Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelorarbeit

Studienrichtung

Informatik

Oleksiy Reznytskyy

Cloud Native Application Development - Evaluierung von Java EE und Spring Cloud

Prüfer: Prof. Dr. Gerhard Meixner

Abgabe der Arbeit am: 08.11.2018

Betreuer der Firma ARS Computer und Consulting:

Herr Michael Heiß

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Bachelorarbeit:
Droste-Hülshoff-Str. 30
86157 Augsburg
Telefon:+49 177 3634749
oleksiy.reznytskyy@hs-augburg.de

Hochschule
Augsburg University of
Applied Sciences

Erklärung zur Abschlussarbeit

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

_____ _____

Ort, Datum Unterschrift des/der Studierenden

Zusammenfassung

Diese Arbeit geht der Frage nach, welche Technologie sich besser für die Erstellung von *Native Cloud Applications* eignet - Java EE oder Spring Cloud. Dazu werden Anforderungskriterien in den Kategorien Zuverlässigkeit, Umfang, Entwicklung und Betrieb bestimmt. Anhand von Recherchen und der Erstellung eines prototypischen Online-Shops wird dokumentiert, inwieweit die untersuchten Technologien diesen Anforderungen gerecht werden. Dabei wird festgestellt, dass Spring Cloud insbesondere aufgrund der größeren Verbreitung im Cloud-Native-Bereich, aber auch wegen der vorbildhaften und ausführlichen Dokumentation das Mittel erster Wahl für den untersuchten Einsatzzweck ist. Weiterhin bietet es eine große Auswahl an leicht umsetzbaren Lösungen für die Erstellung von cloudfähigen Anwendungen. Java EE kann in Verbindung mit dem Projekt *Eclipse Microprofile* trotz des kleineren Umfangs und der geringeren Verbreitung unter bestimmten Voraussetzungen durchaus als eine mögliche Alternative für den Cloud-Einsatz in Betracht kommen und punktet mit Herstellerunabhängigkeit und Abwärtskompatibilität.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AMQP *Advanced Message Queuing Protocol*

API *Application Programming Interface*

CaaS *Container-as-a-Service*

CNCF *Cloud Native Computing Foundation*

DI *Dependency Injection*

EAR *Enterprise Archive*

IaaS *Infrastructure-as-a-Service*

IDE *Integrated Development Environment*

IoC *Inversion Of Control*

J2EE *Java 2 Enterprise Edition*

JAR *Java Archive*

Java EE *Java Platform, Enterprise Edition*

JCP *Java Community Process*

JDK *Java SE Development Kit*

JPA *Java Persistence API*

JSR *Java Specification Request*

LDAP *Lightweight Directory Access Protocol*

LoC *Lines Of Code*

MDB *Message Driven Bean*

MTTF *Mean Time To Failure*

MTTR *Mean Time To Recovery*

NCA *Native Cloud Application*

NoSQL *Not Only SQL*

PaaS *Platform-as-a-Service*

POM *Project Object Model*

ReST *Representational State Transfer*

SaaS *Software-as-a-Service*

SSO *Single Sign On*

STS *Spring Tool Siute*

VM *Virtuelle Maschine*

WAR *Web Application Archive*

1 Einführung

Softwarehersteller und Dienstleister, welche bisher auf Java EE setzten, überlegen zunehmend, ob sie künftig das Spring Framework verwenden sollen. Insbesondere im Kontext von Cloud-Umgebungen stellt sich die Frage, ob Java EE noch die passende Technologie ist. Angeheizt wird diese Diskussion durch die neuesten Ereignisse rund um das Java EE - Universum wie die Übergabe an die Eclipse Foundation[Del17] und die damit verbundene Unsicherheit bezüglich der Zukunft dieser Technologie[Gua18]. Insbesondere im Cloud-Native-Bereich scheint sich Spring Cloud als Mittel der Wahl durchzusetzen. Diese Arbeit geht der Frage nach, welche Technologie besser für die Erstellung von *Native Cloud Applications* geeignet ist: Java EE oder Spring Cloud. Dazu werden die beiden Technologien auf die Erfüllung von Cloud-Anforderungen untersucht und miteinander verglichen. Das Ergebnis dieser Arbeit soll eine Entscheidungsgrundlage für die Technologieausrichtung von Unternehmen bilden, welche ihre Dienste zukünftig in einer Cloud-Umgebung anbieten wollen und vor der Entscheidung stehen, ob sie ihre bestehende Expertise in Java EE weiterverwenden oder sich auf Spring Cloud ausrichten sollen.

