Am Ende des Build-Prozesses wird eine Fat-Java Archive (JAR)-Datei erzeugt. Die Anwendung ist in einer JAR-Datei gepackt und enthält die vollständige Infrastruktur mit den benötigten Libraries. Ein Microservice kann dadurch sehr leicht mit einem "java -jar"gestartet werden. Um die Dropwizard Framework zu benutzen, muss jeder

Microservice eine Maven Abhängigkeit zu dropwizard-core, welche die Basisfunktionalität von Dropwizard verfügbar macht, importieren. Jede Dropwizard Anwendung implementiert eine Unterklasse von `io.dropwizard.Configuration`, die umgebungsspezifische Parameter angibt. Diese Parameter werden in einer YAML-Konfigurationsdatei angegeben. Wird eine Dropwizard Anwendung gestartet, deserialisiert und validiert die Konfigurationsklasse die Parameter aus der YAML-Datei. Der Haupteinstiegspunkt einer Dropwizard Applikation ist die Klasse `io.dropwizard.Application`, die grundlegende Funktionen bereitstellt und bei der weitere Dropwizard Bundles registriert werden können. Ein Dropwizard Bundle ist eine Funktionalität, die der Applikation hinzugefügt werden kann, um das Verhalten der Applikation zu verändern.

4.2 Service Discovery

Als Service Discovery wurde Apache ZooKeeper ausgewählt. Um mit ZooKeeper zu kommunizieren, wurde in der Architektur die Java-Bibliothek Apache Curator[1] benutzt. Das Curator Framework ist eine API, die das Arbeiten mit Apache ZooKeeper vereinfacht. Es fügt Funktionen hinzu, die auf ZooKeeper aufbauen. Beispielsweise übernimmt es die Verwaltung von wiederholten Verbindungsaufbau aufgrund von Fehlern und verringert somit die Komplexität mit dem Zugriff, die mit ZooKeeper verbunden sind. Curator löst auf höherer Ebene häufig vorkommende Probleme, die in verteilten Systemen vorkommen.

Microservices sind in dieser Architektur selbst für die Registrierung an der Service Registry verantwortlich und folgen dem in Kapitel 3.1.1 vorgestellten Self-Registration Pattern. Dafür wird das *dropwizard-discovery* Bundle der Applikation hinzugefügt. In der Dropwizard Konfigrationsdatei (YAML-File) müssen weitere Parameter, wie der Name des Microservices und unter welcher Adresse die ZooKeeper Instanz erreichbar ist angegeben. Für jeden Microservice Namen (beispielsweise "article-resource-microservice") wird eine ZNode mit dem jeweiligen Namen des Microservice Typs registriert (siehe Abbildung 4.1). Für jeden Article-resource Microservice, der in der Microservice Architektur gestartet wird, wird eine neue Kind-ZNode mit dem ephemeral-Flag unter den Microservice Typ registriert. Zusätzlich werden Nutzdaten, wie Hostname und Port des Microservices als Nutzdaten zum Kind-ZNode serialisiert gespeichert. Wenn ein Microservice beendet wird, deregistriert sich der Microservice explizit beim ZooKeeper, um ihn aus der Liste der verfügbaren Microservices zu entfernen. Bei Ausfall eines Microservices wird dieser aufgrund der ephemeral Eigenschaft automatisch aus der ZooKeeper Liste genommen.

[1] http://curator.apache.org/curator-x-discovery/index.html

```
services (base path)
      |------ article-resource-microservice (name of microservice)
                  |--------- instance 1 id --> (serialized ServiceInstance)
                  |--------- instance 2 id --> (serialized ServiceInstance)
                  |--------- ...
      |------ image-resource-microservice (name of microservice)
                  |--------- instance 1 id --> (serialized ServiceInstance)
      |------ ...
```

Listing 4.1: Speicherhirarchie der Instanzen in ZooKeeper.

Möchte ein Microservice einen Service eines anderern Microservices in Anspruch nehmen, benötigt er dessen Standort. Diesen kann er bei ZooKeeper abfragen. Hier wird ebenfalls auf die Curator Library zurückgegriffen. Eine zentrale Abstraktionsklasse im Curator Framework ist die ServiceProviderBuilder Klasse, die den Discovery Prozess für einen bestimmten Servicenamen (Name des Microservices) und einer Auswahlstrategie kapselt (siehe Listing **??**). Es kann zwischen drei Strategien gewählt werden: Round-Robin (sequenzielles Rotieren durch die Liste aller gespeicherten Serviceinstanzen), Random (zufälliges Auswählen einer Instanz) und Sticky (Auswählen der gleichen Instanz). Es wurde die Round-Robin Strategie ausgewählt. Load Balancing wird dadurch von der Service Registry übernommen. Vom ServiceInstance-Objekt, welches das Curator Framework zurückgibt, kann anschließend die IP-Adresse und der Port ausgelesen werden. Vor jedem Request eines Clients an einen anderen Microservice, wird die Service Discovery Registry gefragt, unter welchem Standort der anzufragende Microservice ist.

```
ServiceInstance<InstanceMetadata> instance =
    serviceDiscovery.serviceProviderBuilder()
                    .serviceName(serviceName)
                    .providerStrategy(new RoundRobinStrategy<>())
                    .build()
                    .getInstance();
```

