FH | JOANNEUM
University of Applied Sciences

Microservices mit Spring Boot

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science in Engineering (BSc)
eingereicht am
Fachhochschul-Studiengang **Software Design**

FH JOANNEUM (University of Applied Sciences), Kapfenberg

Betreuer: FH-Prof. DI Dr. Egon Teiniker

eingereicht von: Benjamin Krenn
Personenkennzahl: 1410418084

September 2017

Zusammenfassung

Die weltweite Vernetzung und die Anzahl der Internet-Nutzer und Nutzerinnen steigt kontinuierlich an. Unternehmen stehen vor der Herausforderung ihre Applikationen an die wachsende Benutzer und Benutzerinnen Anzahl anzupassen. Neue Technologien wie Cloud Computing ermöglichen es Unternehmen ihre Applikationen dynamisch, effizient und auf Abruf zu skalieren. Diese neuen Möglichkeiten gehen jedoch einher mit neuen Herausforderungen an die Architektur von Enterprise-Anwendungen, denen monolithische Anwendungen nur schwer gerecht werden können. Um diese Herausforderungen zu bewältigen setzen große Internet-Unternehmen wie Netflix, Amazon und The Guardian, bereits seit einigen Jahren auf das Microservice-Architektur-Muster. In dieser Arbeit wird das Microservice-Architektur-Muster anhand einer Prototyp-Implementierung vorgestellt und auf etwaige Unterschiede zu einer monolithischen Architektur untersucht. Im ersten Teil der Arbeit werden die untersuchten Architektur-Muster und Technologien zur Umsetzung einer Microservice-Architektur vorgestellt. Der zweite Teil behandelt die, zur Entwicklung der Prototyp-Anwendung, eingesetzten Technologien im Detail und zeigt deren Einsatz anhand der Prototyp-Anwendung.

Kapfenberg, September 2017

Akademischer Betreuer:
FH-Prof. DI Dr. Egon Teiniker

Benjamin Krenn

Abstract

The world-wide interconnectedness and the number of Internet users is steadily growing. This issues enterprises to adapt their applications to handle this growing user base. New technologies such as cloud-computing are allowing enterprise to scale their applications on-demand in a more efficient and dynamic way. However, this new opportunities walk hand in hand with new challenges for enterprise software architecture which might not be handled efficiently by monolithic applications. Large Internet companies like Netflix, Amazon and The Guardian have been using and advocating the microservice architecture pattern successfully during the last years. The first part of this thesis briefly describes the evaluated architectural patterns and microservice technologies used to realize the prototype application. In the subsequent practical part concrete solution approaches are presented in detail which depict the foundation for the prototype application.

Kapfenberg, September 2017

Academic adviser:

FH-Prof. DI Dr. Egon Teiniker

Benjamin Krenn

Eidesstattliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Bachelorarbeit selbstständig angefertigt und die mit ihr verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für gutes wissenschaftliches Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die vorliegende Originalarbeit ist in dieser Form zur Erreichung eines akademischen Grades noch keiner anderen Hochschule vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Kapfenberg, September 2017

Benjamin Krenn

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Im Jahr 2015 war die Streaming-Plattform Netflix, mit damals 40 Millionen Nordamerikanischer Benutzer und Benutzerinnen, für 37% des Nordamerikanischen Internetverkehrs verantwortlich. Benutzerzahlen dieser Größenordnung stellen Inhaltsanbieter wie Netflix vor große technische Herausforderungen. Es werden hoch-skalierbare, flexible und widerstandsfähige Software-Systeme benötigt, um Benutzerzahlen dieser Größenordnung bedienen zu können. Aus dieser Notwendigkeit heraus, haben einige große Unternehmen wie Amazon, Netflix und The Guardian die Microservice-Architektur in ihren Applikationen adaptiert. Umfangreiche und komplexe Applikationen werden in eine Menge kleiner und lose-gekoppelter Dienste, die über leichtgewichtige Mechanismen kommunizieren, aufgeteilt. Die draus entstehenden Microservices können einzelnen entwickelt, getestet, ausgeliefert, aktualisiert und skaliert werden. So wird diesen Unternehmen eine agilere Vorgehensweise, höhere Erreichbarkeit und einer flexiblen On-demand Skalierung der Rechenkraft ermöglicht. Diese Vorteile gehen mit einer erhöhten Komplexität durch die Nutzung eines verteilten Systems einher und viele Design-Prinzipien und Möglichkeiten einer Microservice-Architektur sind noch unklar. Architekten und Architektinnen stehen vor der Herausforderung die entstehende Komplexität durch den Einsatz unterstützender Technologien zu bändigen. Ziel dieser Arbeit ist es die wesentlichen Aspekte einer Microservice-Architektur und deren Umsetzung mit Hilfe des Java Enterprise-Frameworks Spring zu untersuchen. Es wird untersucht wie eine Microservice-Architektur mit dem Spring-Framework umgesetzt werden kann, welche unterstützenden Technologien genutzt werden können, um die Komplexität zu verringern und wie sich Microservice-Architekturen in den untersuchten Punkten von monolithischen Architekturen unterscheiden.

Kapitel 2

Stand der Technik

In diesem Kapitel werden sowohl das monolithische Architekturmuster als auch das Microservice Architekturmuster vorgestellt. Außerdem wird auf die zur Erstellung des Microservice-Prototypen relevanten Technologien, Techniken und Konzepte eingegangen.

2.1 Monolithische Architektur

Ein Großteil der heute eingesetzten Java Enterprise-Software-Applikationen wurden ursprünglich als große, monolithische und komplexe Applikationen entworfen sowie entwickelt. Das Ergebnis des Erstellungsprozess (engl. Build process) ist eine ausführbare Applikation oder ein Deployment-Artefakt, das auf einen Applikationsserver eingespielt wird. Am Entwicklungsprozess einer monolithischen Applikation sind meist eine große Anzahl von Entwicklerinnen und Entwicklern, die an einer gemeinsamen Quellcode-Basis arbeiten, beteiligt. Eine große Anzahl der am Entwicklungsprozess beteiligten Personen bedeutet dass Änderungen am Quellcode achtsam orchestriert und kommuniziert werden müssen. Monolithische Applikationen besitzen oft eine sehr große Anzahl von Regressions-, Integrations- und Unit-Tests, welche bereits bei kleinen Änderungen am Quellcode im Zuge des Build-Prozesses ausgeführt werden. Dies resultiert in langen Feedback-Zyklen und einem langwierigen Build-Prozess, der oft Stunden in Anspruch nehmen kann (vgl. Bakshi 2017, S. 1). Architektonisch sind monolithische Applikationen meist in Schichten unterteilt und besitzen eine große Anzahl an Services, die über unterschiedlichste Schnittstellen kommunizieren können. Die Skalierung einzelner Services ist nicht möglich und es kann nur in eine Dimension skaliert werden: Es müssen weitere Kopien der gesamten Applikation hochgefahren

werden. Zudem stellen monolithische Deployment-Einheiten eine einzelne Schwachstelle (engl. Single point of failure) dar: Fällt die Applikation aus fallen gleichzeitig auch alle Services aus. Obendrein erfordern Änderungen an einzelnen Services eine neue Auslieferung der gesamten Applikation (vgl. Villamizar u. a. 2015, S. 583ff.).

2.1.1 Service-orientierte Architektur

Die ersten Bestrebungen individuelle Business-Funktionalitäten und Prozesse einer Applikation in lose gekoppelte Services zu kapseln gibt es bereits seit dem Beginn der Jahrtausendwende. Ein Service innerhalb der Service-oriented Architecture (SOA) repräsentiert eine Menge von Business-Funktionalitäten. Zwischen den Services und den Konsumenten steht der Enterprise Service Bus (ESB). Der ESB stellt eine Plattform, welche als Mediator zwischen Service-Anbietern und Konsumenten auftritt, dar. Er soll eine hohe Zuverlässigkeit und Flexibilität garantieren. Typischerweise übernimmt der ESB Infrastruktur-Aufgaben wie Messaging, Orchestrierung der Services und Routing. Das fundamentale Prinzip der SOA sei laut Xiao allerdings die lose Koppelung und minimale Abhängigkeiten zwischen den Services (vgl. Zhongxiang, Inji und Xinjian 2016, S.60ff). Die Implementierung von SOA Strategien wäre jedoch langwierig, teuer und sehr komplex. ESB Produkte seien zwar dem Workload von tausenden Benutzern und Benutzerinnen gewachsen, ließen sich laut Villamizar et al. nicht auf eine, für Internet-Applikationen ausreichende Größe skalieren. Sie würden einen Single-Point of Failure und einen Flaschenhals darstellen und seien nicht für die Cloud entworfen worden (vgl. Villamizar u. a. 2015).

2.2 Microservice Architektur und relevante Technologien

Microservices sind kleine, autarke Systeme, die durch Kommunikation und Interaktion ein komplexes Gesamtsystem darstellen, um eine wohl-definierte Aufgabe zu erfüllen. Im Wesentlichen stellen Microservices ein Modularisierungskonzept dar, welches große Software-Systeme in kleine Einheiten aufteilen soll (vgl. Wolff 2017, S.3). Dies ist jedoch kein neuer Ansatz (siehe Kapitel 2.1.1). Wolff nennt als wichtigsten Unterschied zwischen SOA und Microservices, dass SOA das gesamte Unternehmen betrachte, während Microservices ein Architekturmuster für ein einzelnes System sei. Microservices seien flexibler als SOA (vgl. Wolff 2017, S. 89ff.). Im Kern jedoch sei-

en, so Wolff, Microservices und SOA zwei unterschiedliche Herangehensweisen mit einem gemeinsamen Ziel. Als Hauptunterschiede sind die Größe, Eigenständigkeit und die Entwicklungszyklen zu nennen (vgl. Zhongxiang, Inji und Xinjian 2016). Allerdings würden SOA Systeme oftmals als monolithische Anwendung ausgeliefert (vgl. Wolff 2017, S. 94). Im Gegensatz zu monolithischen Anwendungen können Microservices einzeln und unabhängig voneinander ausgeliefert (engl. deployed) und skaliert werden. Microservices ermöglichen eine einfache horizontale Skalierung, da nicht das System als ganzes sondern einzelne Services skaliert werden können und ein hohes Modularisierungsniveau mit undurchlässigen Grenzen erreicht wird, die schwer überschritten werden können (vgl. C. Richardson 2017, S. 9ff.). Bondi zufolge sei die Skalierbarkeit und Anpassungsfähigkeit eines Systems an wachsende Nachfrage entscheidend für den langfristigen Erfolg eines Systems. Ein System, welches nicht effizient skaliert werden kann, erhöhe die Arbeitskosten und verringere die Servicequalität (vgl. Bondi 2000, S.195).

Carnell definiert vier wesentlichen Charakteristika Microservice-Architektur:

Lose Koppelung Zwischen den einzelnen Microservices soll lediglich eine lose Koppelung bestehen. Interagieren Microservices über eine sprach-neutrale Schnittstelle wie beispielsweise eine wohl-definierte Representational State Transfer (REST)ful Hypertext Transfer Protocol (HTTP) API, sind die Services nicht von den Implementierungsdetails anderer Services abhängig und die Koppelung zwischen den Services kann minimiert werden (vgl. Spichale 2017, S. 9). Ein Microservice sollte eine möglichst geringe Abhängigkeit zu anderen Microservices haben, da dieser Umstand es erlaubt einzelne Microservices zu ändern ohne dass andere Services davon betroffen sind (vgl. Wolff 2017, S.101ff.).

Eingeschränktheit Jeder Microservice besitzt eine klar definierte Aufgabe, welche möglichst effizient erfüllt werden soll. Außerdem sollte innerhalb eines Services eine sehr hohe Kohäsion herrschen. Eine hohe Kohäsion stellt sicher, dass die einzelnen Komponenten eines Microservices wirklich homogen sind.

Abstraktion Ein Microservice nutzt seine eigenen Datenstrukturen und Datenquellen. Daten welche in den internen Bereich des Bounded Context fallen können nur vom Service selbst geändert werden.

Unabhängigkeit Die Microservices können unabhängig voneinander kompiliert, modifiziert und ausgeliefert werden. Die Microservices nutzen keinen geteilten Code um gemeinsame Abhängigkeiten und die Koppelung zwischen den Services möglichst gering zu halten (vgl. Newman 2015, S. 30).

Die Microservice Architektur verspricht schnelles Feedback, kurze Release-Zyklen und schnellere Auslieferungszeiten.

2.2.1 Spring Boot

Spring Boot ist eine, von Pivotal Software entwickelte, Neuauslegung des quelloffenen Enterprise Java-Frameworks Spring. Während weiterhin auf grundlegende Kernelemente des Spring-Frameworks gesetzt wird, sei Spring Boot, laut Carnell als ein Java-basierendes, REST-Orientiertes Microservice-Framework anzusehen (vgl. Carnell 2017, S. 6). Laut Walls sei das Hauptmerkmal von Spring Boot die vereinfachte Nutzung des eigentlichen Spring-Frameworks durch die Einführung automatischer Konfiguration der Komponenten, um den Anforderungen komplexer Geschäftsanwendungen besser gerecht werden zu können (vgl. Walls 2016, S.12). Spring ermöglicht eine Entkoppelung der einzelnen Anwendungskomponenten und vereinfacht die Entwicklung von komplexen Anwendungen und deren Geschäftslogiken. Walls nennt vier Schlüsselstrategien des Spring-Frameworks: Die Leichtgewichtigkeit von Spring, die lose Koppelung, deklarative Programmierung und die Reduzierung von redundanter Standardcodierung (vgl. Walls 2014 S. 30).

Spring lässt sich grob in zwei wesentliche Hauptmerkmale, welche gleichzeitig die Programmierparadigmen für dieses Framework festlegen, untergliedern: Zum einen die “Dependency Injection”, welche die Abhängigkeiten eines Objekts zur Laufzeit festlegt und zum anderen die aspektorientierte Programmierung, welche die logischen Aspekte einer Anwendung von der eigentlichen Geschäftslogik trennt (vgl. Walls 2016, S.26). Im Zentrum jeder Spring-Anwendung steht der Spring Container oder auch Applikationskontext genannt. Der Applikationskontext instanziiert die einzelnen Objekte und legt die Abhängigkeiten zur Laufzeit fest. Im Gegensatz dazu werden bei konventionellen Java Applikationen ohne Dependency Injection die Abhängigkeiten zwischen den einzelnen Objekten zur Kompilierungszeit festgelegt und können demnach nicht mehr zur Laufzeit geändert werden.