1.1 Ziele der Arbeit

Die Ziele der Arbeit sind Antworten auf nachfolgende Fragen:
Was sind Native Cloud Applications und welche Anforderungen werden an sie gestellt?
Welche Kriterien muss eine Technologie erfüllen, damit sie für die Erstellung von Native Cloud Applications eingesetzt werden kann?
Werden diese Kriterien von Java EE und Spring Cloud erfüllt?
Welche Technologie eignet sich besser für die Erstellung von cloudfähigen Anwendungen?

1.2 Vorgehensweise

Einleitend wird der Begriff *Native Cloud Application* (NCA) definiert und festgelegt, welche Anforderungen an diese gestellt werden. Anschließend wird ein Katalog mit Kriterien erstellt, welche eine Technologie, die zum Erstellen von NCAs eingesetzt wird, erfüllen muss. Zur Durchführung der Untersuchungen ist es notwendig, mit den beiden Technologien jeweils eine prototypische Anwendung mit der selben Funktionalität zu erstellen. Dazu wird festgelegt, welche Funktionalität eine Referenzanwendung umfassen soll. Anhand von Recherchen und von den entstandenen Anwendungen wird dokumentiert, in welchem Maße die Kriterien erfüllt werden. Anschließend werden die Evaluierungsergebnisse miteinander verglichen. Dabei wird herausgearbeitet, welche Technologie den Anforderungen besser gerecht wird. Anhand dieser Erkenntnisse wird eine Empfehlung ausgesprochen, welche Technologie unter welchen Voraussetzungen die bessere Wahl zum Erstellen von Native Cloud Applications ist.

1.3 Aufbau der Arbeit

Im Kapitel **Grundlagen des Cloud Computing** wird der Begriff *Cloud Computing* erläutert sowie die Modelle und Bereitstellungsarten von Cloud-Infrastrukturen erläutert. *Native Cloud Applications* und ihre Eigenschaften - *Containerisierung, Orchestrierung* und *Microserviceorientierter Ansatz* - werden dabei beschrieben. Aus den Anforderungen nach *The Twelve Factor App* werden Komponenten einer cloudfähigen Anwendung abgeleitet.

Das Kapitel **Cloud Native Anwendungen mit Spring Cloud** beschreibt die Technologien Spring Boot und Spring Cloud und demonstriert den Einstieg in die Programmierung anhand eines praktischen Beispiels.

Das Kapitel **Cloud Native Anwendungen mit Java EE** beschreibt den Technologiestandard Java EE sowie die aktuellen Ereignisse rund um die Übergabe des Standards an die Eclipse Foundation und die damit einhergehende Umbenennung in Jakarta EE. Der de-facto-Standard zur Erstellung von Native Cloud Anwendungen und Microservices *Eclipse Microprofile* sowie Application Server, welche diesen Standard unterstützen, werden vorgestellt.

Im Kapitel **Kriterienkatalog** werden Anforderungen festgelegt, welche an eine Technologie gestellt werden, damit sie sich für die Erstellung von NCAs eignet. Die einzelnen Kriterien wie Zuverlässigkeit, Umfang, Entwicklung und Betrieb werden beschrieben.

In **Referenzanwendung** werden die fachlichen und technischen Anforderungen an prototypische Anwendungen herausgearbeitet, welche im Rahmen der Evaluierung erstellt werden. Dabei werden die Domäne festgelegt sowie *User Stories* definiert. Daraus wird die benötigte Architektur abgeleitet und Schnittstellen definiert. Die Architektur der erstellten Anwendungen wird dokumentiert.

In den Kapiteln **Evaluierung Spring Cloud** und **Evaluierung Java EE** wird anhand von Recherchen und der entstandenen Referenzanwendungen dokumentiert, in welchem Maße die im Kriterienkatalog definierten Anforderungen durch die jeweilige Technologie erfüllt werden.

Im Kapitel **Vergleich** werden die beiden untersuchten Technologien anhand der Evaluierungsergebnisse einander gegenübergestellt und eine Empfehlung ausgesprochen, welche Technologie unter welchen Umständen die bessere Wahl für die Erstellung von Native Cloud Applications ist.

2 Grundlagen des Cloud Computing

2.1 Cloud Computing

2.1.1 Definition

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*[NIS11]

Bei Cloud Computing handelt es sich um die dynamische Bereitstellung von konfigurierbaren Computerressourcen über das Netzwerk.
Laut *The NIST Definition of Cloud Computing*[NIS11] besitzt Cloud Computing folgende Eigenschaften:

On-demand self-service
: Nutzer eines Cloud-Dienstes können benötigte Ressourcen selbstständig anfordern, einrichten und wieder freigeben.