Listing 4.2: ZooKeeper Service Registry Client

4.3 Transformation Microservice

Der Transformation Microservice hat eine Abhängigkeit zur Sweble Core-Library, da er Zugriff auf die Domänenklassen benötigt. Die REST-Schnittstelle wurde mit dem Jersey-Framework umgesetzt. Die angebotenen Endpoints des Transformation Microservices sind in Tabelle 4.1 zu sehen. Der Transformation Microservice ist in die ResourceTypeRegistry und TransformationRegistryRestResource untergliedert. Bei der ResourceTypeRegistry können ResourceTypes abgefragt oder

PATH	HTTP-Verb	Body	Result
/resourcetypes	POST	ResourceType	-
/resourcetype	GET	-	ResourceType
/womresourcetype	GET	-	ResourceType
/externalresourcetype	GET	-	External-ResourceType
/transformation	PUT	WomTransformation	-
/transformation/location	PUT	WomTransformation-GetRequest	String

Tabelle 4.1: REST-Endpunkte des Transformation Microservice.

weitere ResourceTypes dem Wiki hinzugefügt werden. Listing 4.3 zeigt den Endpunkt der ResourceTypeRegistry, um einen neuen ResourceType hinzuzufügen.

```
1 @POST
2 @Path("/resourcetypes")
3 @Consumes(MediaType.APPLICATION_JSON)
4 public synchronized Response registerType(ResourceType type)
5 { ... }
```

Listing 4.3: REST Endpunkt zum Registrieren eines neuen ResourceTypes.

Um einen neuen ResourceType zu registrieren, ist der ResourceType serialisiert in JSON an den Endpoint *"/resourcetypes"* als POST Request zu senden. Als Parameter wird das Interface vom Typ ResourceType erwartet. In der monolithischen Anwendung wurde das Registrieren über einen lokalen Methodenaufruf mit einem Parameter vom Typ ResourceType realisiert. Hier hat die JVM überprüft, ob der übergebene Parameter eine Implementierung des Interfaces ResourceType ist. In der Microservice Architektur ist die ResourceTypeRegistry ein eigener Microservice und ein lokaler Methodenaufruf ist nicht mehr möglich. Stattdessen wird ein Aufruf über das Netzwerk ausgeführt. Dafür muss ein Object vom Typ ResourceType serialisiert werden und als Body in den HTTP-Request gepackt werden. Das Serialisieren übernimmt ein JAX-RS Client, der mit dem REST-Service kommuniziert. Die Deserialisierung findet in der REST-Schnittstelle des Microservices statt. Bei der Deserialisierung muss Jackson wissen, welche Implementierung des Interfaces ResourceType übergeben wurde, damit es das richtige Objekt erzeugen kann. Dafür wird das Interface ResourceType mit den Jackson Annotation *@JsonTypeInfo* und *@JsonSubTypes* annotiert (siehe Anhang Listing 7.1). Dadurch wird beim Serialisieren einer Implementierung von ResourceType ein zusätzliches Property hinzugefügt, welches angibt um welche Klasse es sich handelt. Bei der Deserialisierung weiß somit Jackson welches Objekt es erzeugen muss.
Um ResourceTypes aus der Registry abzufragen, wird ein Request an den End-

punkt in Listing 4.4 gesendet. Ein ResourceType wird durch einen Media Type identifiziert. Ein Media Type besteht aus einem "type" und "subtype". Diese Parameter werden mittels QueryParametern am Endpunkt angegeben. Wenn die Registry die angefragte ResourceType enthält, wird mit einem ResourceType serialisiert in JSON geantwortet.

```
1 @GET
2 @Path("/resourcetype")
3 @Produces(MediaType.APPLICATION_JSON)
4 public synchronized ResourceType getType(@QueryParam("type")
     String type, @QueryParam("subtype") String subtype)
5 { ... }
```

Listing 4.4: REST Endpunkt zum Abfragen eines ResourceTypes.

Zusätzlich bietet der Microservice Endpunkte an, um WomResourceTypes und ExternalResourceTypes abzufragen (siehe Tabelle 4.1).