Spring kann als Intermediär zwischen den verschiedenen Java Klassen einer Applikation angesehen werden (vgl. Carnell 2017, S. 5). Jedes von Spring verwaltete Objekt, auch Spring-Bean genannt, befindet sich während seines gesamten Lebenszyklus in

diesem Container. Spring bietet die Möglichkeit in diesen Lebenszyklus der Spring-Beans programmatisch einzugreifen.

Der Applikationskontext kann wahlweise über eine XML- oder eine Java-Annotation-Konfiguration aufgebaut werden. Spring Boot vereinfacht und abstrahiert diese, oftmals komplexe, Konfiguration von Spring und baut auf dem Prinzip "Konvention vor Konfiguration" auf. Spring Boot erkennt zur Laufzeit welche Bibliotheken sich im Klassenpfad der Anwendung befinden und konfiguriert sich auf Basis dieser Information selbstständig (vgl. **Pivotal2017BootAutoConfig**). Spring stellt sogenannte "Starter-Dependencies" zur Verfügung, dessen Zielsetzung die oftmals komplexe Verwaltung der Projektabhängigkeiten zu vereinfachen. Im Wesentlichen sind die "Starter-Dependencies" herkömmliche Maven- oder Gradle-Abhängigkeiten, welche die transitive Auflösung von Abhängigkeiten ausnutzt um häufig genutzte Abhängigkeiten zur Verfügung zu stellen. So genügt es die "Starter-Dependency" `org.springframework.boot:spring-boot-starter-web` einzubinden, um eine voll funktionsfähige Spring Webanwendung zu erstellen. Diese Abhängigkeit verfügt über notwendigen transitive Abhängigkeiten wie beispielsweise dem eingebetteten Tomcat Server (vgl. Walls 2016, S. 5). Ferner stellt Spring Boot ein Gradle-Plugin, dass die Java Archive (JAR) Dateien aller Abhängigkeiten in eine einzelne, ausführbare JAR-Datei sammelt, zur Verfügung. Vor der Veröffentlichung von Spring Boot war Spring laut Walls zwar leichtgewichtig im Bezug auf Komponenten-Quellcode, allerdings schwergewichtig in der Konfiguration (vgl. Walls 2016, S.19).

Ein weiterer wichtiger Aspekt von Spring Boot ist die Möglichkeit, eine Spring Boot-Anwendung direkt auf einem eingebetteten HTTP-Server wie Tomcat, Jetty oder Undertow einzusetzen, was sich vor allem im Bereich von Microservices als wichtiger Punkt darstellt. Es muss kein Web Application Resource (WAR) oder Enterprise Application Archive (EAR) auf einem Applikationsserver eingespielt werden, da alle zur Ausführung notwendigen Komponenten bereits im JAR enthalten sind. Diese JAR-Datei kann mittels der Java Runtime Environment (JRE) ausgeführt werden. Wolff beschreibt diesen Prozess als simpler und leichtgewichtiger als den Einsatz von Applikationsservern wie Wildfly oder Websphere (vgl. Wolff 2017, S. 307). Zusätzlich profitiert Spring Boot vom umfangreichen Spring-Ökosystem, welches mehr als 20 Projekten umfasst. Diese Projekte können nahtlos in eine Spring Boot-Anwendung integriert und verwendet werden (vgl. Webb u. a. 2017). Dazu zählen beispielweise Spring Data, welches den Umgang mit Datenbanken durch Abstraktion und Objekt-Mapping-Funktionalität deutlich vereinfacht und Spring Cloud das eigens Funktionalität für Microservices und verteile Systeme bereitstellt (vgl. Pivotal Inc. 2017c). Spring Boot biete somit einige wichtige, für Microservices relevante Funktionalität

und ermögliche eine einfache und rasche Auslieferung von Anwendungen (vgl. Walls 2016, S.21ff.).

2.2.2 Spring Cloud Config

Spring erlaubt es spezifische Konfigurationsdetails mittels Properties-Dateien vom eigentlichen Quellcode zu trennen. Diese Dateien werden automatisch durch Spring vom Klassenpfad (engl. Classpath) in den Applikationskontext eingespielt und es kann auf die Konfigurationsdetails mittels Platzhalter im Quellcode zugegriffen werden. Müssen Konfigurationsdetails angepasst werden, erfordert dies jedoch gleichzeitig auch ein erneutes Packaging der Anwendung. Dieser Ansatz würde laut Carnell zwar bei monolitischen oder einer kleiner Anzahl von Microservices funktionieren, bei cloudbasierenden Anwendungen, die hunderte von Microservices umfassen können, würde für das Konfigurationsmanagement jedoch ein differenzierter Ansatz benötigt werden (vgl. Carnell 2017, S. 65). Spring stellt zu diesem Zweck den Spring Cloud.Konfigurationsserver zur Verfügung. Dieser erlaubt eine komplette Trennung der Konfiguration und des Quellcodes. Es wird also keine Konfigurationsdatei im Klassenpfad benötigt. Der Spring Cloud-Konfigurationsserver ermögliche es die einzelnen Services generisch zu halten und die Konfigurationen zentral zu verwalten (vgl. Pivotal Inc. 2017b). Wird ein Microservice gestartet, versucht er in der „bootstrapping“-Phase zunächst den Konfigurationsserver zu kontaktieren um seine Konfigurationsdetails über eine HTTP REST Application Programming Interface (API) zu erhalten. Gelingt dies nicht, greift der Microservice auf die lokale Konfigurationsdatei zurück. Die Konfigurationsdateien können wahlweise in einem Repository eines Versionsverwaltungssystem, wie beispielsweise Git, hinterlegt sein oder auch wie bei jeder anderen Spring-Applikation im Klassenpfad des Konfigurationsservers. Da die Konfigurationsdetails lediglich über eine offene REST-API zur Verfügung gestellt werden, können über den Konfigurationsserver auch Mircoservices, die mittels anderer Technologien implementiert sind, konfiguriert werden.

2.2.3 Spring Cloud Sleuth

Die Fehler-, Performance- und Latenzanalyse bei verteilten Systemen kann sich bei zunehmender Komplexität des Systems als zeitintensiver und schwieriger Prozess gestalten. Spring stellt zur Bewältigung dieses Prozesses das Spring Cloud Sleuth-Projekt zur Verfügung (vgl. Cole u. a. 2017). Sleuth erweitert HTTP-Anfragen mit einem Kor-

relationsidentifikator, welcher es erlaubt Services eindeutig zu identifizieren. Weiteres besteht die Möglichkeit Sleuth mit OpenZipkin zu verbinden, um die Verfolgung einzelner Service-Aufrufe, deren Verarbeitungsdauer auf Service-Ebene und etwaige Fehler visuell darzustellen (vgl. Carnell 2017, S. 260). Sleuth erweitert den Spring Mapped Diagnostic Context (MDC) und bereichert Log-Einträge mit einem „trace“ und einem „span“ Identifikator. Dadurch können Log-Einträge eindeutig einem Service zugeordnet werden und in einen dezidierten Log-Aggregator eingespeist werden.

2.2.4 Netflix Eureka

Eine Herausforderung bei verteilen Systemen ist die Zuordnung der Systemteilnehmer zu ihrer physischen Adresse. Dieses Konzept ist auch bekannt als Service-Discovery. Nach Carnell sei dieses Konzept bei Cloud-basierenden Microservice aus zwei Gründen als äußert wichtig einzustufen: Es erlaube die rasche horizontale Skalierung einzelner Services und die eigentlichen physikalischen Adressen der einzelnen Services würden durch Service-Discovery für etwaige Service-Konsumenten abstrahiert. Dies stelle sicher dass Services miteinander kommunizieren können ohne das einem Service die physikalische Adresse des anderen Services bekannt sei (vgl. Carnell 2017, S. 96ff.). Spring stellt zum Zwecke der Service-Discovery die von Netflix entwickelte, quelloffene, Discovery-Engine Eureka zur Verfügung. Eureka ist ein REST-basierendes Service, welcher die Aufgabe besitzt Microservices zu lokalisieren, registieren und zu überwachen. Zusätzlich dazu enthält Eureka Netflix Ribbon eine mitgelieferte, clientseitige Lastverteilung, welche die Netzwerklast auf die einzelnen Microservices mittels des Rundlauf-Verfahrens gleichmäßig verteilt. Wird ein neuer Microservice oder eine neue Instanz eines bestehenden Microservices hochgefahren, registriert sich dieser bei Eureka. Hierzu werden der Service Name und die entsprechende Adresse gespeichert. Ein Eureka-Server kann diese Information auf beliebig viele andere Eureka Server replizieren. Registrierte Services senden in einem 30-Sekunden-Intervall einen sogenannten Herzschlag (engl. Heartbeat), um Eureka ihren Status mitzuteilen. Die Lastverteilung wird durch Ribbon (siehe Kapitel 2.2.5) gesteuert und funktioniert ohne vorhergehende Konfiguration. Eine effektive Lastenverteilung mache Microservices hoch-skalierbar und erhöhe die Robustheit des Gesamtsystems (vgl. Pivotal Inc. 2017d). Neben der eigentlichen Service-Discovery bietet Eureka ein einfaches Monitoring der registrierten Services.

2.2.5 Netflix Ribbon

Zur Lastenverteilung in verteilten Systemen werden häufig proxy-basierende Lastenverteiler (engl. Load-Balancer) eingesetzt. Alle Client-Anfragen werden zunächst an den Load-Balancer, welcher die Anfragen schließlich unter Berücksichtigung der momentanen Serverlast an einen Server weiterleitet, gerichtet. Ribbon hingegen ist ein clientseitiger Load-Balancer. Der Client (ein Service) richtet seine Anfragen direkt an den Server und verteilt die Last eigenständig. Somit ist kein Proxy von Nöten. Somit entfällt der Proxy als Kapazitätsengpass und „Single Point of Failure“. Clientseitiges Load-balancing erleichtere laut Wolff auch die dynamische Skalierung von Microservice-Anwendungen (vgl. Wolff 2017, S. 333ff.).

2.2.6 Netflix Feign

Netflix Feign ist eine Java-Programmbibliothek, welche HTTP REST Anfragen an Services vereinfachen soll. Im Wesentlichen ist Feign ein Remote-Proxy und eine Abstraktion auf konventionelle HTTP Anfragen. Der Nutzer, die Nutzerin muss die Logik der eigentlichen HTTP Anfrage allerdings nicht selbst implementieren. Es muss lediglich ein Java Interface, welches den Remote-Service repräsentieren soll, bereitgestellt werden. Dieses Interface und dessen Methoden erhalten dann über Java-Annotationen die notwendigen Metadaten zur Kommunikation mit dem Remote-Service. Die Implementierung des Interfaces wird von der Programmbibliothek zur Kompilierungszeit der Anwendung auf Grundlage der angegebenen Metadaten dynamisch generiert.

2.2.7 Netflix Zuul

Auch bei verteilten Systemarchitekturen stellen querschnittliche Belange (engl. Cross-Cutting-Concerns), wie Logging, Sicherheit und das Verfolgen von Nutzerverhalten, über mehrere Service-Aufrufe eine große Herausforderung dar. Diese Belange sollten laut Carnell nicht auf der Ebene einzelner Microservices implementiert werden, sondern in einem dezidierten Service umgesetzt werden. Dieses Service wäre eigenständig und agiere als Filter und Router für alle anderen Microservices. Ein Service dieser Art wird als Service Gateway bezeichnet. Carnell beschreibt ein Service-Gateway als Intermediär zwischen einem Service-Client und dem auszurufenden Service (vgl. Carnell 2017, S. 154ff.). Spring stellt als Service-Gateway die Netflix Bibliothek Zuul zur Verfügung. Zuul stellt die Eingangstür für alle Anfragen von außen und innen an das Gesamtsystem dar und dient als Fassade für die einzelnen Microservices. Zuul

bietet intelligentes Routing und könne als integraler Bestandteil einer Microservice-Architektur angesehen werden (vgl. Pivotal Inc. 2017a). Sendet ein Client eine Anfrage an das Gesamtsystem, verteilt Zuul diese Anfrage an die entsprechenden Services und liefert dem Client die Ergebnisse der Anfrage. So wird verhindert, dass ein Client direkt mit der Menge an Microservices kommunizieren muss, um eine Antwort auf seine Anfrage zu erhalten. Es bietet sich an, die Antworten der einzelnen Microservices am Zuul-Server zu einer wohl-definierten Schnittstelle zu bündeln. Somit kann Zuul auch als API-Gateway genutzt werden. Es können so einzelne Teilantworten der Microservices, zu einer, vom Client erwarteten, Gesamtantwort zusammengeführt werden.

2.2.8 Zipkin

Zipkin ist eine quelloffene Daten-Visualisierungsanwendung, die den Datenfluss über mehrere Microservices hinweg darstellen kann. Es wird die benötigte Verarbeitungszeit einer Transaktion innerhalb jedes beteiligten Services angezeigt. Mithilfe dieser Darstellung können etwaige Performance Hotspots identifiziert werden und es könne laut Carnell eine ungefähre Schätzung der Antwortzeiten einzelner Services getroffen werden. Man solle Zipkin jedoch nicht einem Application Performance Management System verwechseln. Im Gegensatz zu diesen System beschränke sich Zipkin auf Antwortzeiten und könne keine Daten über die Speicher- oder der Central Processing Unit (CPU)-Auslastung liefern. Dennoch sei Zipkin ein wertvolles Werkzeug innerhalb einer Microservice-Architektur (vgl. Carnell 2017, S.274ff.).