Broad network access
: Der Zugriff auf die Infrastruktur ist standardisiert und kann über verschiedene Endgeräte wie PC oder Smartphone erfolgen.

Resource pooling
: Ressourcen werden dynamisch vergeben. Benutzer haben keine Kenntnisse über den physikalischen Standort der darunterliegenden Hardware.

Rapid elasticity
: Ressourcen werden flexibel bereitgestellt, freigegeben und skaliert. Für die Endnutzer erscheinen Ressourcen jederzeit unbegrenzt.

Measured service
: Cloud-Systeme kontrollieren und optimieren Ressourcen automatisch durch den Einsatz von Metriken.

2.1.2 Modelle von Cloud Computing

Es wird zwischen drei Servicemodellen von Cloud Computing unterschieden[NIS11]:

Software-as-a-Service (SaaS)

SaaS bezeichnet die Bereitstellung von in der Cloud-Infrastruktur laufenden Anwendungen. Der Zugriff erfolgt über Endgeräte der Kunden. Die Konfigurationsmöglichkeiten beschränken sich dabei auf benutzereigene Einstellungen und beeinflussen nicht die darunterliegende Infrastruktur.

Platform-as-a-Service (PaaS)

Bei einer PaaS wird die zum Bereitstellen und Betreiben von Anwendungen benötigte Umgebung zur Verfügung gestellt. Diese beinhaltet unterschiedliche Laufzeitumgebungen, Werkzeuge und Bibliotheken. Das Betriebssystem und Netzwerkressourcen werden dabei nicht vom Kunden, sondern vom Dienstleister konfiguriert und verwaltet.

Infrastructure-as-a-Service (IaaS)

Bei der IaaS handelt es sich um die Zurverfügungstellung von Rechnerressourcen. Der Kunde hat dabei Kontrolle über die Konfiguration und Wartung der Ressourcen, nicht aber über die Hardware, auf der die ihm bereitgestellte Infrastruktur basiert.

Neuere Publikationen definieren zusätzlich zwischen der IaaS und PaaS eine vierte Schicht, die *Container-as-a-Service* (CaaS). Bei der CaaS wird dem Kunden eine Laufzeitumgebung zum Betreiben von Containern zur Verfügung gestellt[Rou16].

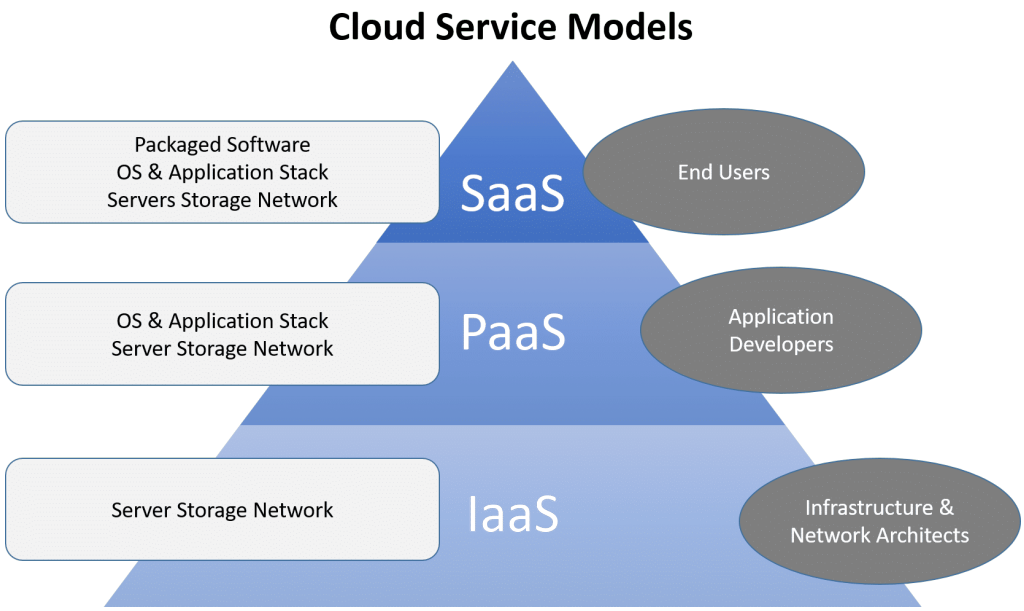

Abbildung 2.1: Modelle von Cloud Computing[Fu17]

2.1.3 Bereitstellungsformen von Cloud Computing

Eine Cloud-Infrastruktur kann auf vier unterschiedliche Arten bereitgestellt werden[NIS11]:

Private Cloud

Bei einer *Private Cloud* wird die Cloud-Infrastruktur zur alleinigen Nutzung durch eine Organisation eingerichtet. Ihre Betreuung wird dabei von der Organisation selbst oder von einem Dienstleister übernommen.