Die Klassen, die für die Transformation zuständig waren, die in Kapitel 3.2 vorgestellt wurden, müssen an die verteilte Infrastruktur angepasst werden. Die Transformation Registry stellt ihre Services nun per REST-Schnittstelle zur Verfügung. Die Transformation Registry bietet zwei Endpunkte an (siehe Tabelle 4.1). Ein Endpunkt zum Registrieren von Transformationen und einer zum Abfragen der Location einer Transformation. Alle Microservices, die eine Transformation anbieten, registrieren sich an der Transformation Registry, indem sie ein Objekt vom Typ WomTransformation an den Endpunkt *"/transformation"* schicken. Die Klasse WomTransformation (siehe Anhang Listing 7.2) beinhaltet als Attribute den Namen der Transformation, den Namen des Microservices, der die Transformation anbietet, und die Media Types der Source und des Targets. Die Media Types geben dabei an, von welchem Media Type zu welchem Media Type die Transformation transformieren kann. Eine TransformerFactory, wie in der veralteten TransformerRegistry (Kapitel 3.2) ist nicht mehr nötig. Die Validierung anhand der Media Types, ob die Transformation durchgeführt werden kann, wurde aus der Factory in die Registry verschoben.
Will ein Microservice den Standort einer Transformation abfragen, sendet er an die Registry einen PUT-Request an den Endpunkt *"/transformation/location"*, im Body ein serialisiertes Objekt vom Typ WomTransformationGetRequest (siehe Anhang Listing 7.3). Die Klasse WomTransformationGetRequest beinhaltet den Namen der Transformation, und die Media Types von Source und Target. Die Media Types geben an, von welchem Media Type zu welchem Media Typ transformiert werden soll. Anhand der Media Types prüft die Registry ab, ob eine passende Transformation in der Transformation Registry gespeichert ist. Wenn eine passende Transformation gefunden wurde, wendet sich die Transformation Registry an die ZooKeeper Service Registry. Die Transformation Registry fragt ZooKeeper nach einer aktiven Instanz anhand des Namens des Microservices der

gespeicherten Transformation. ZooKeeper antwortet mit einer Service Instanz, welche die Location (IP-Adresse und Port) des Microservices enthält. Die Transformation Registry kann nun diese Location als String am Endpunkt *"/transformation/location"* zurückgegeben. Die Transformation Registry nutzt somit ZooKeeper um die Location von Microservices, die eine Transformation anbieten, herauszufinden. Dies hat folgenden Grund: In der Microservice Architektur kann es mehrere Microservices geben, die die gleiche Transformation anbieten. Zur gleichen Zeit können beispielsweise mehrere Instanzen des Article-resource Microservices gestartet sein. Alle Article-resource Microservices müssen sich an der Transformation Registry registrieren, da sie eine Transformation von WOM zu einem HTML-Artikel anbieten. Ohne die Verwendung von ZooKeeper müsste der Transformation Registry alle Instanzen verwalten, wie das Deregistrieren oder das Überprüfen mittels Heartbeats, ob Microservices noch laufen. Da ZooKeeper sich schon darum kümmert, verwendet die Transformation Registry die Einträge von ZooKeeper.

Jeder Microservice, der eine Transformation über eine REST-API im Wiki anbieten will, soll das Interface in Listing 4.5 implementieren. Dadurch wird eine einheitliche Schnittstellenbeschreibung geschaffen, um einen einheitlichen Zugriff für alle Transformationen im Wiki anzubieten.

```
1 @Path("/transform")
2 public interface WomTransformationRestResource
3 {
4         @PUT
5         @Consumes(MediaType.MULTIPART_FORM_DATA)
6         @Produces(MediaType.APPLICATION_JSON)
7         Response transform(MultiPart multiPart);
8 }
```

Listing 4.5: WomTransformationRestResource Interface.

Das Interface legt eine Methode *"transform"* fest, die als PUT-Request angefordert werden muss. Die Methode verarbeitet ein Media Type vom Typ Multipart. Generell werden bei HTTP-Requests im HTTP-Header im Content-Type-Feld festgelegt, als welcher MIME-Type der Body des Requests vorliegt. Bei der Transformation müssen allerdings zwei Media Types angegeben werden. Einmal die Parameter für die Transformation und einmal das zu transformierende Objekt. Diese beiden Parts können unterschiedliche MIME-Types besitzen. Um dies zu realisieren, wird ein Multipart im HTTP-Body versendet. Multipart ermöglicht es, im Body mehrere Parts zu setzen, die unterschiedliche MIME-Types besitzen. Beim Transformation-Endpunkt hat der erste BodyPart als Media Type APPLICATION_JSON_TYPE und im Body ein serialisiertes TransformParams-Object (siehe Anhang Listing 7.4). Die TransformParams Klasse hat zwei Attribute, um die Media Types von Source und Target anzugeben, und eine Map von String zu Objekten, um weitere Parameter für die Transformation zu setzen. In Monolithen

wurden diese Parameter in der Klasse WomTransformer gesetzt (siehe Abbildung 3.9). Im zweiten BodyPart wird die jeweilige Ressource gepackt, die transformiert werden soll. Als Media Type wird im BodyPart der Media Type der Ressource angegeben.
Als Ergebnis wird die transformierte Ressource im Body der Response zurückgeliefert.