2.2.9 Netflix Hystrix

Wie auch in anderen Architektur-Mustern müssen verteilte Systeme in der Lage sein auf Systemfehler entsprechend reagieren zu können um eine möglichst hohe Stabilität zu gewährleisten. Verteile System wären laut Wolff per Definition als Fehleranfälliger einzustufen: Netzwerke und Server seien unzuverlässig und mit einer wachsenden Anzahl an Services auf unterschiedlichen Servern steige simultan auch die Fehleranfälligkeit des Systems. Es müsse vermieden werden, dass der Ausfall eines Services einen Totalausfall des Gesamtsystems zur Folge habe (vgl. Wolff 2017, S. 203). Diese Eigenschaft eines Microservice-Systems wird „Resilience“ genannt. Die Netflix Hystrix Java-Programmbibliothek implementiert gängige Resilience-Muster und stellt diese dem Entwickler, der Entwicklerin über eine einfache Schnittstelle zur Verfügung. Ei-

ne hochwertige Implementierung der Resilience-Muster benötige laut Carnell ein hohes Maß an Wissen über Nebenläufigkeit und komplexem Thread-Pool-Management. Hystrix abstrahiere diese Komplexität und sei eine wohl getestete und produktionsreife Programmbibliothek, welche auch bei Netflix intensiv genutzt wird (vgl. Carnell 2017, S. 126ff.). Intern nutzt Hystrix die Reaktive Extension Programmbibliothek (RxJava), welche Java um asynchrone und event-getriebene Funktionalitäten erweitert. Hystrix kann einfach in Spring-Anwendungen integriert werden und lässt sich, ähnlich wie viele Spring Funktionalitäten, über Java-Annotationen steuern und konfigurieren.

2.2.10 Docker

Docker ist ein Kommandozeilen-Programm und ein im Hintergrund laufender Systemdienst, der es erlaubt Software innerhalb eines isolierten Containers auszuführen. Jede Software die mittels Docker ausgeführt wird, läuft innerhalb eines Containers, was bedeutet dass, sofern nicht explizit konfiguriert, der Prozess keinerlei Zugriff auf Ressourcen außerhalb des Containers besitzt. Im Gegensatz zu virtuellen Maschinen nutzt Docker keine Hardware-Virtualisierung und es wird keine zusätzliche Virtualisierung-Schicht und keine virtuelle Hardware zwischen dem Host-System und dem Docker-Container benötigt. Die Container besitzen über die Container-Engine eine Schnittstelle mit dem Kernel des Betriebssystems. Somit sei Docker laut Nickoloff keine Virtualisierungs-Software sondern nutze lediglich die Container-Technologie, welche bereits im Betriebssystem vorhanden ist (vgl. Nickoloff und Balkan 2016, S. 4ff). Welche Prozesse innerhalb eines Containers ausgeführt werden sollen wird über ein sogenanntes Docker-Image bestimmt. Dieses Image enthält alle Dateien, welche für die Programmausführung innerhalb des Containers notwendig sind und können frei definiert werden. Jedes Image basiert auf einem Basis-Image dem weitere Bestandteile hinzugefügt werden können. Es können beliebig viele Container mit dem selben Image hochgefahren werden und dennoch sind alle Container in sich gekapselt und isoliert. Der Inhalt eines Docker-Images wird über eine Menge von Instruktionen bestimmt. Diese Instruktionen werden in einer Datei, dem Dockerfile, gesammelt. Soll ein neues Docker-Image erstellt werden, liest die Docker-Engine die Instruktionen aus dem Dockerfile und gibt das fertige Image zurück. Aus diesem Image kann folgend ein neuer Container-Prozess gestartet werden (vgl. Jaramillo, Nguyen und Smart 2016, S. 3). Soll Software mittels Docker ausgeliefert werden, müssen lediglich diese Images von einer zentralen Stelle (engl. Registry) heruntergeladen und gestartet werden. Docker erhöhe laut Nickoloff die Sicherheit und ein Nutzer müsse sich keine Gedanken über die Installation, Upgrade-Management und die Kompatibilität mit anderer, am

System laufender Software machen (vgl. Nickoloff und Balkan 2016, S. 6ff).

2.2.11 Gradle

Gradle ist ein quelloffenes, auf Java basierendes Build Management-Tool, welches mit Apache Ant oder Apache Maven, vergleichbar ist. Zur Beschreibung der auszuliefernden Projekte setzt Gradle auf eine eigene, auf Groovy basierende domänenspezifische Sprache. Im Gegensatz zu Maven sind die Gradle-Skripte keine reinen XML-Konfigurationsdateien, welche eingelesen werden, sondern ausführbarer Code, welcher von der Java Virtual Maschine (JVM) interpretiert wird (vgl. Muschko 2014, S.47ff.) Mit Gradle können die benötigten Phasen für den Bau eines Projektes definiert werden. Dazu gehören beispielsweise die Kompilierung, Testausführung und die Verteilung der erstellten Artefakte. Weiteres lassen sich mittels Gradle die Abhängigkeiten eines Projektes zu anderen Projekte oder Fremdbibliotheken definieren (engl. Dependency Management). Dazu nutzt Gradle standardmäßig, wie Maven, die weitverbreiteten Maven-Repositories. Gradle lässt sich mittels Plugins, erweitern. Es existieren bereits einige Plugins welche die Integration von Gradle in andere Technologien ermöglichen. Die Auslieferung und das Management mehrerer Microservices würde durch den Einsatz eines Build Management-Tools wie Gradle deutlich vereinfacht, da das Gesamtsystem gleichzeitig gebaut, verteilt und ausgeliefert werden kann (vgl. Muschko 2014, S.338ff.).

2.2.12 CircleCI

CircleCI ist ein cloud-basierender kontinuierlicher Integrations-Dienst (engl. Continuous integration). Im Gegensatz zu On-Premise Lösungen wie Jenkins bedarf es keiner Installation auf einem Server. Somit entfallen auch alle Wartungs- und Verwaltungstätigkeiten. CircleCI kopiert den Quellcode der Anwendung aus einem Versionsverwaltungssystem und führt den Build-Prozess und etwaige andere Prozesse aus. Zum Zeitpunkt der Veröffentlichung dieser Arbeit werden die Online-Dienste Github und Atlassian BitBucket unterstützt. Beide Online-Dienste nutzen das kollaborativ Versionsverwaltungssystem Git. CircleCI unterstützt die Erstellung von Workflows. Diese erlauben es die unterschiedlichen Phasen des Auslieferungsprozesses abzubilden. Ferner bedarf es nur geringfügiger Konfiguration um CircleCI nutzen zu können. Es muss lediglich eine YAML Ain't Markup Language (YAML) Datei `.circleci/config.yml)` im Quellcode Repository befinden. Über diese Datei kann CircleCI vollständig

konfiguriert werden (vgl. CircleCI 2017).

2.2.13 HTTP REST

REST ist im Wesentlichen eine Abstraktion auf das Verhalten des modernen, Hyper-Media getriebenen, World-Wide-Web und keine konkrete Technologie. (vgl. Spichale 2017, S. 143) Die meisten Webseiten sind über eine Adresse, einen sogenannten Uniform Resource Locator (URL) erreichbar, wobei jede URL eine Ressource darstellt. Sendet ein Web-Browser eine HTTP-Anfrage, sendet der Server eine entsprechende Antwort. Diese Antwort, sei es nun ein Hypertext Markup Language (HTML)-Dokument oder eine Grafik, ist die Repräsentation einer einzigartigen, identifizierbaren Ressource. Jede URL solle genau eine Ressource repräsentieren (vgl. L. Richardson und Mike 2013, S. 34ff.). Diese Repräsentation stelle laut Webber et. al. die Transformation oder Anzeige einer Ressource zum momentanen Zeitpunkt dar. Somit sind Ressourcen nicht statisch, sondern bilden den gegenwärtigen Zustand einer Ressource ab (vgl. Webber, Parastatidis und Robinson 2010, S. 26). Die externe Darstellung einer Ressource sei völlig losgelöst von ihrer internen Repräsentation (vgl. Newman 2015, S.99). Die Abbildung einer Ressource innerhalb einer RESTful Schnittstelle geschieht meist über die unabhängigen Aufzeichnungssprachen Extensible Markup Language (XML) oder JavaScript Object Notation (JSON). So wird die Interoperabilität zwischen API-Konsument und API-Anbieter gewährleistet. Systeme unterschiedlichster Plattformen und Programmiersprachen können dadurch zusammen funktionieren und kommunizieren.

Die möglichen Zustande und die Verbindungen zu anderen Ressourcen können über Hyperlinks, welche auf den weitere Zustände oder auf andere Ressourcen weiterleiten, abgebildet werden. Diese Vorgehensweise wird als Hypermedia As The Engine Of Application State (HATEOAS) bezeichnet (vgl. Webber, Parastatidis und Robinson 2010, S. 14). Dieser Ansatz ist besonders in der automatischen Verarbeitung von Ressourcen praktikabel und wird häufig für REST APIs eingesetzt. Newman beschreibt REST Schnittstellen in Verbindung mit HTTP als ein geeignetes Mittel zur Kommunikation und Schnittstellen-Definition für Microservices. Allerdings sei die Performance von HTTP als Kommunikationsprotokoll fragwürdig, da es nicht für niedrige Latenzen ausgelegt sei. Vergleichbare, auf dem Transmissition Control Protocol (TCP) aufbauende Netzwerkprotokolle, wären ebenfalls für Microservices geeignet, vor allem wenn niedrige Latenz-Zeiten benötigt würden (vgl. Newman 2015, S. 105).

2.3 Abgrenzung zur eigenen Arbeit

Die im Kapitel 2.2 vorgestellten Technologien und Techniken bilden die technische Grundlage der im Kapitel 3 vorgestellten Prototyp-Applikation. Anhand der Prototyp-Applikation soll demonstriert werden, wie eine Microservice-Architektur mit dem Spring-Framework und den weiteren, im Kapitel 2.2 vorgestellten Technologien, umgesetzt werden kann. Außerdem sollen auch Unterschiede und Gemeinsamkeiten zu einer monolithischen Architektur aufgezeigt werden. Um den Umfang dieser Arbeit im Rahmen zu halten wurde auf die Implementierung einer monolithischen Applikation, um etwaige direkte Vergleiche anzustellen verzichtet. Dies, oder die Refaktorisierung einer bestehenden monolithischen Applikation zu einer Microservice-Applikation könnten in einer weiterführenden Arbeit behandelt werden.

Kapitel 3

Implementierung

Ausgehend der im Kapitel 2 vorgestellten Technologien, Techniken und Konzepte wird im folgenden Kapitel auf die praktische Umsetzung eines, auf der Microservice-Architektur basierenden, Prototypen eingegangen. Es wird die Architektur der Prototyp-Applikation vorgestellt, Design-Entscheidungen erläutert und auf die einzelnen Komponenten und Services des Systems eingegangen. Das im Zuge dieser Arbeit erstellte System stellt lediglich mögliche Ansätze einer Microservice-Architektur, von der Gesamtarchitektur des Systems bis hin zur Auslieferung, dar. Zusätzlich werden etwaige Unterschiede zu monolithischen Architekturen untersucht.

3.1 Idee, Aufbau und Umsetzung

Die Prototyp-Applikation soll ein vereinfachtes Flug-Management-System darstellen. Es erlaubt dem Benutzer, der Benutzerin Flüge zu erstellen und diese in Echtzeit auf einer Karte zu verfolgen. Die Applikation bedient sich dem Microservice-Architektur-Muster und jeder Business-Service stellt ein Self-Contained-System dar. Dies bedeutet, dass jeder Service seine eigene Persistierungs- und Businessschicht besitzt. Außerdem läuft jeder Service in einem eigenständigen Prozess. Die Business-Services stellen eine HTTP RESTful-API zur Verfügung welche als wohl-definierte Schnittstelle und Kommunikationskanal zwischen den einzelnen Services dient. Bei der Erstellung der Kommunikationsschnittstellen wurde besonderes Augenmerk auf eine optimale Vorgehensweise und empfohlenen Entwicklungstechniken gelegt. Sämtliche RESTful APIs wurden nach dem HATEOAS- und Command Query Separation (CQS) Prinzip erstellt. Das CQS Prinzip separiert Kommandos (engl. Commands), die Seiteneffekte aufweisen und den Zustand des Systems ändern von Operationen die lediglich Daten

abfragen (engl. Query). Dies erlaubt eine automatisierte Konsumtion der Schnittstelle durch andere Service und macht das System einfacher verständlich.

Um die Komplexität überschaubar zu gestalten, wurden neben der für die Infrastruktur benötigen Services zwei Business-Services implementiert:

Flight-Service

Die Aufgabe des Flight-Service ist die zentrale Verwaltung der Flüge und der flugspezifischen Metadaten im System. Flüge können mittels einer REST-API oder über ein Angular 2 User Interface (UI) erstellt, bearbeitet und gelöscht werden. Wird ein neuer Flug erstellt, übermittelt der Flight-Service den neu erstellten Flug, also eine Repräsentation der Daten, mittels HTTP POST an den Flight-Monitoring-Service. Außerdem stellt der Flight-Service flugspezifischen Metadaten, wie beispielsweise die Liste aller verfügbaren Flughäfen über eine REST API für etwaige andere Services zur Verfügung. Diese Daten können von anderen Services zwar gelesen, jedoch nicht bearbeitet oder gelöscht werden. Sämtliche Daten werden beim Start des Services aus Comma-separated Values (CSV) Dateien in eine H2 In-Memory-Datenbank eingespeist und mittels Spring Data Java Persistence API (JPA) und Spring Web als REST API zur Verfügung gestellt.

Flight-Monitoring-Service

Der Flight-Monitoring-Service dient dem Zweck der Echtzeit-Überwachung (engl. Monitoring) der Flüge. Dazu werden Informationen über neu erstellte Flüge durch den Flight-Service benötigt. Der Flight-Monitoring-Service tritt als Konsument der vom Flight-Service emittierten Flüge auf. Die vom Flight-Service erhaltenen Flugdaten werden verarbeitet und in eine MongoDB Datenbank gespeichert. Der Flight-Monitoring-Service übernimmt auch die Berechnung der geographischen Koordinaten der momentan im System vorhandenen Flüge. In einem 2-Sekunden-Intervall werden, ausgehend von der letzten Position die neuen geographischen Koordinaten des Flugzeugs errechnet und in die Datenbank persistiert. Diese Daten werden über eine Schnittstelle mittels Server-sent Events (SSE) als Endlos-Stream emittiert und für der UI zur Verfügung gestellt. Die Daten werden auf der UI visualisiert und auf einer OpenStreetMap-Karte in Echtzeit angezeigt.