Community Cloud

Bei der *Community Cloud* handelt es sich um eine auf mehrere miteinander kooperierende Organisationen ausgebreitete Form der *Private Cloud*.

Public Cloud

Eine *Public Cloud* wird für die öffentliche Nutzung bereitgestellt. Sie kann von einer wirtschaftlichen, akademischen oder öffentlichen Organisation verwaltet werden.

Hybrid Cloud

Eine *Hybrid Cloud* beschreibt einen Verbund aus zwei oder mehreren unterschiedlichen Arten der Cloud.

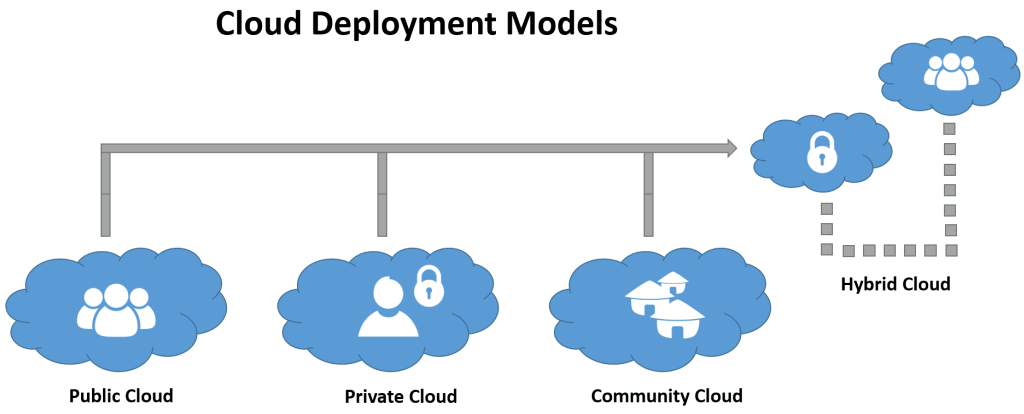

Abbildung 2.2: Bereitstellungsmodelle von Cloud Computing[Fu17]

2.2 Native Cloud Applications

2.2.1 Definition

Native Cloud Applications sind Anwendungen, welche für den Einsatz in einer Cloud-Infrastruktur erstellt werden und ihre Leistungen in vollem Umfang nutzen. Als Definition von *Cloud-Native* hat sich der Ansatz der *Cloud Native Computing Foundation* (CNCF) durchgesetzt.

Abbildung 2.3: Logo CNCF

Die CNCF, eine 2015 gegründete Organisation, welche sich mit der Entwicklung von Cloud-Standards beschäftigt, definiert den Begriff *Cloud-Native* wie folgt[CNC18]:

- ***Container packaged***
  Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation. Improves overall developer experience, fosters code and component reuse and simplify operations for cloud native applications.

- ***Dynamically managed***
  Actively scheduled and actively managed by a central orchestrating process. Radically improve machine efficiency and resource utilization while reducing the cost associated with maintenance and operations.

- ***Micro-services oriented***
  Loosely coupled with dependencies explicitly described (e.g. through service endpoints). Significantly increase the overall agility and maintainability of applications. The foundation will shape the evolution of the technology to advance the state of the art for application management, and to make the technology ubiquitous and easily available through reliable interfaces.

Somit handelt es sich laut dieser Definition nicht bei jeder Anwendung, welche in der Cloud bereitgestellt werden kann, automatisch um eine NCA. Erst das Erfüllen der oben genannten Kriterien, nämlich Containerisierung, dynamische Verwaltung und Microservice-orientierte Architektur qualifiziert sie dazu. In den nachfolgenden Abschnitten werden diese Anforderungen beschrieben.