4.4 Search Microservice

Der Search Microservice muss seinen Suchindex persistent abspeichern und benötigt dafür eine eigene Datenbank. Dropwizard bietet für den Datenbankzugriff ein zusätzliches Bundle an. In der Dropwizard Configuration Klasse werden Get- und Set-Methoden für eine DataSourceFactory hinzugefügt. Die DataSourceFactory wird aus den Konfigurationseinträgen aus der YAML-Konfigurationsdatei deserialisiert. In der YAML-Datei wird ein *database*-Eintrag mit Parameter, um sich an eine lokale PostgreSQL Datenbank zu verbinden, hinzugefügt. Es wird Username, Password, die URL zur Datenbankverbindung und der JDBC-Treiber für eine PostgreSQL Datenbank angegeben. Letztendlich wird in der Applikationsklasse das Bundle hinzugefügt.

Der Search Microservice hat weiterhin eine Maven Abhängigkeit zum Sweble Library persistence-common. In der Library befinden sich Hilfsklassen für den Datenbank Zugriff. Einer davon ist die Klasse *AbstractDataSourceGuiceModule*. Die Klasse ist eine abstrakte Basisklasse, welche mit Goolge Guice die Persistenzschnittstelle mithilfe einer DataSource bereitstellt, damit auf die DataSource in anderen Swebleklassen zugegriffen werden kann. Generell stellt eine DataSource in der Java Library eine logische Datenbankschnittstelle für eine Datenbankverbindung dar. Die Komplexität wird dadurch hinter der logischen Datenbankschnittstelle verborgen. Die Klasse *AbstractDataSourceGuiceModule* stellt grundlegende Dienstleistungen mithilfe der DataSource bereit. Beispielsweise wird der SQL-Dialekt festgestellt, ein Transaction-Manager für den Zugriff auf die DataSource initialisiert und jOOQ für die DataSource konfiguriert. Die genannten Objekte, welche mit der DataSource initialisiert werden müssen, werden mithilfe von Guice an den Injector übergeben und können in Klassen des Search Microservices injected und benutzt werden. Die Klasse *AbstractDataSourceGuiceModule* ist eine abstrakte Klasse und lässt sich somit nicht initialisieren. Die abstrakte Klasse benötigt eine DataSource um die genannten Initialisierungen vorzunehmen. Diese DataSource wird in der Unterklasse angegeben. Im Monolithen existieren bereits Unterklassen der Klasse *AbstractDataSourceGuiceModule*, die den Zugriff auf eine PostgreSQL und eine H2 Datenbank kapseln. Beim Search Microservice wird die DataSource mit dem oben vorgestellten Bundle von Dropwizard verwaltet. Es

wird eine Unterklasse (siehe Anhang Listing 7.5) der Klasse *AbstractDataSource-GuiceModule* erstellt, welche die DataSource des Dropwizard Bundles entgegen nimmt und diese beim Google Injector registriert.

PATH	HTTP-Verb	Body	Result
/search	GET	ResourceSearchResult	-
/search	PUT	-	ResourceProxy
/search	DELETE	-	-
/search/drop	DELETE	-	-

Tabelle 4.2: REST-Endpunkte des Search Microservices.

In Tabelle 4.2 sind die REST-Endpunkte des Search Microservices aufgelistet. Um eine neue Ressource den Suchindex hinzuzufügen, wird ein serialisiertes Objekt vom Typ ResourceProxy (siehe Anhang Listing 7.6) an den Endpunkt *"/search"* als PUT-Request gesendet. Die Klasse ResourceProxy hat als Attribute eine WRI, um die Ressource eindeutig zu identifizieren, ein Media Type, der den Media Type der Ressource angibt, die eigentliche Ressource als JSON, und zusätzliche Attribute (CommitAuthor, CommitId, CreationTime), welche der Ressource einen Commit zuordnen. Der Search Microservice muss die Ressource vorerst zu einer ResourceSummary transformieren, bevor er diese in seiner Datenbank abspeichert. Dafür stellt er eine Anfrage an die Transformation Registry, wer eine ResourceSummary Transformation anbietet. In der aktuellen Architektur ist diese Transformation noch im Monolithen enthalten. Der Monolith stellt einen Endpunkt, der ein Multipart als Parameter erwartet, wie in Listing 4.5 für eine Transformation von einer Ressource zu einer ResourceSummary bereit. Der Search Microservice stellt somit zuerst ein Anfrage an diesen Endpunkt des Monolithen. Im ersten Multipart werden die TransformParams gesetzt. In der TransformParams-Klasse werden Media Types für Source und Target angegeben. Als zusätzlichen Parameter für die Transformation (TransformParams) werden die WRI, die CommitId, CommitAuthor und die CreationTime gesetzt. Im zweiten Bodypart des Multipart wird die Ressource als JSON gesetzt. Mit diesen Parametern setzt der Search Microservice den Request an den Monolithen ab. Der Monolith transformiert die Ressource in eine ResourceSummary und sendet diese dem Search Microservice zurück. Schließlich speichert der Search Microservice die ResourceSummary in seine Datenbank ab.