Sämtliche Infrastruktur- und Business-Services wurden mit Java 8 und Spring Boot implementiert. Abhängig von der benötigten Funktionalität des Services wurde wahlwei-

se Spring Boot 1.5.6 oder Spring-Boot 2.0.0 Milestone 3 genutzt. Grund dafür die native Unterstützung des reaktiven und funktionalen Programmierparadigmas von Spring-Boot 2.0.0 Milestone 3. Jede Einheit der Prototyp-Applikation kann autark wahlweise als Docker-Container oder als ausführbare JAR-Datei hochgefahren und ausgeliefert werden.

3.2 Architektur des Gesamtsystems

Die größte Herausforderung bei der Architektur von Microservice-Systemen ist die Abgrenzung und Aufteilung der Services gemäß der fachlichen Anforderungen an das Gesamtsystem. Microservices sollten laut Wolff eine fachliche Domäne, wie beispielsweise Geschäftsprozesse modellieren. Dies ließe sich durch eine Aufteilung mittels Domain-driven Design (DDD) und des Bounded Context erreichen (vgl. Wolff 2017, S. 102). Eine Domäne kann aus einer Menge von Bounded Contexts bestehen, wobei jeder Bounded Context aus internen und externen Modelle besteht. Externe Modelle sind jene Modelle, die mit anderen Bounded Contexts geteilt werden. Ein Bounded Context besitzt eine wohl-definierte Schnittstelle über die festgelegt wird welche Modelle mit anderen Bounded Contexts geteilt werden sollen. Ein Geschäftsprozess wie beispielsweise Auftragserstellung könnte einen Bounded Context darstellen. Dieser Ansatz ermöglicht eine klare Trennung zwischen internen und externen Datenmodellen wobei die interne Repräsentation und die externe Repräsentation eines Modells durchaus Unterschiede aufweisen können (vgl. Newman 2015, S. 67ff.).

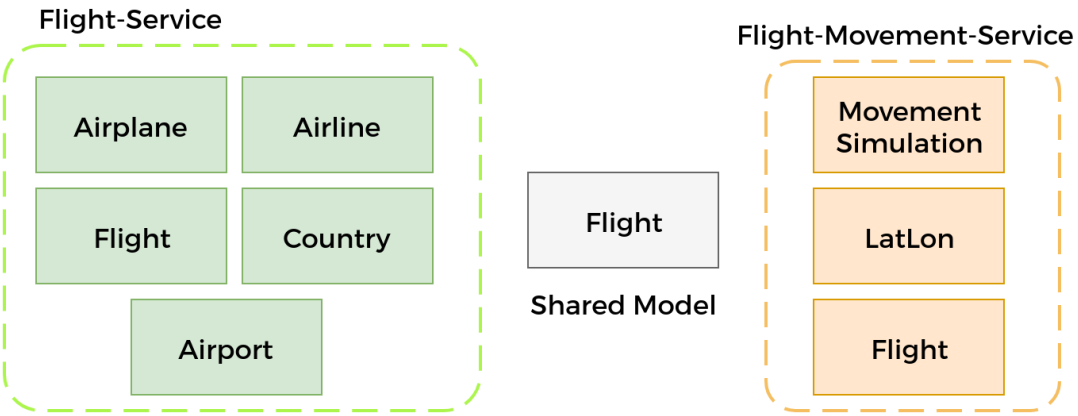

Abbildung 3.1: Grafische Darstellung des Bounded Context des Flight- und Flight-Monitoring-Service

Die Abbildung 3.1 zeigt den Bounded-Context des Flight- und Flight-Monitoring-Service. Die Services teilen sich das Flight-Modell, wobei sich die internen und exter-

nen Repräsentationen des Modells allerdings je nach Kontext hinsichtlich Datentypen und Eigenschaften unterscheiden.

Eine korrekte fachliche Aufteilung hilft dabei die von Carnell definierten vier Charakteristiken einer Microservice-Architektur zu erfüllen (siehe 2.2). Unter Rücksichtnahme dieser Charakteristiken steht der Software-Architekt, die Software-Architektin vor der Aufgabe das Business-Problem zu zerlegen, die Granularität der Services zu bestimmen und die Service-Schnittstellen zu definieren (vgl. Carnell 2017, S. 36ff).

Durch die Zerlegung des Business-Problems in kleine Teile ergibt sich bereits ein vereinfachtes Modell der möglichen Microservices. Dieses Modell kann als Grundlage einer möglichen Architektur heran gezogen werden. Ist dieses Modell noch zu grobgranular sollten die Microservices in kleinere Microservices refaktoriert werden. Allerdings müsse laut Carnell darauf geachtet werden, dass Microservices nicht bloß reine Create Update Read Delete (CRUD)-Services seien. Microservices wären als eine Darstellung von Business-Logik zu verstehen und keine reine Abstraktion von Datenquellen (vgl. Carnell 2017, S. 43).

Die angemessene Größe eines Microservice ist schwer zu definieren. Allerdings sollten die individuellen Microservices möglich klein, im Sinne der Codebasis gehalten werden. Nach Newman können kleine Microservices kostengünstig durch bessere Implementierungen ersetzt werden und eine kleine Codebasis kann einfacher verwaltet und getestet werden. Vergleichsweise ist es bei monolithischen Anwendungen mit einer großen Codebasis schwieriger und teurer Fragmente auszutauschen (vgl. Newman 2015, S. 27). Das Gesetz von Conway besagt, dass eine Software-Anwendung immer die Struktur des Unternehmens, das die Software entwickelt hat widerspiegelt (vgl. Conway 1968, S. 28ff.). Nach diesem Gesetz sollte demnach jeder Microservice von einem kleinen Entwicklungsteam entwickelt werden um den Microservice selbst klein zu halten.

Bei der Definition der Service-Schnittstellen sollte auf einige Qualitätsmerkmale geachtet werden: Laut Spichale sollte eine API konsistent, intuitiv verständlich, stabil, minimal und erweiterbar sein (vgl. Spichale 2017, S. 23). Ein besonderes Augenmerk sollte in diesem Zusammenhang auf die Erweiterbarkeit der Schnittstelle gelegt um auf etwaige Änderungen der Anforderungen reagieren zu können. Ein weiterer wichtiger Aspekt ist die Stabilität der API. Die Stabilität stünde laut Spichale nicht im Widerspruch zur Erweiterbarkeit, denn eine erweiterte API sollte weiterhin kompatibel sein (vgl. Spichale 2017, S.22ff.). Die Versionierung, Rückwärts- und Vorwärtskompabilität stellen große Herausforderungen an die Schnittstellen der einzelnen Services. Durch die Nutzung von dynamischen Dateiformaten, wie beispielsweise JSON, kann

die Kompatibilität aufrecht erhalten werden.

Neben der Aufteilung gemäß der fachlichen Anforderungen kann zwischen Business- und Infrastruktur-Services unterschieden werden. Infrastruktur-Services erfüllen, wie ihre Business Gegenstücke ebenfalls eine wohl definierte Aufgabe. Ein Infrastruktur-Service sollte keine Business-Aufgaben erfüllen und vice versa. Ein Beispiel eines Infrastruktur-Services im Kontext des Prototypen ist das Eureka-Service (siehe Kapitel 2.2.4). Dieses erfüllt eine wohl-definierte Aufgabe: Die Service-Discovery.

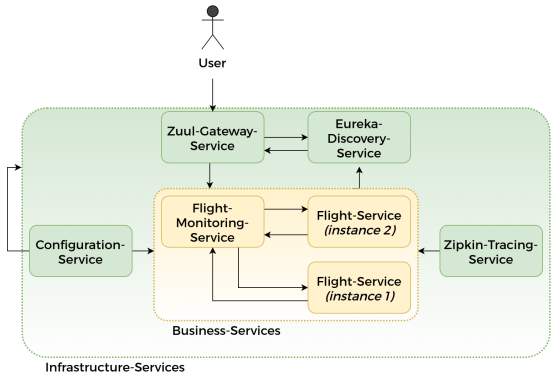

Abbildung 3.2: Grafische Darstellung der Architektur des Gesamtsystems

3.2.1 Business-Services

Microservices sollten, wie auch Services in ein einer monolithischen Applikation, eine bestimmte Business-Funktionalität, wie beispielsweise Prozesse abbilden. Neben der Implementierung der Business-Logik enthält ein Microservice, sofern benötigt, eine Datenquelle und gegebenenfalls das User-Interface für die abzudeckende Business-Funktionalität. Es erfolgt also eine Aufteilung in Komponenten durch Services, wobei die Services über eine Form von Remote Procedure Call (RPC) kommunizieren. Fowler definiert eine Komponente als unabhängig austauschbare und erweiterbare Software-Einheit (vgl. Fowler und Lewis 2014). Eine Änderung einer Komponenten im Monolithen hat zur Folge, dass die gesamte Applikation den Build-Prozess durchlaufen und erneut ausgeliefert werden muss. (vgl. Bakshi 2017, S. 3).

3.2.2 Infrastuktur-Services

Infrastruktur-Services spielen im Kontext der Gesamt-Architektur eine wichtige Rolle, denn ohne die Infrastuktur-Services würde ein sinnvoller und gewinnbringender Einsatz einer Microservice-Architektur massiv erschwert. Als Beispiel ließe sich die dyna-

mische Skalierung einzelner Services und die Inter-Service-Kommunikation nennen: Ohne Service-Discovery wäre es schwierig die Service-Kommunikation und dynamische Skalierung zu ermöglichen. Die Adressen der Services müssten in jedem Service hart-codiert werden. Dies würde das Nutzen einer Cloud-Umgebung, wo Adressen meist dynamisch vergeben werden, massiv erschweren. Es bestünde dadurch eine sehr starke Koppelung zwischen den Services und das Gesamtsystem wäre sehr statisch und änderungsresistent. Die Infrastuktur-Services sind somit als wichtiger Grundbaustein einer dynamischen und skalierbaren Microservices-Anwendung anzusehen. Jeder Infrastuktur-Service der Prototyp-Applikation wurde mit Spring Boot umgesetzt und in den folgenden Kapiteln wird die Integration dieser Services in das Gesamtsystem und die Umsetzung untersucht.

Infrastruktur-Services dienen demnach dazu die Komplexität eines verteilten Systems zu reduzieren.

3.2.3 Inter-Service Kommunikation

Jeder Business-Service der Prototyp-Applikation stellt eine RESTful API zur Inter-Service-Kommunikation zur Verfügung. Als Kommunikationskanal dient HTTP und als Austauschformat JSON. Grundlage für diese Entscheidung ist die Einfachheit und Erweiterbarkeit einer solchen API und der geringe Overhead des JSON Formats. Die Nutzung einer RESTful HTTP API vereinfacht auch die Anbindung eines Web Front-Ends für jeden Service. Beziehungen zwischen den Ressourcen lassen sich gemäß dem HATOAS-Prinzip mittels Hyperlinks abbilden (vgl. Wolff 2017, S.177). Neben den oft genutzten textuellen Formaten wie JSON oder XML können auch Binär Protokolle wie Google Protocol Buffers als REST-Austauschformat dienen (vgl. Google Inc. 2017).

Monolithische Applikationen benötigen oftmals intelligente Kommunikationssysteme, um die Kommunikation zwischen den Applikationsschichten sicherzustellen. Ein Ansatz ist die Nutzung eines ESB, welcher Routing, Choreographie und Transformationen für die Kommunikation zur Verfügung stellt. Microservices nutzen, so Fowler, stattdessen einen anderen Ansatz: Intelligente Endpunkte und dumme Kanäle (engl. smart endpoints and dump pipes). Simple Protokolle zur Kommunikation wären demnach zu bevorzugen (vgl. Fowler und Lewis 2014).

Ebenso wären Messaging Technologien wie Apache Kafka, RabbitMQ oder ZeroMQ eine valide Kommunikationsmöglichkeit zwischen den Services, allerdings, so Wolff, würde durch die Einführung eines Messaging Dienstes die Komplexität des Gesamtsystems erhöht: Es müsse ein weiterer Infrastruktur-Service in Form eines Messaging-

Servers zur Verfügung gestellt werden. (vgl. Wolff 2017, S.183). Allerdings sind bei Messaging-Technologien der Sender und der Empfänger bereits per Definition entkoppelt, da sie lediglich den Channel am Messaging-Server kennen müssen. Dies führe laut Wolff dazu, dass bei der Nutzung von Messaging-Technologien die Notwendigkeit eines System-Discovery-Service entfallen würde Wolff 2017, S.141). Ein weiterer Vorteil von Messaging-Systemen gegenüber REST ist die asynchrone Natur von Messaging-Diensten und die Nutzung des Publish-Subscribe Musters. Ein Service kann Informationen in die Message-Queue emittieren (Publisher), welche dann von anderen Services konsumiert werden können (Subscriber). Wie und ob die anderen Services auf diese Information reagieren wird ihnen selbst überlassen. Der Publisher selbst hat keinerlei Information darüber welcher andere Service die emittieren Nachrichten konsumiert. Mittels des Publish-Subscribe-Muster kann die Koppelung zwischen den Services noch weiter reduziert werden.

Wichtig sei jedoch, so Wolff dass die Inter-Service-Kommunikation und Integration von Microservices einheitlich über das gesamte Projekt festgelegt sein sollte (vgl. Wolff 2016, S. 34).

3.3 Architektur eines Spring Boot Microservices

Die Architektur der einzelnen Business-Services ist vergleichbar mit der Architektur monolithischer Webanwendungen und kann in drei Schichten (siehe Abbildung 3.3) unterteilt werden.