2.2.2 Containerisierung

Laut 2.2.1 sollen NCAs in Containern bereitgestellt werden.
Bei einem Container handelt es sich um eine *Virtuelle Maschine* (VM), die einer kompletten Anwendung inklusive ihrer Konfigurationen und Abhängigkeiten entspricht. Diese wird in einem vordefinierten und wiederverwendbaren Format verpackt[Aug17]. Im Gegensatz zu herkömmlichen VMs stellen Container kein komplettes Betriebssystem zur Verfügung. Alle in Containern laufenden Prozesse teilen sich den Kernel des Host-Betriebssystems[Koc18].

Container unterscheiden sich in folgenden Punkten von herkömmlichen VMs[Koc18]:

- **Ein Prozess pro Container**
  Im Gegensatz zu VMs kann innerhalb eines Containers nur ein Prozess zur selben Zeit laufen. Dieser Aspekt begünstigt die Regel, dass jeder Container für eine Aufgabe (und nicht mehrere) verantwortlich sein soll.
- **Persistenz**
  Container sind zustandslos und können keine Daten über ihre Lebenszeit hinaus persistieren. Damit es beim Herunterfahren eines Containers zu keinem Datenverlust kommt, müssen Daten außerhalb des Containers gespeichert werden.
- **Portabilität**
  Ein Container verhält sich auf jedem Host-Rechner identisch.

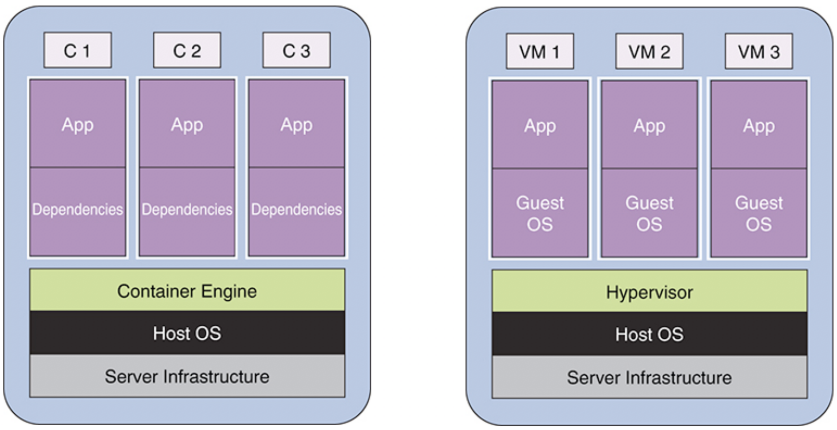

Abbildung 2.4: Unterschied zwischen Containern und VMs[Koc18]

2.2.3 Orchestrierung

Orchestrierung bedeutet die dynamische Verwaltung von Deployment-Instanzen. Diese Aufgabe wird von einer PaaS wie *Cloud Foundry* oder einem Framework wie *Kubernetes* übernommen.

2.2.4 Microservices

Definition

Bei einer *Native Cloud Application* (NCA) handelt es sich laut [CNC18] um eine nach dem Microservice-Ansatz erstellte Anwendung. Der Begriff *Microservices* wird in [MF14] wie folgt beschrieben:

> *"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."*

Bei Microservices handelt es sich somit laut [MF14] um einen Architekturansatz, bei welchem eine Anwendung in Form von mehreren kleinen Services erstellt wird. Diese werden unabhängig voneinander in einem eigenen Prozess ausgeführt und kommunizieren miteinander über ein Netzwerkprotokoll. Sie lassen sich unabhängig bereitstellen und werden zentral verwaltet.

Unterschied zwischen Microservices und Monolithen

Bei einem Monolithen handelt es sich um ein großes Software-System, welches nur als Ganzes auf einmal bereitgestellt werden kann. Es muss als Ganzes alle Phasen der *Continuous-Delivery-Pipeline* durchlaufen. Durch die Größe des Monolithen dauert dieser Prozess länger als bei kleineren Systemen. Das reduziert die Flexibilität und erhöht die Kosten der Prozesse[Wol15b].