Soll der Index des Search Microservice durchsucht werden, wird ein HTTP-GET Request mit den zu suchenden Stichwörtern als Query Parameter an den Endpunkt *"/search"* gestellt. Des Weiteren werden noch Endpunkte zum Löschen einer ResourceSummary mit Angabe der WRI und zum Löschen des kompletten Suchindex angeboten.

Im Monolithen wurde die Bibliothek Liquibase verwendet, um Änderungen am Datenbankschema vorzunehmen. Dropwizard bietet ein Bundle[2] an, um Liquibase einfach in die Applikation zu integrieren. Änderungen am Schema werden wie gewohnt in den Changelog- bzw. Changeset-Dateien definiert. Nachdem die Dropwizard Applikation gebuildet worden ist, können durch Dropwizard-Befehle Änderungen getestet, durchgeführt und bei Bedarf rückgängig gemacht werden.

[2]http://www.dropwizard.io/0.7.1/docs/manual/migrations.html

5 Evaluation

In diesem Kapitel wird die Microservice Architektur des Sweble Wikis evaluiert. Der Betrieb einer monolithischen Umgebung ist natürlich komfortabler als der Betrieb einer Microservice Umgebung. Der Monolith wurde mit Maven erstellt und mithilfe der Java Klasse StartWiki.java gestartet. Anschließend konnte auf das Wiki über einen Browser zugegriffen werden. Bei der Microservice Architektur müssen mehrere Artefakte deployed werden. Für die Microservice Architektur wurden zwei Microservices implementiert. Jeder Microservice wird mit Maven gebuildet und als JAR-File bereitgestellt. Ein Microservice kann manuell von der Kommandozeile mit Angabe des JAR-File gestartet werden. Da der Search Microservice noch eine Abhängigkeit zum Monolithen benötigt, muss dieser in der jetzigen Architektur ebenfalls gestartet werden. Zu der Architektur kommen nach dem ausgearbeiteten Konzept weitere Microservices hinzu. Ein manuelles Starten der Microservices ist dabei nicht mehr empfehlenswert und das Deployment sollte durch Skripte oder durch Werkzeuge wie Docker, Puppet oder Chef automatisiert werden.

Neben den Microservices, die die eigentliche Applikation darstellen, benötigt die Microservice Architektur aufgrund der verteilten Infrastruktur weitere Komponenten zur Koordinierung wie eine Service Discovery und ein Message-Queue. Zusätzlich können noch weitere Komponenten wie ein Load Balancer, ein Configuration Server und ein Edge Server der Umgebung hinzugefügt werden. Diese Komponenten müssen ausfallsicher betrieben werden. Die Microservices sind abhängig von diesen Komponenten und durch ein Ausfall dieser Komponenten kann die komplette Infrastruktur lahmgelegt werden, da die Microservices ihre Dienste nicht mehr zur Verfügung stellen können.

ZooKeeper kann ausfallsicher mit Replikaten auf mehreren Rechnern betrieben werden. Im Bezug auf das CAP-Theorem ist ZooKeeper ein CP-System. Bei einem Ausfall bedeutet dies, dass das System konsistent (und nicht verfügbar) ist. Dies ist ein notwendiger Kompromiss, da ZooKeeper in erster Linie als Koordinierungsdienst konzipiert wurde. ZooKeeper ist durch das Zab Protokoll darauf ausgelegt stark konsistent zu sein. Wenn eine Partition im System auftritt, wird ein Teil des Systems nicht mehr in der Lage sein, neue Registrierungen durch-

zuführen oder andere Services abzurufen. Alle Lese- und Schreibanfragen an die ZooKeeper-Instanzen, die nicht im Quorum enthalten sind, werden mit einem Fehler beantwortet. Für Service Discovery ist es manchmal besser Informationen über den Standort von Services zu erhalten, welche veraltet sind, als überhaupt keine Information zu bekommen (Kelley, 2014). Bei einem CA-System ist dies beispielsweise der Fall. Hier wird starke Konsistenz für Verfügbarkeit aufgeben. Allerdings nutzen große Unternehmen wie Pinterest ZooKeeper als Service Discovery (Prabhu, 2014). Auch Airbnb benutzt eine eigene Implementierung SmartStack, welche auf ZooKeeper beruht und für Service Registrierung und Discovery verantwortlich ist (Serebryany & Rhoads, 2013). Für die implementierte Microservice Architektur ist ZooKeeper im Grunde völlig ausreichend. Die Service Discovery mit ZooKeeper erfüllt die Anforderungen an eine Service Discovery in der jetzigen Architektur. Microservices können sich an der Service Registry registrieren, deregistrieren und mithilfe eines Heartbeat Mechanismus ihre Verfügbarkeit der Registry mitteilen. Ebenfalls übernimmt ZooKeeper das Load Balancing in der jetzigen Architektur und ein zusätzliche Load Balancer-Komponente wird eingespart, welche die Architektur weiter verkomplizieren würde. Sobald der Sweble-Monolith komplett in eine Microservice Architektur aufgeteilt und vollständig implementiert wurde und vorgesehen ist, Sweble im produktiven Einsatz betreiben zu wollen, kann abgewägt werden, ob ZooKeeper noch den Anforderungen entspricht. Zwischen ZooKeeper und dem Service Discovery Client ist darauf geachtet worden, dass wenig Kopplung besteht. Dementsprechend ist es einfach, auf eine andere Service Discovery umzusteigen.