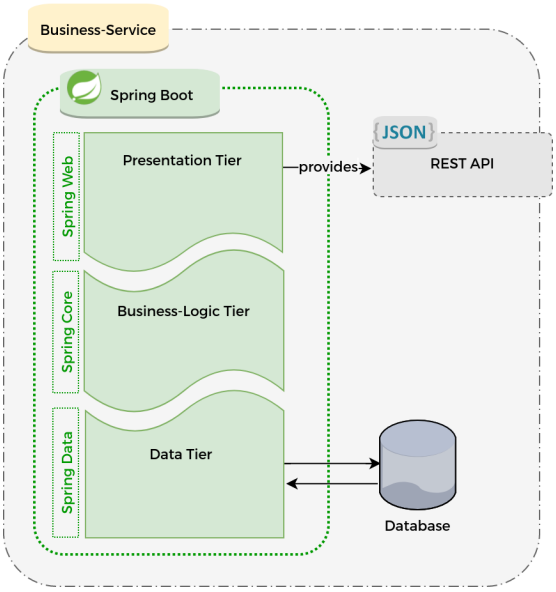

Abbildung 3.3: Schichten Architektur eines Spring Boot Business-Microservice

Jeder Microservice, sei es Business-Service oder Infrastruktur-Service, der Prototyp-Applikation wurde mit Spring Boot umgesetzt. Durch den Einsatz der Starter-Dependencies können die Microservices bereits auf ihren jeweiligen Einsatzzweck vorbereitet werden. Beispielsweise hält jeder Business-Service eine Abhängigkeit auf die Starter-Dependency `org.springframework.boot:spring-boot-starter-web`. Diese Abhängigkeit enthält sämtliche notwendige Abhängigkeiten, um eine RESTful HTTP API zur Verfügung zu stellen.

```
1  ...
2  @RestController
3  @CrossOrigin
4  @RequestMapping("${api.version}/country")
5  public class CountryController extends GenericCrudController<
       CountryResource> {
6
7          @Autowired
8          public CountryController(CountryDataService dataService) {
9                  super(dataService);
10         }
11
12         @GetMapping(params = "code", produces = MediaType.
               APPLICATION_JSON_VALUE)
13         public ResponseEntity<CountryResource>
               getCountryByCountryCode(@RequestParam String code) {
14                 CountryDataService countryDataService = (
                       CountryDataService) dataService;
15
16                 return new ResponseEntity<>(countryDataService.
                       findByCountryCode(code), HttpStatus.OK);
17         }
18 }
```

Listing 3.1: Quellcode des Flight-Service Spring CountryController

Das Listing 3.1 zeigt den Quellcode eines REST API-Controllers des Flight-Service. Der `CountryController` stellt eine Schnittstelle zur Verfügung, um Informationen über ein Land mittels des ISO 3166-1 alpha-2 Ländercodes abzufragen. Die RESTful API kann von anderen Microservices oder von etwaigen Front-End Implementierungen konsumiert werden. In der Prototyp-Applikation stellt jeder Service seine eigene Front-End-Implementierung bereit.

Die Datenzugriffsschicht wurde mit Spring Data realisiert. Spring Data reduziert den notwendigen Boilerplate-Code und stellt eine Abstraktion auf JPA oder den MongoDB Java-Treiber dar. Es genügt Java-Interfaces zur Verfügung zu stellen, um eine CRUD

Schnittstelle zur Datenbank zu erhalten. Listing 3.2 zeigt ein solches Interface.

```
1 ...
2 @Repository
3 public interface FlightRepository extends JpaRepository<Flight, Long
      > {
4         Optional<Flight> findByUuid(UUID uuid);
5
6         Optional<Flight> findFlightByFlightNumber(String
              flightNumber);
7 }
```

Listing 3.2: Quellcode des Flight-Service FlightRepository

Die Logikschicht enthält sämtliche Businesslogik, wie beispielsweise die Berechnung der geographischen Koordinaten eines Fluges.

Es zeigt sich, dass einzelne Microservices ähnlich zu monolithischen Anwendungen aufgebaut werden können um die Anforderungen an einen Microservice effizient umsetzen zu können. Es können die gleichen Architektur-Muster und Clean-Code-Prinzipien eingesetzt werden.

Ferner können diese Gemeinsamkeiten ausgenutzt werden, um monolithische Anwendungen bei Bedarf in eine Microservice-Anwendung zu transformieren. Fowler empfiehlt bei neuen Anwendungen zunächst den klassischen Ansatz zu wählen und eine monolithische Anwendung zu entwickeln und diese, sofern die Umstände dies erfordern schrittweise in eine Microservice-Anwendung zu transformieren (vgl. Fowler 2015). Auch Newman warnt davor Microservices als “Silver-Bullet”, also die Lösung aller Probleme anzusehen. Die verteilte Natur von Microservices und daraus entstehende Komplexität seien keinesfalls zu unterschätzen (vgl. Newman 2015, S. 33).

3.4 Verteile Konfiguration mit Spring Cloud Config

Zur Konfiguration der Microservices und der Infrastruktur-Dienste wurde in der Prototyp-Applikation der Spring Cloud Config-Server eingesetzt. Grundsätzlich ist dieser Dienst ein Key-Value-Store, der Konfigurationsparameter über eine REST API zur Verfügung stellt. Die Konfigurationsparameter können unter anderem vom Dateisystem des Servers, dem Klassenpfad oder von einem Git Repository eingelesen werden. Es besteht die Möglichkeit Konfigurationswerte zu verschlüsseln, um kritische Parameter wie beispielsweise Passwörter zu schützen. Die Verwendung eines Git Repositories als Speichort für die Konfigurationsdateien bietet entscheidende Vorteile gegenüber den ande-

ren Optionen: Die Versionierung der Konfigurationsdateien und bei Änderungen der Konfigurationsdetails muss der Spring Cloud Config-Server nicht erneut gebaut und hochgefahren werden. Aus diesem Grund wurde in der Prototyp-Applikation Git als Backend für Spring Cloud Config eingesetzt. Listing 3.3 zeigt die Konfiguration des Spring Cloud Config-Servers der Prototyp-Applikation. Auch Carnell empfiehlt aus den genannten Gründen die Nutzung eines eigenen Git Repositories als Speicherort für die Spring-Konfigurationsdateien (vgl. Carnell 2017, S. 74).

In monolithischen Applikationen sind die Konfigurationsdateien oft über mehrere Projekte verteilt und liegen in verschiedenen Klassenpfaden. Daraus kann eine hohe Komplexität und eventuell Bugs in der Applikation entstehen. Zusätzlich erfordert die Änderung eines Konfigurationswertes meist erneutes Ausführen des Build-Prozesses um die Änderungen wirksam zu machen (vgl. Carnell 2017, S. 66).

Neben dem Vorhandensein der Kompilierzeitabhängigkeit auf `org.springframework.cloud:spring-cloud-config-server`, muss die Bootstrap-Klasse der Anwendung mit der Annotation `@EnableConfigServer` versehen werden. Die weitere Konfiguration des Spring Cloud Config-Servers kann in der lokalen `application.yml` Konfigurationsdatei vorgenommen werden.

```
1  # Vishnu configuration server properties
2  server:
3          port: 8888
4
5  spring:
6          cloud.config.server:
7                  git:
8                          uri: https://github.com/fridayy/vihsnu-
                               config
9          application:
10                 name: vishnu-config
11
12 eureka:
13         instance:
14                 preferIpAddress: true
```

Listing 3.3: Die Konfigurationsdatei des Spring Cloud Config Servers - application.yml

Wie auch alle anderen Services wurde der Spring Cloud Config-Server in der Prototyp-Applikation bei Eureka (siehe Kapitel 3.5) registriert, um vom clientseitigen Load-Balancing und der Service-Discovery zu profitieren. Auch der Konfigurationsserver kann so beliebig skaliert werden.

Die Clients des Spring Cloud Config-Servers benötigen die Abhängigkeit auf die Starter-Dependency `org.springframework.cloud:spring-cloud-starter-config` und die Adresse des Spring Cloud Config Servers. Die Adresse des Config Servers wird in der lokalen Klassenpfad Datei `bootstrap.yml` angegeben. Diese Konfigurationsdatei wird von Spring sofort nach dem Start der Applikation geladen, noch bevor die automatische Konfiguration von Spring Boot ausgeführt wird.

```
1 spring:
2   application:
3     name: vishnu-flight-monitoring-service
4   cloud:
5     config:
6       uri: http://vishnu-config:8888
```

Listing 3.4: bootstrap.yml des Flight-Monitoring-Service

Das Listing 3.4 zeigt die `bootstrap.yml` Datei des Flight-Monitoring-Service. Neben der Adresse des Konfigurationsservers wird auch der Name der Applikation definiert. Dieser Name muss mit dem Ordnernamen, in welchem sich die Konfigurationsdateien des Services befinden, übereinstimmen, da der Konfigurationsserver die Namen der von ihm verwalteten Ordner mit dem Namen der anfragenden Spring-Applikation abgleicht. Zusätzlich können mehrere Konfigurationsdateien für unterschiedliche Spring-Profile abgelegt werden. Je nach aktiviertem Spring Profil des Clients, wird die gewünschte Konfigurationsdatei vom Spring Cloud Config-Server geladen.

Der Spring Cloud Config-Server ist nahtlos in Spring integriert und die geladenen Konfigurationsparameter können vom Entwickler, der Entwicklering wie lokale Konfigurationsparameter genutzt werden. Für den Entwickler, die Entwicklerin eines Microservices ist seitens des Abstraktionsniveaus kein Unterschied zur Verwendung lokaler Konfigurationsdateien wie dem `application.yml` zu erkennen. Ist der Konfigurationsserver nicht erreichbar, oder kann keine Konfiguration für den anfragenden Service gefunden werden fällt Spring auf die lokalen Konfigurationsdateien zurück.

Der Spring Cloud Config Server vereinfacht die zentralisierte Konfiguration von Microservices um ein hohes Maß. Im Zusammenspiel mit einem Versionsverwaltungssystem wie Git können die Konfigurationen effizient verwaltet werden. Er ist in der Lage auf Änderungen der Konfigurationsdateien im Git Repository zu reagieren und stellt immer die neueste Revision der Konfigurationsparameter zur Verfügung. Die Clients des Konfigurationsserver fragen die Konfigurationsparameter standardmäßig nur beim Start der Applikation, und nicht während der Laufzeit, ab. Dies bedeutet, dass alle Instanzen des Services, welches die neuen Konfigurationsdetails benötigt neu gestartet

werden müsste um die neuen Parameter zu erhalten. Zwar ist es möglich mit Hilfe des Spring Boot Actuator Projekts und der Annotation `@RefreshScope` die Konfigurationsdetails dynamisch zur Laufzeit der Applikation abzufragen, allerdings werden lediglich benutzerdefinierte Konfigurationsparameter aktualisiert. Werden Spring interne Konfigurationsparameter, wie beispielsweise Datenbank-Konfigurationsparameter, geändert, muss die Applikation trotzdem neu gestartet werden, damit die Konfiguration greift. Dieses Verhalten liegt am Mechanismus der automatischen Konfiguration von Spring: Anhand von Konfigurationsparametern werden bestimmte Autokonfigurationsklassen geladen.

3.5 Service Discovery mit Spring Cloud Eureka

Der Service-Discovery-Dienst kann als Basis der Kommunikation der Microservices angesehen werden. Er dient als Anlaufstelle für die Microservices um mit anderen Services kommunizieren zu können. Es wäre theoretisch möglich auf Service-Discovery zu verzichten und alternative Ansätze wie die DNS-Namensauflösung und einen Netzwerk-Load-Balancer zu nutzen. Carnell rät von dieser Vorgehensweise, aufgrund der limitierten Skalierbarkeit und der Komplexität, ab. Auch würde diese Vorgehensweise eine zentrale Schwachstelle des Systems (engl. Single Point of Failure) darstellen (vgl. Carnell 2017, S.98ff.). Laut Wolff sollte jede Microservice-Architektur Service Discovery von Anfang an als festen Bestandteil der Architektur integrieren (vgl. Wolff 2017, S. 143). Monolithische Anwendungen benötigen keine Service Discovery, da jeder Service bereits innerhalb der eigentlichen Applikation vorhanden ist.

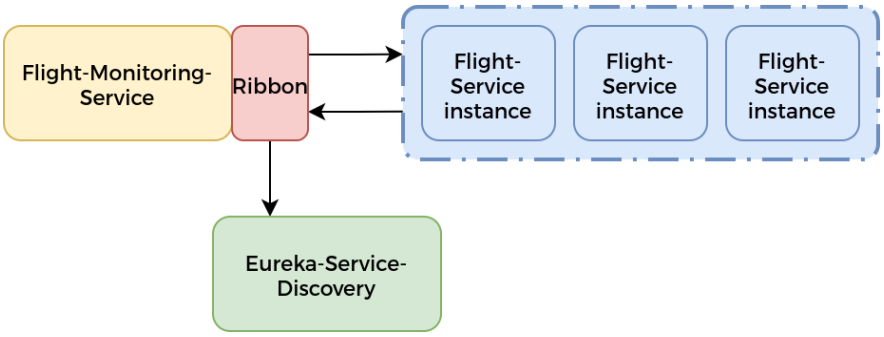

Abbildung 3.4: Grafische Darstellung der Funktionsweise von Eureka und Ribbon

Im Prototyp-System wurde Spring Eureka (siehe 2.2.4) als Service Discovery Dienst und Netflix Ribbon (siehe 2.2.5) als clientseitiger Load-Balancer eingesetzt. Neben der Erstellung eines neuen Gradle-Projekts muss lediglich die Kompilierzeit-Abhängigkeit auf `org.springframework.cloud:spring-cloud-starter-eureka-server` in der

`build.gradle` Datei angeben werden. Befindet sich diese Abhängigkeit im Klassenpfad wird dies von Spring erkannt und es wird die von Spring Boot vordefinierte Autokonfiguration in den Kontext injiziert.

Die Abbildung 3.4 stellt die Funktionsweise von Ribbon in Verbindung mit Eureka vereinfacht dar. Werden Instanzen der Microservices hochgefahren, registrieren sie ihre Adresse bei Eureka. Findet nun eine Transaktion zwischen dem Flight-Monitoring- und dem Flight-Service statt, werden die Adressen der Flight-Service Instanzen von Eureka abgefragt und von Ribbon zwischengespeichert. Bei weiteren Transaktionen werden die Adressen direkt von Ribbon abgerufen und nicht bei Eureka angefragt. In einem festgelegten Intervall erneuert Ribbon die zwischengespeicherten Adressen. Ribbon agiert also als clientseitiger Cache für die Adressen der Microservice-Instanzen (vgl. Carnell 2017, S.103ff.).