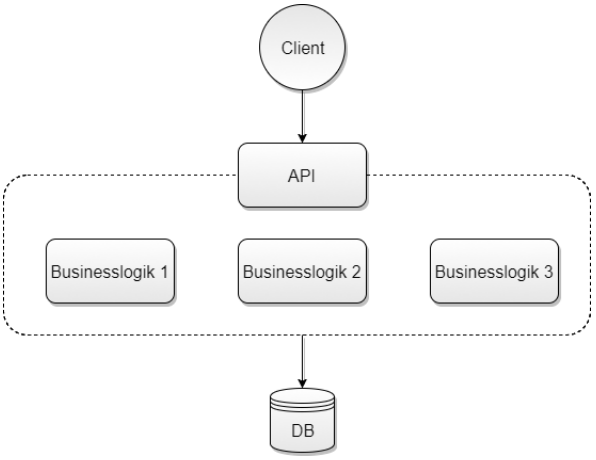

Abbildung 2.5: Monolith

Eine weitere Eigenschaft von Monolithen ist die enge Koppelung ihrer Komponenten untereinander[Til15]. Eine Änderung an einer Stelle kann zu unerwarteten Effekten an anderen, indirekt davon abhängigen Stellen führen. Dadurch ist es schwierig, an mehreren Bereichen eines Monolithen gleichzeitig zu arbeiten. Da ein Monolith in Form einer ganzen Einheit bereitgestellt wird, muss bei jeder Änderung die gesamte Applikation neu erstellt werden[MF14]. Die Anwendung lässt sich nur als Ganzes und nicht in Teilen skalieren[MF14]. Im Gegensatz dazu besteht eine Microservice-Anwendung aus mehreren in sich geschlossenen Einheiten ohne direkte Abhängigkeiten zueinander. Damit lässt sie sich horizontal skalieren und unabhängig von anderen Instanzen erstellen.

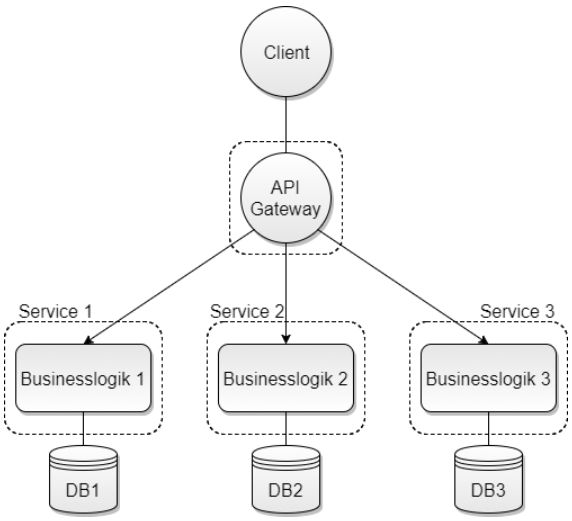

Abbildung 2.6: Microservices

Martin Fowler beschreibt in [Fow15], dass sich der Einsatz von Microservices erst mit steigender Komplexität der Geschäftsanwendung lohnt. Der Mehraufwand zur Verwaltung der Microservice-Architektur bei einer weniger komplexen Anwendung würde die Produktivität verringern. So empfiehlt er, Anwendungsarchitektur zuerst monolitisch zu gestalten und bei steigender Komplexität in einzelne Microservices aufzuteilen.
Es stellt sich die Frage, wo die Grenze zwischen einem Microservice und einem Monolithen verläuft. Wann ist ein Microservice zu groß und wird damit zu einem Monolithen? Neben der Regel *Eine Zuständigkeit pro Service*[MF14] existiert die sogenannte *Zwei-Pizza-Regel*. Laut dieser sollte ein Team maximal so groß sein, dass es von zwei Pizzen satt werden kann[Deu04]. Angewandt auf die Größe von Microservices sollte damit jede Microservice-Instanz von einem Team aus fünf bis maximal sieben Personen betreut werden.

2.2.5 Anforderungen nach The-Twelve-Factor-App

Das Manifest *The-Twelve-Factor-App*[Wig17] wurde im Jahr 2011 von Adam Wiggins, Mitbegründer von *Heroku*, erstellt[Wig18]. Die darin enthaltenen Entwurfsmuster setzen den Fokus auf Performance, Sicherheit und horizontale Skalierbarkeit, indem sie einen Wert auf deklarative Konfiguration, Zustandslosigkeit, Skalierbarkeit und lose Koppelung legen[Sti15]. PaaS-Dienste wie Cloud Foundry, Heroku oder Amazon Elastic Beanstalk sind für den Betrieb von nach diesen Mustern erstellten Anwendungen optimiert[Sti15]. Im nachfolgenden Kontext wird unter *Anwendung* eine unabhängige Microservice-Einheit als Teil eines gesamten verteilten Systems verstanden[Sti15].

1. **Codebase**
   Eine im Versionsverwaltungssystem verwaltete Codebase, viele Deployments
   Jede Anwendung besitzt eine (nicht mehrere) in einem Versionsverwaltungssystem wie Git verwaltete Codebase. Von einer Anwendung mit dem selben Quellcode können mehrere Instanzen in unterschiedliche Umgebungen bereitgestellt werden[Sti15].