Der Transformation und Search Microservice importieren die Core-Library, da sie Domänenobjekte aus der Library benötigen. Auch müssen weitere Microservices, die noch implementiert werden, die Core-Library importieren. Dadurch sind Microservices indirekt voneinander abhängig. Änderungen an der Core-Library führen dazu, dass alle Microservices von der Änderung betroffen sind. Es kann passieren, dass alle Microservices neu deployed werden müssen, wenn eine Änderung in der Core-Library durchgeführt wird. Deshalb müssen Änderungen an der Library koordiniert werden. Die Library sollte möglichst klein gehalten werden und nur nötigsten Code enthalten. Auch können weitere Teile der Core-Library in einen Microservice ausgelagert werden, damit die Library nur den nötigsten Code enthält.

Bei Endpunkten der Transformation Registry werden Interface-Implementierungen des Interfaces ResourceType als Parameter übergeben oder als Ergebnis zurückgeliefert. Wie in Kapitel 3.2 vorgestellt, hat eine ResourceType ein Verweis auf seine Eltern-ResourceType und es wird eine Baumstruktur von ResourceTypes erzeugt. Die ResourcType wird im Body des HTTP-Request mitgeschickt. Dabei wird eine ResourceType inklusive aller ihrer Elternknoten, serialisiert in JSON, mitgeschickt. In der aktuellen Architektur besitzt der ResourceType-Baum maximal ein Tiefe von n = 5 (dieser wird in der Klasse SwebleWikiAppGuiceModule

initialisiert). Somit hat das Eltern-ResourceType eine Kette aus maximal vier ResourceTypes. Die gesendete Datenmenge ist dabei noch akzeptabel. Es sollte darauf geachtet werden, dass die Hierarchie von ResourceTypes nicht in die Tiefe wächst, da sonst mehr Daten serialisiert werden müssen und große Datenmengen über das Netzwerk geschickt werden müssen.

Die Umstellung der Sweble-Architektur auf Microservices steht erst am Anfang und erst ein Teil der Microservices ist implementiert worden. Dies macht eine Evaluation beispielsweise im Bezug auf Performance oder auf (automatische) Skalierung der Umgebung schwierig. Abschließend kann gesagt werden, dass eine Umstellung eines Softwaresystems auf Microservices eine gewisse Zeit benötigt. Dies liegt daran, dass das Fundament der Sweble Architektur wesentlich geändert wurde. Es müssen architektonische Veränderungen durchgeführt werden, die bei der Implementierung des Monolithen natürlicherweise nicht beachtet worden sind, da zum damaligen Zeitpunkt andere Anforderungen an das System gestellt wurden.

6 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Konzept für eine Microservice Architektur für das Sweble Wiki und eine daraus resultierende Implementierung vorgestellt. Es wurde der Architekturstil Microservice und die jeweiligen Eigenschaften der daraus resultierenden verteilten Infrastruktur eingeführt. Die unterschiedlichen Möglichkeiten auf welcher Art sich Microservices an einer Service Registry anmelden wurde dargelegt. Anschließend wurde die aktuelle Architektur Swebles aufgezeigt. Dabei wurde unter anderem die Transformation Domäne genau betrachtet, da diese eine zentrale Komponente im Wiki darstellt. Die monolithische Architektur wurde in vertikale Teile geschnitten, aus denen Microservices abgeleitet wurden. Im Konzept ist ein Identity, Search, Data Integration und die Transformation-Microservices vorgestellt. Jede Transformation stellt dabei einen eigenen Microservice bereit, der unabhängig von anderen Komponenten skaliert werden kann. Die Transformation-Microservices registrieren sich beim Transformation Microservice, um ihre Transformation im Wiki verfügbar zu machen. Aus dem Konzept wurden zwei Microservices implementiert. Dies war der Transformation Microservice und der Search Microservice. Die Logik wurden aus dem Monolithen heraus gebrochen und in eigenständige Microservices gepackt. Der Monolith ist in der aktuellen Architektur weiterhin notwendig. Er fungiert als Legacy-System, und enthält die restliche Businesslogik des Wikis. Als Service Discovery wurde Apache ZooKeeper verwendet. Abschließend wurde die Microservice Umgebung evaluiert. Hier wurde gezeigt, dass durch die größere Anzahl an Komponenten weitere Komplexität der Architektur hinzugefügt wurde. Das Deployment und der Betrieb der einzelnen Komponenten wurde im Gegensatz zum Monolithen verkompliziert. Eine Evaluation der Skalierbarkeit in der Cloud konnte noch nicht durchgeführt werden, da erst ein Teil der Architektur implementiert wurde. Insgesamt kann aber gesagt werden, dass die Microservice Architektur eine sinnvolle Architekturentscheidung ist, um Sweble in Zukunft verteilt in die Cloud zu deployen, damit diese skalierbar ist.