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

Listing 3.5: Bootstrap-Klasse des Eureka Services

Durch die Annotation `@EnableEurekaServer` läd Spring die Konfigurationsklasse `EurekaServerAutoConfiguration.java`, welche Eureka mit Standard-Einstellungen bereitstellt. Diese können ohne weiteres mittels der Konfigurationsdatei vom Konfigurationsserver überschrieben werden. So wurden folgende Einstellung überschrieben:

```
1 server:
2   port: 8761
3   waitTimeInMsWhenSyncEmpty: 5
4
5 eureka:
6   client:
7     registerWithEureka: false
8     fetchRegistry: false
```

Listing 3.6: Eureka Konfiguration

Besonderes Augenmerk ist auf die Einstellungen `eureka.client.registerWithEureka` und `eureka.client.fetchRegistry`. Die erste Einstellung verhindert, dass sich der

Eureka-Server bei sich selbst registriert. Die zweite Einstellung verhindert ein lokales Caching der Registry-Informationen, da diese am Eureka-Server immer aktuell sein sollten. Diese Einstellung sind bereits ausreichend um einen voll funktionsfähigen Eureka Server betreiben zu können.

Alle weiteren Services wurden durch das Hinzufügen der Kompilierzeit-Abhängigkeit `org.springframework.cloud:spring-cloud-starter-eureka` und der Annotation `@EnableEurekaClient` als Eureka-Client ausgezeichnet. Durch die Annotation lädt Spring die entsprechende Konfigurationsklasse `EurekaDiscoveryClientConfiguration`, welche beim Start der Applikation dieselbe bei Eureka registriert. Es sollte darauf geachtet werden, dass die Applikation einen eindeutigen Namen besitzt. Dieser wurde für jeden Service in der korrespondierenden Konfigurationsdatei am Konfigurationsserver vergeben.

```
...
eureka:
  instance:
    preferIpAddress: true
  client:
        registerWithEureka: true
        fetchRegistry: true
        serviceUrl:
                defaultZone: http://localhost:8761/eureka/
spring:
  application:
    name: vishnu-flight-service
...
```

Listing 3.7: Konfigurationsdatei des Flight Service

Das Listing 3.7 zeigt einen Ausschnitt der Konfigurationsdatei des Flight-Services. Durch den Konfigurationsparameter `eureka.instance.preferIpAddress: true` wird das Service anhand der IP-Adresse und nicht mittels des Hostnamen beim Eureka-Service registriert. Dies wäre laut Carnell vor allem bei Container-Umgebungen wie Docker unbedingt notwendig, da Docker-Container zufallsgenerierte Hostnamen besitzen und keine DNS Einträge vorhanden wären (vgl. Carnell 2017, S.108).

Für die eigentliche Inter-Service-Kommunikation über Eureka wurden in der Prototypen-Applikation zwei Ansätze mit unterschiedlichem Abstraktionsniveau evaluiert. Zum einen der Spring Discovery Client und Netflix Feign (siehe Kapitel 2.2.6).

```
public CountryResource getCountryByCountryCode(String code) {
        RestTemplate restTemplate = new RestTemplate();
```

```
3         List<ServiceInstance> instances = discoveryClient.
              getInstances(VISHNU_FLIGHT_SERVICE);
4         if (instances == null) {
5                 logger.error("No instance of {0} is reachable!",
                      VISHNU_FLIGHT_SERVICE);
6                 throw new ServiceUnreachableException("No " +
                      VISHNU_FLIGHT_SERVICE + "is currently available")
                      ;
7         }
8         ResponseEntity<CountryResource> restExchange = restTemplate.
              getForEntity(
9         instances.get(0).getUri().toString() + "/v1/country?code=" +
               code, CountryResource.class);
10
11        return restExchange.getBody();
12 }
```

Listing 3.8: Inter-Service-Kommunikation über Eureka mittels dem Spring Discovery Client

Ein entscheidender Nachteil des Spring Discovery Client ist, dass das von Ribbon angebotene clientseitige Load-Balancing nicht genutzt wird. Wie Listing 3.8 zeigt liefert die Methode `discoveryClient.getInstances(VISHNU_FLIGHT_SERVICE)` eine Liste aller bei Eureka registrierter Instanzen des Flight-Service zurück. Der Entwickler, die Entwicklerin muss also selbst entscheiden mit welcher Instanz des Services kommuniziert werden soll (vgl. Carnell 2017, S. 114). Aus diesem Grund wurde in der Prototyp-Applikation die Programmbibliothek Netflix Feign eingesetzt. Diese bietet ein wesentlich höheres Abstraktionsniveau als der Spring Discovery Client und nutzt gleichzeitig auch das clientseitige Load-Balacing von Ribbon.

Um Netflix Feign Nutzen zu können, muss die Kompilierzeit-Abhängigkeit `org.springframework.cloud:spring-cloud-starter-feign` vorhanden sein und die Bootstrap-Klasse mit der Annotation `@EnableFeignClients` annotiert werden.

```
1 @FeignClient("vishnu-flight-service")
2 public interface FlightServiceFeignClient {
3     @RequestMapping(
4             method = GET,
5             value = "/v1/country",
6             params = "code"
7     )
8     CountryResource getCountryByCountryCode(@RequestParam(name = "
          code") String countryCode);
9     ...
```

Listing 3.9: Interservice-Kommunikation über Eureka mittels Netflix Feign

Listing 3.9 verdeutlicht wie Feign eingesetzt wird. In einem Java Interface wird definiert welche REST-Endpunkte der Flight-Service anderen Services zur Verfügung stellt. Die Implementierung des Interfaces wird zur Kompilierzeit der Anwendung von Feign generiert. Diese kann mittels Dependency Injection injiziert werden. So können definierten Methoden des Interfaces genutzt werden um die jeweiligen Ressourcen abzufragen. Feign bietet eine hohes Abstraktionniveau und eine einfache Schnittstelle für die Service-Kommunikation. Jedoch sollte darauf geachtet werden, dass dieses Interface wie eine öffentliche API anzusehen ist und Änderungen dementsprechend schwierig vorzunehmen sind. Breaking-Changes an den REST-Schnittstellen sollten unbedingt vermieden werden, da ansonsten ebenfalls die Feign Interfaces dementsprechend angepasst werden müssen.

3.6 Service Gateway mit Netflix Zuul

Das Service Gateway übernimmt innerhalb der Architektur des Prototypen die Funktion eines Intermediäres zwischen einem Service-Client und den konkreten Microservices. Ein Service-Client könnte beispielsweise ein Benutzer, eine Benutzerin der Applikation oder eine beliebige Front-End Technologie sein. Das Service-Gateway kann als Fassade für das eigentliche, aus Microservices bestehende, System angesehen werden. Es wird verhindert, dass ein Service-Client die Services direkt anspricht und sämtliche Kommunikation mit dem System findet über die vom Service-Gateway verwaltete URL statt. Durch diese privilegierte Stellung innerhalb des Systems eignet sich das Service-Gateway als zentrale Sammelstelle für quer-schneidende Belange (engl. Cross-Cutting-Concerns). Zu den klassischen von Robert C. Martin definierten Cross-Cutting-Concerns zählen beispielsweise Persistenz, Transaktionen, Sicherheit und Ausfallsicherheit (vgl. Martin 2009, S. 202). Diese Definition gilt zwar für monolithische Applikationen und einzelne Microservices, allerdings bringt eine Microservice-Architektur weitere Cross-Cutting-Concerns zum Vorschein. Carnell nennt folgende Cross-Cutting-Concerns die innerhalb eines Service-Gateways implementiert werden können:

Statisches Routing Das Sammeln aller von den Microservices bereitgestellten Routen in einer gemeinsamen URL und das Transferieren der Anfragen zum zuständigen

Microservice.

Dynamisches Routing Inspizieren von Anfragen und das intelligente Weiterleiten der Anfrage an bestimmte Services aufgrund der in der Anfrage enthaltenen Parameter oder Informationen.

Authentifizierung und Autorisierung Überprüfung ob ein Service-Client die notwendigen Rechte besitzt, um die Anfrage gegen das System zu richten.

Sammeln von Metriken und Logging Da alle Anfragen durch das Service-Gateway durchgeschleust werden, können wertvolle Metriken wie die Antwortzeit einzelner Services gesammelt werden.

(vgl. Carnell 2017, S. 155ff.)

Es ist praktikabel diese Cross-Cutting-Concerns im Service-Gateway zu implementieren, allerdings muss darauf geachtet werden, dass das Service-Gateway, wie alle anderen Services, skalierbar bleibt und mehrere Instanzen hochgefahren werden können um einen Single-Point of Failure zu vermeiden. Carnell empfiehlt aus diesem Grund, dass die im Service-Gateway implementierte Logik zustandslos und leichtgewichtig sein sollte um die Notwendigkeit Information auf weitere Instanzen des Service-Gateways replizieren zu müssen auszuschließen. Sobald eine Replikation der im flüchtigen Speicher gehaltenen Daten notwendig sei, wäre die Skalierbarkeit des Service-Gateways massiv eingeschränkt (vgl. Carnell 2017, S. 156).

In der Prototyp-Applikation wurde der Netflix Zuul Reverse-Proxy Server als Service-Gateway eingesetzt. Dieser übernimmt die Aufgaben des statischen und dynamischen Routings. Da Zuul nahtlos in eine Spring Boot Anwendung integriert werden kann, stellt dieser auch die initiale HTML Ressource des Systems inklusive des Navigationselements zur Verfügung. Wird Zuul bei Eureka registriert, stellt er automatisch für jeden bei Eureka registrierten Service die entsprechende URL auf Basis des Namens zur Verfügung und nutzt gleichzeitig das clientseitige Load-Balancing von Ribbon um die Last zu verteilen. Da Eureka und Zuul kontinuierlich Information austauschen können neue Services jederzeit bei Eureka registriert werden und Zuul stellt die neuen Routen automatisch zur Verfügung ohne dass ein Neustart erforderlich ist. Weiterhin besteht auch die Möglichkeit Routen statisch zu definieren. So können auch Services welche nicht bei Eureka registriert sind, von Zuul verwaltet werden.

Zur Integration von Zuul muss die Abhängigkeit auf `org.springframework.cloud:spring-cloud-starter-zuul` gegeben sein und die Bootstrap-Klasse wurde mit `@EnableZuulProxy` annotiert werden. Um Zuul bei Eureka zu registrieren müssen die Eureka-spezifischen Konfigurationsparameter entsprechend Listing 3.7 gesetzt werden. Weitere Konfiguration oder Programmcode ist nicht erforderlich, um Zuul nutzen zu können.

3.7 Tracing und verteiltes Logging

Wie auch monolithische Anwendungen sind Microservice-Systeme nicht vor Fehlern und Performance-Problemen gefeit. In dieser Hinsicht zeigt sich einer der Nachteile von Microservice-Architekturen: Die Komplexität von verteilen Systemen. Während bei monolithischen Anwendungen das Logging meist als triviale Aufgabe angesehen wird, ist das Logging und Tracing bei Microservices wesentlich aufwändiger. In herkömmlichen Architekturen werden Log-Einträge meist auf das Dateisystem geschrieben oder in eine Datenbank persistiert. Diese weitverbreitete Vorgehensweise ist bei Microservices nicht praktikabel. Oft besteht ein Microservice-System aus eine großen Anzahl von Services und es werden hunderte Instanzen dieser Services, je nach Auslastung des Systems, hochgefahren und gegebenenfalls wieder gestoppt. Da diese Instanzen meist innerhalb Docker-Container oder virtueller Maschinen laufen, würden Log-Einträge, die in das Dateisystem geschrieben wurden beim Stoppen einer Instanz verloren gehen. Nachdem Microservices per Definition (siehe Kapitel 3.2) auch keine gemeinsame Datenbank nutzen sollten, ist auch die Persistierung in eine Datenbank ineffizient und widerspricht dem Grundgedanken einer Microservice-Architektur.

Ein möglicher Lösungsansatz dem Problem entgegen zu wirken, ist die Nutzung eines zentralisierten Logging-Servers, auch Log-Aggregator genannt. Die Microservices senden ihre Log-Einträge direkt über das Netzwerk zum Log-Server, wo die Einträge verarbeitet werden. Als Übertragungsprotokolle können gängige Protokolle wie TCP, HTTP, syslog oder auch spezielle Protokolle wie Graylog Extended Log Format (GELF) eingesetzt werden (vgl. Wolff 2017).

Gängige Log-Aggregatoren Lösungen sind Graylog oder der ELK-Stack (Elasticsearch, Logstash und Kibana), welche als Infrastuktur-Service in die Architektur eingebunden werden können. Weitere Optionen sind cloud-basierende Services die nicht in das bestehende System als Service eingebunden werden müssen und vom Anbieter direkt verwaltet werden. Der Anbieter stellt eine Schnittstelle zur Verfügung an, über die

Log-Daten übertragen werden können. Ein Beispiel hierfür ist Papertrail. Alle Lösungen bieten die Möglichkeit Log-Einträge zu suchen, filtern und benutzerdefinierte Dashboards anzulegen. Dies ermöglicht es den unterschiedlichen Stakeholdern des Loggings eigene Übersichtsseiten, mit den für sie relevanten Informationen anzulegen.

Um einen Request eines Benutzers, einer Benutzerin zu bedienen, arbeiten Microservices oft zusammen. So wird die Anfrage meist nicht nur von einem Service bearbeitet, sondern es können durchaus mehrere Services beteiligt sein. Es entsteht ein Weg durch das System. Dieser Weg eines Requests durch die beteiligten Microservices muss nachvollziehbar sein, um analysefähige Log-Einträge zu erhalten und etwaige Performance-Engpässe zu identifizieren. Die Verfolgung eines Requests durch das System wird auch Tracing genannt.

Zur eindeutigen Identifizierung wird jeder Request mit einer Korrelations-ID versehen. Diese ID identifiziert einen Request an das Gesamtsystem und wird von jedem beteiligten Microservice an den Nächsten, für die Verarbeitung beteiligten Microservice, weitergegeben. Da es sich um einen Cross-Cutting-Concern handelt, bestünde eine Möglichkeit die Korrelations-ID durch einen HTTP-Interceptor im Zuul Service-Gateway an den initialen Request zu haften. Jeder Microservice müsste eine ähnliche Logik implementieren, damit die Korrelations-ID auch bei der Inter-Service-Kommunikation vorhanden ist (vgl. Wolff 2017, S.245).