2. **Abhängigkeiten**
   Abhängigkeiten explizit deklarieren und isolieren
   Eine Anwendung deklariert ihre Abhängigkeiten explizit über ein Werkzeug wie Maven oder NPM. Sie ist nicht von implizit durch von der Umgebung bereitgestellte *Dependencies* abhängig[Sti15]. Durch die explizite Deklaration der Abhängigkeiten steigt die Stabilität des Gesamtsystems, weil damit vermieden wird, dass Abhängigkeiten nicht miteinander kompatible Versionen haben oder gar fehlen.

3. **Konfiguration**
   Die Konfiguration in Umgebungsvariablen ablegen
   Konfigurationen und alles was sich je nach bereitgestellter Umgebung unterscheidet werden in Form von Umgebungsvariablen zur Verfügung gestellt[Sti15]. Konfigurationsdateien werden unabhängig von der Codebase verwaltet. Damit wird Portabilität der einzelnen Services ermöglicht, weil sie dadurch ohne Anpassungen am Quellcode in unterschiedlichen Plattformen bereitgestellt werden können.

4. **Unterstützende Dienste**
   Unterstützende Dienste als angehängte Ressourcen behandeln
   Unterstützende Dienste wie Datenbanken oder Message Broker werden als angehängte Ressourcen behandelt und in allen Umgebungen identisch genutzt[Sti15].

5. **Build, release, run**
   Build- und Run-Phase strikt trennen
   Die Phasen der Erstellung der Artefakten, der Bereitstellung und des Startens sind voneinander zu trennen[Sti15].

6. **Prozesse**
   Die App als einen oder mehrere Prozesse ausführen
   Die Anwendung führt einen oder mehrere zustandslose Prozesse. Jeder Zustand wird an Dienste wie *Cache* oder *Object Store* ausgelagert[Sti15]. Dadurch wird erreicht, dass die Anwendung jederzeit ohne Datenverlust ersetzt werden kann.

7. **Bindung an Ports**
   Dienste durch das Binden von Ports exportieren

8. **Nebenläufigkeit**
   Mit dem Prozess-Modell skalieren
   Mit Nebenläufigkeit ist in diesem Kontext die horizontale Skalierbarkeit der Services gemeint[Sti15].

9. **Einweggebrauch**
   Robust mit schnellem Start und problemlosen Stopp
   Die Prozesse sollen schnell starten und beenden. Dieser Aspekt begünstigt Skalierbarkeit, Bereitstellung von Änderungen und Wiederherstellung nach Abstürzen[Sti15].

10. **Dev-Prod-Vergleichbarkeit**
    Entwicklung, Staging und Produktion so ähnlich wie möglich halten
    Die Entwicklungsumgebung soll so weit wie möglich mit der Produktionsumgebung identisch sein, um das Verhalten des Systems besser im Griff zu haben und unvorhersehbare Nebeneffekte gering zu halten.

11. **Logs**
    Logs als Strom von Ereignissen behandeln
    Die Behandlung von *Logging* soll nicht in der Anwendung selbst stattfinden, sondern nach außen delegiert werden. Diese werden zentral von einer externen Stelle verwaltet[Sti15].

12. **Admin-Prozesse**
    Admin/Management-Aufgaben als einmalige Vorgänge behandeln
    Administrative Prozesse sollen wie Betriebsprozesse behandelt werden[Sti15].

2.2.6 Bestandteile einer Cloud-Native-Anwendung

Aus *The-Twelve-Factor-App* (2.2.5) sowie aus den Anforderungen an Microservices (2.2.4) lassen sich die wichtigsten Komponenten ableiten, welche Anwendungen, die in modernen Cloud-Systemen betrieben werden, beinhalten sollen:

Verteilte und versionierte Konfiguration

In 2.2.5 wird beschrieben, dass umgebungsspezifische Konfigurationen der Anwendungen wie etwa Hostnamen, Ports und Zugangsdaten in Form von Umgebungsvariablen bereitgestellt werden sollen. Eine verteilte Konfigurationsverwaltung in Form eines Konfigurationsservers bietet darüber hinaus eine Lösung für die zentralisierte Verwaltung der Konfigurationsdateien. Ein Konfigurationsserver hält Konfigurationsdateien für alle zu der Gesamtanwendung gehörenden Microservices bereit und stellt diese über ReST-Schnittstellen zur Verfügung. Die Konfigurationsdateien bezieht der Server dabei von einem Versionsverwaltungssystem wie Git oder Mercurial[Min18].