7 Ausblick

Es wurde bis jetzt nur ein Teil der Microservice Umgebung implementiert. Als erstes können die restlichen Microservices implementiert werden, wie der Identity, Data Integration und die Transformation-Microservices. Da die Microservice-Typen unterschiedliche Umgebungen (wie beispielsweise Datenbankanbindung) benötigen, kann jeder Service in einer eigenen Virtuelle Maschine (VM) deployed werden. Dadurch werden die Microservices isoliert voneinander betrieben. Ein Nachteil von Virtualisierung ist, dass eine Menge von Ressourcen für die Isolation verschwendet werden (Balalaie u. a., 2015a). Alternativ kann ein Microservice auch in einem Docker[1]-Container deployed werden. Docker kann als "leichtgewichtige" Virtualisierung angesehen werden. Es können mehrere Container auf einem Betriebssystem betrieben werden, die alle den selben Betriebssystem-Kernel teilen. Dadurch hat Docker einen geringeren Overhead im Vergleich zur Virtualisierung.
Sobald Microservices in einer VM oder in einen Container deployed worden sind, kann die Microservice Umgebung in der Cloud betrieben werden. Hier können die einzelnen Microservices unabhängig skaliert werden. Es kann eine automatisierte Skalierung umgesetzt werden. Die Microservices können Metriken zu einer zentralen Stelle liefern, um die Belastung der Microservices zu überwachen. Dropwizard bietet eine umfangreiche Library[2] für das Sammeln von Metriken einer Applikation an. Es werden Kennzahlen in Form von Countern und Timern unterstützt. Ebenfalls kann eine Kennzahl in einem bestimmten Zeitrahmen definiert werden. Dadurch können beispielsweise die Anzahl der Transformationen, die in den letzten zehn Minuten durchgeführt wurden, ermittelt werden. Bei Überschreitung eines bestimmten Wertes können weitere Aktionen durchgeführt werden. Es bietet sich an, dass eine neue VM oder ein neuer Docker-Container gestartet wird, um eine zu hohe Last auszugleichen. Wenn die Last abnimmt und die Metriken einen bestimmten Wert unterschreiten, können VMs bzw. Docker-Container beendet werden, um Ressourcen zu sparen. Mit diesem Ansatz wird eine dynamische Microservice Umgebung umgesetzt, die sich automatisch an die anliegende Last anpasst.

[1] https://www.docker.com
[2] metrics.dropwizard.io

Anhang

Interface ResourceType

```
@JsonTypeInfo(
    use = JsonTypeInfo.Id.NAME,
    include = JsonTypeInfo.As.PROPERTY,
    property = "resType")
@JsonSubTypes({
    @Type(value = ResourceTypeImpl.class, name =
       "resourceTypeImpl"),
    @Type(value = ExternalResourceTypeImpl.class, name =
       "externalResourceTypeImpl"),
    @Type(value = WomResourceTypeImpl.class, name =
       "womResourceTypeImpl") })
public interface ResourceType
{
...
}
```

Listing 7.1: JSON-Annotationen für das Interface ResourceType.

WomTransformation

```
public class WomTransformation
{
        private String transformationName;

        private String microserviceType;

        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType transformerSource;

        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType transformerTarget;
...
}
```

Listing 7.2: Ausschnitt der Klasse WomTransformation.

WomTransformationGetRequest

```
public class WomTransformationGetRequest
{
        private String transformationName;

        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType transformerSource;

        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType transformerTarget;
...
}
```

Listing 7.3: Ausschnitt der Klasse WomTransformationGetRequest.

TransformParams

```
public class TransformParams
{
        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType sourceContentType;

        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType targetContentType;

        private Map<String, Object> properties = new HashMap<>();
```

Listing 7.4: Ausschnitt der Klasse TransformParams.