Spring bietet mit Spring Cloud Sleuth hierfür eine einfachere Lösung, welche auch im Prototyp eingesetzt wurde. Sleuth generiert und injiziert die Korrelations-ID automatisch in jeden Request, auch in interne Requests zwischen den Services und in Spring MDC. Somit ist die Korrelations-ID in jedem Log-Eintrag enthalten. Damit die Microservices Sleuth nutzen können muss lediglich eine Kompilierzeit-Abhängigkeit auf das Spring Cloud Sleuth Projekt gegeben sein.

Die On-Premise Logging-Aggregator Lösungen, wie der ELK-Stack erfordern einen hohen Konfigurations- und Ressourcen Aufwand sowie die Einführung von mindestens einem weiteren Infrastruktur-Service inklusive Datenquelle. Aus diesem Grund wurde im Prototyp die cloud-basierende Log-Lösung Papertrail genutzt.

Papertrail bietet als Schnittstelle für die Log-Einträge eine syslog TCP Schnittstelle an. Diese kann von den Java Logging-Frameworks “log4j” und “logback” mittels eines speziellen syslog-tcp Appenders einfach bedient werden. Spring nutzt standardmäßig das Logging-Framework “logback”. Um die Log-Einträge an Papertrail zu senden wurde die in Listing 3.10 gezeigte `logback.xml` Datei in den Klassenpfad von jedem Infrastruktur- und Business-Service abgelegt. Diese Konfigurationsdatei nutzt den von

logback inkludieren `SyslogAppender`, um die Log-Einträge an Papertrail zu übermitteln.

```
<?xml version="1.0" encoding="UTF-8"?>
        <configuration>
                <include resource="org/springframework/boot/logging/
                   logback/base.xml"/>
                <appender name="SYSLOG-PAPERTRAIL" class="ch.qos.
                   logback.classic.net.SyslogAppender">
                     <syslogHost>logs6.papertrailapp.com</
                        syslogHost>
                     <port>37423</port>
                     <facility>USER</facility>
                </appender>

        <root level="INFO">
                <appender-ref ref="CONSOLE" />
                <appender-ref ref="SYSLOG-PAPERTRAIL" />
        </root>
</configuration>
```

Listing 3.10: logback Konfiguration

Zur Visualisierung des Zeitverhaltens wurde in der Prototyp-Applikation Netflix Zipkin (siehe 2.2.8) eingesetzt. Zipkin visualisiert die benötigte Verarbeitungszeit von jedem an der Verarbeitung des Requests beteiligten Microservices. Dadurch kann das Performance-Verhalten von jedem Services ermittelt werden.

Um eine Spring Boot Applikation zu einem Zipkin-Server aufzurüsten, wird lediglich die Abhängigkeiten auf `io.zipkin.java:zipkin-server`, `io.zipkin.java:zipkin-autoconfigure-ui` und die Annotation `@EnableZipkinServer` auf der Bootstrap-Klasse benötigt. Standardmäßig nutzt Zipkin eine In-Memory-Datenbank zur Persisitierung der erhaltenen Daten. Allerdings werden auch weitere Datenbank-Systeme wie MySQL, Cassandra oder Elasticsearch unterstützt. Im Prototyp wurde die In-Memory-Variante gewählt.

```
spring:
        application:
                name: vishnu-flight-service
zipkin:
        baseUrl: http://zipkin
sleuth:
        sampler:
                percentage: 1.0
```

Listing 3.11: Konfigurationsdatei des Flight-Service

Damit Zipkin die Daten, von den Sleuth nutzenden, Microservices erhalten kann, mussten die Konfigurationsdateien der Services wie in Listing 3.11 angepasst werden. Da auch Zipkin nahtlos mit Eureka integriert werden kann, reicht es die Service-ID, in diesem Fall `zipkin`, als `baseUrl` anzugeben. Der Konfigurationsparameter `sleuth.sampler.percentage` definiert den Prozentsatz der an Zipkin gesendeten Transaktionsdaten an. Wird der Parameter nicht gesetzt, werden lediglich 10 Prozent der Transaktionen an Zipkin gesandt. Im Falle des Prototypen wurden die Daten aller Transaktionen zwischen den Microservices an Zipkin weitergereicht.

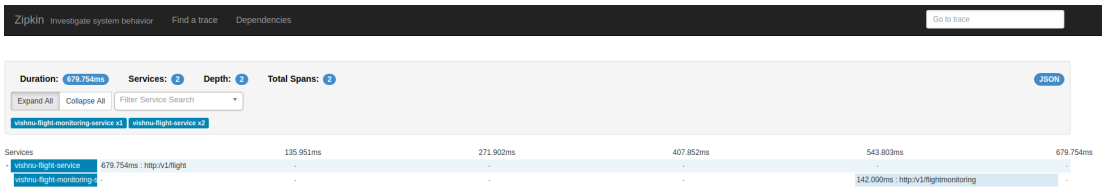

Abbildung 3.5: Tracing eines Requests mit Zipkin

Wird nun ein Request gegen das System getätigt, werden auf dem Zipkin Web-GUI die Tracing-Details angezeigt. Die Abbildung 3.5 zeigt das von Zipkin dargestellte Tracing bei der Erstellung eines neuen Fluges durch den Flight-Service. Wurde der Flug vom Flight-Service erstellt, sendet dieser den eben erstellen Flug an den Flight-Monitoring-Service. Wie auf der Grafik ersichtlich betrugen die Verarbeitungszeiten 142.0 Millisekunden ms am Flight-Service und 679.754 Millisekunden am Flight-Monitoring-Service.

3.8 Widerstandsfähigkeit und Stabilität

Komplexe Systeme sind nicht vor Fehlern und Ausfällen gefeit. Wolff attestiert verteilen Systemen sogar eine wesentlich höhere Anfälligkeit als anderen Software-Architekturen: Netzwerke und Server seien unzuverlässig und da Microservices oft verteilt auf vielen Server laufen würden, wäre die Ausfallquote dementsprechend höher als bei monolithischen Anwendungen (vgl. Wolff 2017, S. 203).

Durch die höher Ausfallgefahr ist es umso wichtiger, dass der Ausfall eines Services keinen Dominoeffekt zur Folge hat und das gesamte System ausfällt. Jeder Service

muss gegen den Ausfall der anderen Services abgesichert werden. Der Totalausfall eines Services stellt nicht den einzigen problematischen Punkt dar, auch eine kurzzeitig schwache Performance der Services oder des Netzwerks muss ausgeglichen werden. Um die Stabilität von verteilten Systemen zur erhöhen nennt Nygard einige Stabilitätsmuster wie Time-out, Circuit Breaker, Bulkhead und Fail Fast (vgl. Nygard 2007, S. 110ff.). Diese Muster werden häufig in Microservice-Architekturen eingesetzt. Da die Implementierung dieser Muster ein hohes Verständnis von Multi-Threading und Thread-Management benötigt, rät Carnell davon ab, diese selbst zu implementieren. Netflix stellt mit Hystrix eine Programmbibliothek zur Verfügung, die bereits getestete und eingesetzte Implementierungen der Circuit Breaker, Fallback und Bulkhead-Muster enthält. Hystrix kommt, wie viele andere bereits genannte Bibliotheken, täglich im Microservice-System der populären Streaming-Plattform Netflix zum Einsatz (vgl. Carnell 2017, S. 126ff.). Hystrix wurde auch in der Prototyp-Applikation eingesetzt. Folgend werden zwei eingesetzten Resilience Muster Circuit Breaker und Fallback und deren Einsatz erläutert.

Circuit Breaker Der Terminus Circuit Breaker (Leistungsschutzschalter) ist eigentlich der Elektrotechnik zuzuordnen. Ein Circuit Breaker in der Elektrotechnik hat die Aufgabe bei Fehlern, resultierend aus Überlast oder Kurzschlüssen, den Stromkreis zu Unterbrechen um das Gesamtsystem zu schützen. Ein Software Circuit Breaker hat eine ähnliche Funktionsweise: Ist ein System nicht erreichbar, greift der Circuit Breaker ein und unterbindet weitere Anfragen an das System. Diese Funktionsweise kann mit einem Time-out kombiniert werden. Dauert beispielsweise ein Datenbankzugriff zu lange, greift der Circuit Breaker ein und verhindert ein Blockieren des Systems.

```
1  @HystrixCommand(
2          commandProperties = {@HystrixProperty(
3          name="execution.isolation.thread.timeoutInMilliseconds",
4          value = "500")})
5  private RawFlightResource setFlightLanded(RawFlightResource
       rawFlightResource) {
6          //update the flight status and send it to the flight service
7          RawFlightResource resource = new RawFlightResource(
               rawFlightResource);
8          resource.setStatus("landed");
9          flightServiceFeignClient.updateResource(resource);
10
11         return resource;
12 }
```

Listing 3.12: Einsatz eines Hystrix Circuit Breakers mit Timeout

Listing 3.12 zeigt den Einsatz eines Hystrix Circuit Breakers. Ein Hystrix Circuit Breaker wird mittels der Annotation `@HystrixCommand` auf einer Methode aktiviert. Spring erkennt diese Annotation und generiert dynamisch einen Proxy, der die Zugriffe der Methode über einen Thread-Pool steuert.

Die Methode `setFlightLanded()` des Flight-Monitoring-Services besitzt die Aufgabe einen Flug als gelandet zu markieren, sofern dieser den Zielflughafen erreicht hat. In weiterer Folge wird das Flug-Objekt mittels Feign (siehe Kapitel 3.5) an den Flight-Service zu übermittelt. Ist der Flight-Service allerdings nicht erreichbar, oder antwortet nicht innerhalb des festgelegten Timeouts von 500 Millisekunden, greift der Hystrix Circuit Breaker ein und löst eine `com.nextflix.hystrix.exception.HystrixRuntimeException` aus. So wird ein langes Blockieren durch den Remote-Aufruf `flightServiceFeignClient.updateResource(resource)` verhindert.

Fallback Neben dem Timeout lässt sich das Circuit Breaker-Muster auch mit einem Fallback kombinieren. Von einem Fallback (Ersatzfunktion) spricht man, wenn im Falle eines Fehlers damit auf eine Alternative zurückgegriffen werden kann. So bietet Hystrix die Option anstatt der `HystrixRuntimeException` eine Fallback-Methode im Falle eines Fehlers zu definieren.

```
1  @HystrixCommand(
2          commandProperties = {@HystrixProperty(
3                  name="execution.isolation.thread.
                       timeoutInMilliseconds",
4                  value = "500"
5          )},
6          fallbackMethod = "scheduleForLaterTransmission")
7  private RawFlightResource setFlightLanded(RawFlightResource
       rawFlightResource) {
8          ...
9  }
10
11 private void scheduleForLaterTransmission(RawFlightResource
       rawFlightResource) {
12         this.failedRemoteCalls.add(rawFlightResource);
13 }
```

Listing 3.13: Einsatz eines Hystrix Circuit Breakers mit Timeout und Timeout

Listing 3.13 zeigt den Einsatz einer Fallback Methode. Anstatt der `HystrixRuntimeException` wird die Methode `scheduleForLaterTransmission()` aufgerufen. Die Methoden fügt die Ressource, die nicht Übertragen werden konnte, zu einer Liste hinzu, ändert

ansonsten aber nichts am Zustand der Ressource. Die Objekte innerhalb dieser Liste werden in einem definierten Zeitintervall erneut an den Flight-Service gesendet um den korrekten Status des Fluges zu übertragen. Die Fallback Methode muss innerhalb derselben Klasse definiert sein, und die Signatur muss mit der `@HystrixCommand` annotierten Methode identisch sein.

Hystrix bietet noch weitere Konfigurationsmöglichkeiten hinsichtlich der Thread-Verwaltung und des Thread-Pools. Diese Konfigurationen waren in der Prototyp-Applikation allerdings nicht notwendig und werden aus diesem Grund nicht weiter erläutert. Es zeigt sich, dass durch die Nutzung von Hystrix die Robustheit und Stabilität einzelner Services und in Folge des Gesamtsystems erhöht werden kann.

3.9 Auslieferung

Die Auslieferung und der Betrieb (engl. Deployment and Operation), so Newman 2015 einer monolithischen Applikation sei wesentlich einfacher als bei einer Microservice-Applikation (vgl. Newman 2015, S. 188). Im Gegensatz zu einer monolithischen Applikation liegen viele Artefakte vor, die überwacht und ausgeliefert werden müssen. Ein wichtiger Aspekt ist, dass jedes Artefakt, also jeder Service, unabhängig von den anderen Services ausgeliefert werden kann (siehe Kapitel 3.2). Dieser Prozess der Auslieferung müsse laut Carnell folgenden Kriterien erfüllen:

Automatisiert Es ist keinerlei menschliche Intervention in den Auslieferungs- und Versorgungsprozess (engl. Deployment and Provisioning) notwendig. Gemäß des Continuous-Delivery Auslieferungsprozesses sollte ein Service nach jeder Quellcode-Änderung, die in das Versionsverwaltungssystem eingespielt wurde, neu ausgeliefert und in Produktion gebracht werden.

Wiederholbar Der Build- und Auslieferungsprozess sollte vorhersehbar und uneingeschränkt wiederholbar sein.

Komplett Das auszuliefernde Artefakt (engl. Deployment artifact) muss vollständig im Sinne der Ausführbarkeit sein. Das bedeutet, dass das Artefakt alle zur Ausführung notwendigen Laufzeitumgebungen beinhalten sollte. Dazu zählen auch etwaige Datenbanken, die Konfigurationsparameter für cloud-basierende Datenquellen und Konfigu-

rationsdetails. So könnte das Artefakt beispielsweise eine komplette virtuelle Maschine oder ein Docker-Image sein.

Unveränderbar Wurde das Deployment-Artefakt erstellt darf es keinerlei Gründe geben das Artefakt oder die Laufzeitkonfiguration zu ändern. Sind Änderungen notwendig, müssen diese an den Skripten, dem Quellcode oder Konfigurationsdateien im Versionsverwaltungssystem vorgenommen werden.