DropwizardDataSourceGuiceModule

```
public class DropwizardDataSourceGuiceModule
                extends
                        AbstractDataSourceGuiceModule
{
        private DataSource dataSource;

        public DropwizardDataSourceGuiceModule(DataSource
           dataSource)
        {
                this.dataSource = dataSource;
        }

        @Override
        protected void configure()
        {
        }

        @Provides
        @Singleton
        public DataSource provideDataSource()
        {
                return dataSource;
        }
}
```

Listing 7.5: Klasse DropwizardDataSourceGuiceModule.

ResourceProxy

```
public class ResourceProxy
{
        @JsonDeserialize(using = MediaTypeDeserializer.class)
        private MediaType mediaType;

        private String resourceAsJson;

        private WRI wri;

        private CommitId commiId;

        private CommitAuthor author;

        private DateTime creationTime;
...
}
```

Listing 7.6: Ausschnitt der Klasse ResourceProxy.

Lizenzen

Dropwizard:
Apache License 2.0

Apache Curator:
Apache License 2.0

Apache ZooKeeper:
Apache License 2.0

Literaturverzeichnis

Abbott, M. L. & Fisher, M. T. (2009). *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise* (1st). Addison-Wesley Professional.

Balalaie, A., Heydarnoori, A. & Jamshidi, P. (2015a). Microservices migration patterns. *http://ase.ce.sharif.edu/pubs/techreports/TR-SUT-CE-ASE-2015-01-Microservices.pdf*, (TR-SUT-CE-ASE-2015-01).

Balalaie, A., Heydarnoori, A. & Jamshidi, P. (2015b). Migrating to cloud-native architectures using microservices: an experience report. *CoRR*.

Calcado, P. (2015). How we ended up with microservices. Zugriff 21. September 2016, unter http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html

Dohrn, H. & Riehle, D. (2011a). Design and implementation of the sweble wikitext parser: unlocking the structured data of wikipedia. In *Proceedings of the 7th international symposium on wikis and open collaboration* (S. 72–81). WikiSym '11. Mountain View, California: ACM.

Dohrn, H. & Riehle, D. (2011b). Wom: an object model for wikitext. *University of Erlangen, Dept. of Computer Science, Technical Reports, CS-2011-05*.

Eaves, J. (2014). Micro services, what even are they? Zugriff 21. September 2016, unter http://techblog.realestate.com.au/micro-services-what-even-are-they

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Diss.). AAI9980887.

Fowler, M. & Lewis, J. (2014). Microservices a definition of this new architectural term. Zugriff 21. September 2016, unter http://martinfowler.com/articles/microservices.html

Gilbert, S. & Lynch, N. (2002 Juni). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, *33*(2), 51–59.

Goldberg, Y. (2014). Scaling gilt: from monolithic ruby application to distributed scala micro-services architecture. Zugriff 21. September 2016, unter https://www.infoq.com/presentations/scale-gilt

Ihde, S. & Parikh, K. (2015). From a monolith to microservices + rest: the evolution of linkedin's service architecture. Zugriff 21. September 2016, unter https://www.infoq.com/presentations/linkedin-microservices-urn

Jamshidi, P., Ahmad, A. & Pahl, C. (2013). Cloud migration research: a systematic review. *IEEE Transactions on Cloud Computing*, *1*(2), 142–157.

Kelley, P. (2014). Eureka! why you shouldn't use zookeeper for service discovery. Zugriff 21. September 2016, unter https://tech.knewton.com/blog/2014/12/eureka-shouldnt-use-zookeeper-service-discovery

Mauro, T. (2015). Adopting microservices at netflix: lessons for architectural design. Zugriff 21. September 2016, unter https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices

Newman, S. (2015). *Microservices: Konzeption und Design*. mitp Professional. MITP-Verlags GmbH & Co. KG.

Prabhu, R. (2014). Zookeeper resilience at pinterest. Zugriff 21. September 2016, unter https://engineering.pinterest.com/blog/zookeeper-resilience-pinterest

Richardson, C. (2014a). Pattern: api gateway. Zugriff 21. September 2016, unter http://microservices.io/patterns/apigateway.html

Richardson, C. (2014b). Pattern: monolithic architecture. Zugriff 21. September 2016, unter http://microservices.io/patterns/monolithic.html

Rotem-Gal-Oz, A. (2006). Fallacies of distributed computing explained. *URL http://www.rgoarchitects.com/Files/fallacies.pdf*, *20*.

Serebryany, I. & Rhoads, M. (2013). Smartstack: service discovery in the cloud. Zugriff 21. September 2016, unter http://nerds.airbnb.com/smartstack-service-discovery-cloud

Tilkov, S. (2014). Web-based frontend integration. Zugriff 21. September 2016, unter https://www.innoq.com/blog/st/2014/11/web-based-frontend-integration/

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing colombian conference (10ccc), 2015 10th* (S. 583–590).

Wolff, E. (2015). *Microservices: Grundlagen flexibler Softwarearchitekturen*. Dpunkt.Verlag GmbH.