Eine robuste und generalisierte Build-Pipeline sei laut Carnell unverzichtbar und erfordere bei zunehmender Komplexität oftmals den Einsatz eines spezialisierten Teams, genannt DevOps, dessen Aufgabe darin liege den Build- und Auslieferungsprozess zu optimieren (vgl. Carnell 2017, S.289ff.).

Im Vergleich dazu sind die ausgelieferten Artefakte bei monolithischen Applikationen oft nicht als komplett im Sinne der von Carnell definierten, Kriterien anzusehen. Häufig werden im Java-Umfeld lediglich EAR oder WAR Artefakte ausgeliefert, welche dann erst auf einem Applikationsserver installiert werden müssen. Ferner ist es oftmals notwendig, dass Konfigurationsdateien nach der Auslieferung angepasst werden müssen. Beispielsweise müssen Datenbank-Verbindungs-Parameter oder andere Parameter in der Konfiguration des Applikationsserver angepasst werden.

Um den Auslieferungsprozess der Prototyp-Applikation möglichst realitätsnah zu gestalten und die vier, von Carnell definierten Kriterien zu erfüllen wurde Git als Versionskontrollsystem, CircleCI (siehe 2.2.12) als Continuous Integration System genutzt. Die auszuliefernden Artefakte sind im Falle des Prototypen Docker-Images die nach erfolgreichen Abschließen des Build-Prozesses durch CircleCI auf die Docker Registry Docker-Hub hochgeladen werden. Diese könnten dann in weiterer Folge automatisiert in eine Cloud-Lösung wie Amazon AWS oder Cloud-Foundry ausgeliefert und gestartet werden.

Zur Erstellung der Docker-Images wurde ein Gradle Docker-Plugin genutzt. Dieses erlaubt es Docker-Images mittels Gradle zu erstellen und auf einer Docker-Registry wie Docker-Hub zu veröffentlichen (vgl. Transmode Systems AB 2017). Durch das Plugin entfällt die Notwendigkeit Dockerfiles manuell für jeden Service zu erstellen, da diese durch das Plugin generiert werden. Es kann auch ein Basis-Image, welches als Grundlage für das selbst erstellte Image dient, definiert werden. Im Prototyp wurde für Services ohne eigene Datenbank das Image `frolvlad/alpine-oraclejdk8:slim` als Basis-Image herangezogen. Dieses enthält bereits eine leichtgewichtige Linux Umgebung mit dem Oracle JDK 8 und erlaubt dadurch das Ausführen von JAR-Dateien.

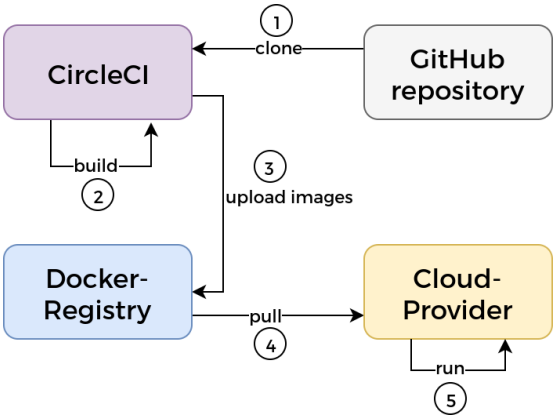

Abbildung 3.6: Grafische Darstellung des Auslieferungsprozess der Prototyp-Applikation

Services die eine Datenbank benötigen, wie beispielsweise der Flight-Service, nutzen als Basis-Image ein Image, welches bereits die MongoDB Datenbank enthält.

```
1  task buildDocker(type: Docker, dependsOn: build) {
2          volume("/tmp")
3          addFile("$buildDir/libs/${project.name}-${project.version}.
               jar", "app.jar")
4          runCommand("touch /app.jar")
5          entryPoint(["java", "-jar", "/app.jar"])
6          push = true
7          tag = "bnjm/vishnu"
8          tagVersion = "${project.name}-${project.version}"
9          doFirst {
10                 copy {
11                         from jar
12                         into stageDir
13                 }
14         }
15 }
```

Listing 3.14: Gradle Task für das Deployment des Spring Cloud Config Servers

Listing 3.14 zeigt den Gradle Docker-Task des Spring Config-Servers. Dieser Gradle Task erstellt das Docker-Image des Spring Cloud Config-Servers und veröffentlicht das Image auf Docker-Hub. Durch den Befehl `addFile()` wird die durch den Gradle Build-Prozess erstellte JAR-Datei dem Docker Container hinzugefügt. Der Befehl `entryPoint()` bewirkt, dass die JAR-Datei nach dem Hochfahren des Containers gestartet wird. Nach der Veröffentlichung auf Docker-Hub könnten die Docker Images in eine Cloud oder auf einem beliebigen Server ausgeliefert und in einem Container gestartet werden.

Kapitel 4

Resumee und Ausblick

In dieser Arbeit wurde untersucht ob, und wie sich eine Microservice-Architektur mit dem Spring-Framework und unterstützenden Technologien, umsetzen lässt. Die Arbeit zeigt dass, das Spring-Framework in Verbindung mit den Netflix-Bibliotheken eine solide Wahl zur Umsetzung einer Microservice-Architektur darstellt. Durch die automatische Konfiguration und der nahtlosen Integration der Netflix-Bibliotheken wird der Konfigurationsaufwand der notwendigen Infrastruktur-Services stark reduziert. Es werden bereits vorgefertigte Lösungen für bekannte Herausforderungen der Microservice-Architektur wie beispielsweise die Service-Discovery zur Verfügung gestellt. Diese Lösungen erlauben eine Reduzierung der Komplexität und vereinfachen die Betreibung einer Microservice-Anwendung. Ferner werden sowohl das Spring-Framework als auch die Netflix-Bibliotheken bereits weitläufig in Produktionsumgebungen betrieben. Somit ist eine stetige Weiterentwicklung beider Technologien sehr wahrscheinlich. Dies erhöht wiederum die Zukunftssicherheit und das Potential dieser Technologien.

Zugleich lässt die Arbeit, die dennoch hohe Komplexität von verteilten Anwendungen und der notwendigen Infrastruktur im Vergleich zu monolithischen Anwendungen erkennen. Es ist eine hohe Anzahl von Infrastruktur-Diensten notwendig, um solch ein System effizient zu betreiben. Diese Dienste müssen ebenfalls gewartet, aktualisiert, überwacht und ausgeliefert werden. Allerdings generieren sie keinen unmittelbaren Geschäftswert und können bei neuen Projekten die Time-to-Market erhöhen. Demnach zeigt sich, dass Microservices eher als eine Lösung für spezielle Anforderungen und Probleme angesehen werden sollten. Wird eine hohe Skalierbarkeit oder eine Auslieferbarkeit einzelner Services benötigt, kann eine Microservice-Architektur einen unmittelbaren Mehrwert generieren.

Ein Thema künftiger Forschung könnte die Transformation einer bestehenden monolithischen Applikation unter Rücksichtnahme der in dieser Arbeit beschriebenen Techniken und Technologien zu einer Microservice-Applikation sein. Auch könnte der für diese Arbeit erstelle Prototyp als monolithische Applikation umgesetzt werden, um die beiden Architektur-Muster eingängiger vergleichen zu können. Die Microservice-Architektur ist als relativ neue Strömung der Software-Architekturen noch nicht gänzlich erforscht und es sind bei weitem nicht alle Möglichkeiten des Potentials dieser oder darauf aufbauender Architekturen ausgeschöpft.

Abkürzungen

API Application Programming Interface

CQS Command-Query Segregation

CPU Central Processing Unit

CQS Command Query Separation

CRUD Create Update Read Delete

CSV Comma-separated Values

DDD Domain-driven Design

EAR Enterprise Application Archive

ESB Enterprise Service Bus

GELF Graylog Extended Log Format

JPA Java Persistence API

JVM Java Virtual Maschine

HATEOAS Hypermedia As The Engine Of Application State

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JAR Java Archive

JSON JavaScript Object Notation

JRE Java Runtime Environment

MDC Mapped Diagnostic Context

REST Representational State Transfer

RPC Remote Procedure Call

SOA Service-oriented Architecture

SSE Server-sent Events

TCP Transmissition Control Protocol

UI User Interface

URL Uniform Resource Locator

WAR Web Application Resource

YAML YAML Ain't Markup Language

XML Extensible Markup Language

Referenzen

Bakshi, Kapil (2017). „Microservices-based software architecture and approaches“. In: *Proceedings of the Aerospace Conference, 2017 IEEE*. (4.–11. März 2017). Big Sky, MT, USA: IEEE.

Bondi, André B. (2000). „Characteristics of scalability and their impact on performance“. In: *WOSP '00 Proceedings of the 2nd international workshop on Software and performance*. (17.–20. Sep. 2000). Ottawa, Canada: ACM, S. 195–203. DOI: 10.1145/350391.350432.

Carnell, John (2017). *Spring Microservices in Action*. 1. Aufl. Shelter Island: Manning Publications. ISBN: 978-1617293986.

CircleCI (2017). *CircleCI Language Guide: Java*. Available from: <https://circleci.com/docs/2.0/language-java/> [2. Juni 2017].

Cole, Adrian, Spencer Gibb, Marcin Grzejszczak und Dave Syer (2017). *Spring Cloud Sleuth*. Available from: <http://cloud.spring.io/spring-cloud-sleuth/single/spring-cloud-sleuth.html> [13. Aug. 2017].

Conway, Melvin E. (1968). „How do committees invent?“ In: *Datamation* April 1968.

Fowler, Martin (2015). *MonolithFirst*. Available from: <http://cloud.spring.io/spring-cloud-config/single/spring-cloud-config.html> [7. Juni 2017].

Fowler, Martin und James Lewis (2014). *Microservices - a definition of this new architectural term*. Available from: <https://martinfowler.com/articles/microservices.html> [14. Juli 2017].

Google Inc. (2017). *Protocol Buffers - Developer Guide*. Available from: <https://developers.google.com/protocol-buffers/docs/overview> [20. Aug. 2017].

Jaramillo, David, Duy Nguyen und Robert Smart (2016). „Leveraging microservices architecture by using Docker technology“. In: *Proceedings of the SoutheastCon,*

2016. (30. März–3. Apr. 2016). Norfolk, VA, USA: IEEE, S. 1–5. DOI: 10.1109/SECON.2016.7506647.

Martin, Robert C. (2009). *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code - Deutsche Ausgabe*. 1. Aufl. Heidelberg: MITP-Verlags GmbH Co. KG. ISBN: 978-3-826-69638-1.

Muschko, Benjamin (2014). *Gradle in Action*. 1. Aufl. Shelter Island: Manning Publications. ISBN: 978-1617291302.

Newman, Sam (2015). *Building Microservices: Designing Fine-Grained Systems*. 1. Aufl. Sebastopol: O'Reilly Media Inc. ISBN: 978-1491950357.

Nickoloff, Jeff und Ahmet Alp Balkan (2016). *Docker in Action*. 1. Aufl. Shelter Island: Manning Publications. ISBN: 978-1-63343-023-5.

Nygard, Michael (2007). *Release it! - Design and Deploy Production-Ready Software*. 1. Aufl. Raleigh, North Carolina: The Pragmatic Bookshelf. ISBN: 978-0-9787392-1-8.

Pivotal Inc. (2017a). *Router and Filter: Zuul*. Available from: <http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_router_and_filter_zuul> [15. Juni 2017].

– (2017b). *Spring Cloud Config Reference Guide*. Available from: <http://cloud.spring.io/spring-cloud-config/single/spring-cloud-config.html> [7. Juni 2017].

– (2017c). *Spring Cloud Netflix Reference Guide 1.4.0.BUILD-SNAPSHOT*. Available from: <http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#d0e9> [15. Juni 2017].

– (2017d). *Spring Cloud Netflix Reference Guide - Service Discovery: Eureka Server*. Available from: <http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#spring-cloud-eureka-server> [5. Juni 2017].

Richardson, Chris (2017). *Microservice Patterns*. Manning Early Access Progam Version 4. Shelter Island: Manning Publications. ISBN: 978-1617294549.

Richardson, Leonard und Amundsen Mike (2013). *RESTful Web APIs: Services for a Changing World*. 1. Aufl. Sebastopol: O'Reilly Media Inc. ISBN: 978-1449358068.

Spichale, Kai (2017). *API Design*. 1. Aufl. Heidelberg: dpunkt.verlag GmbH. ISBN: 978-3-86490-387-8.

Transmode Systems AB (2017). *Gradle Docker plugin*. Available from: <https://github.com/Transmode/gradle-docker> [4. Aug. 2017].

Villamizar, Mario, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas und Santiago Gil (2015). „Evaluating the monolithic and the micro-

service architecture pattern to deploy web applications in the cloud“. In: *Proceedings of the Computing Colombian Conference (10CCC), 2015 10th*. (21.–25. Sep. 2015). Bogota, Colombia: IEEE, S. 583–590.

Walls, Craig (2014). *Spring in Action*. 4. Aufl. Shelter Island: Manning Publications. ISBN: 978-1617291203.

– (2016). *Spring Boot in Action*. 1. Aufl. Shelter Island: Manning Publications. ISBN: 978-1617292545.

Webb, Phillip, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons und Vedran Pavić (2017). *Spring Boot Reference Guide*. Available from: <http://docs.spring.io/spring-boot/docs/2.0.0.M3/reference/htmlsingle/> [12. Aug. 2017].

Webber, Jim, Savas Parastatidis und Ian Robinson (2010). *REST in Practice: Hypermedia and Systems Architecture*. 1. Aufl. Sebastopol: O'Reilly Media Inc. ISBN: 978-0596805821.

Wolff, Eberhard (2016). „Erfahrungen, Vor- und Nachteile von Microservices - Schein und Sein“. In: *Javamagazin* Mai 2016.

– (2017). *Microservices: Flexible Software Architecture*. 1. Aufl. Boston: Addison-Wesley. ISBN: 978-0-134-60241-7.

Zhongxiang, Xiao, Wijegunaratne Inji und Qiang Xinjian (2016). „Reflections on SOA and Microservices“. In: *Proceedings of the 2016 4th International Conference on Enterprise Systems (ES)*. (2.–3. Nov. 2016). Melbourne, VIC, Australia: IEEE, S. 60–67.