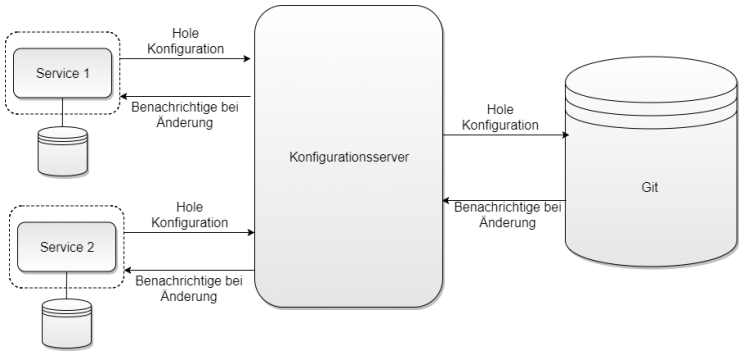

Abbildung 2.7: Konfigurationsserver

Service Discovery

In einer verteilten Systemlandschaft kommt die Frage auf, wie sich die einzelnen Komponenten gegenseitig im Netz finden. In einer vereinfachten Form würde es ausreichen, in den betreffenden Services eine feste Konfiguration zu hinterlegen[Kö16]. In einer Microservice-Infrastruktur mit dynamisch verwalteten Instanzen genügt dieser einfache Ansatz nicht mehr. *Service Discovery* verschafft hierbei Hilfe. Jeder Service meldet sich beim Hochfahren bei einer zentralen Instanz an. Diese speichert seinen Netzwerkstandort unter einem eindeutigen Namen. Andere Services können mithilfe dieser ID über den Discovery-Server auf Instanzen zugreifen, ohne ihren Netzwerkstandort selbst zu kennen. Unter einem Eintrag können beim Einsatz von Load Balancing Adressen von mehreren Instanzen gespeichert werden, welche bei einer Anfrage dynamisch verwaltet werden[Min18].

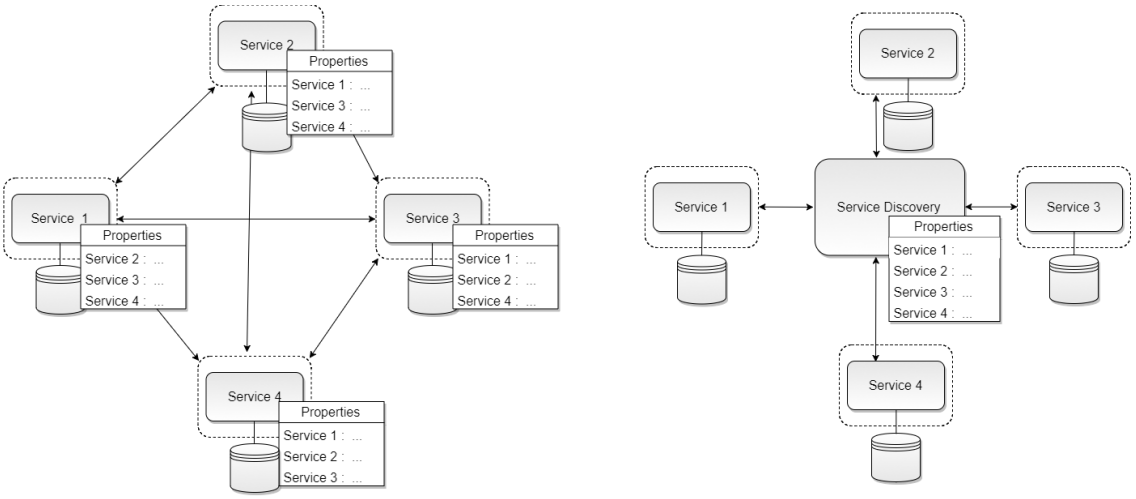

Abbildung 2.8: Kommunikation zwischen Microservices ohne und mit Service Discovery

Routing

Routing erfolgt mithilfe eines *API Gateways*. Ein API-Gateway bietet einen einheitlichen Zugangspunkt für die Kommunikation nach außen. Damit kommunizieren Clients nicht direkt mit den einzelnen Komponenten der Anwendung, sondern mit einer zentralen Instanz. So reicht es aus, dass dem Client die Adresse des Gateways bekannt ist, welches Anfragen an entsprechende Services weiterleitet.

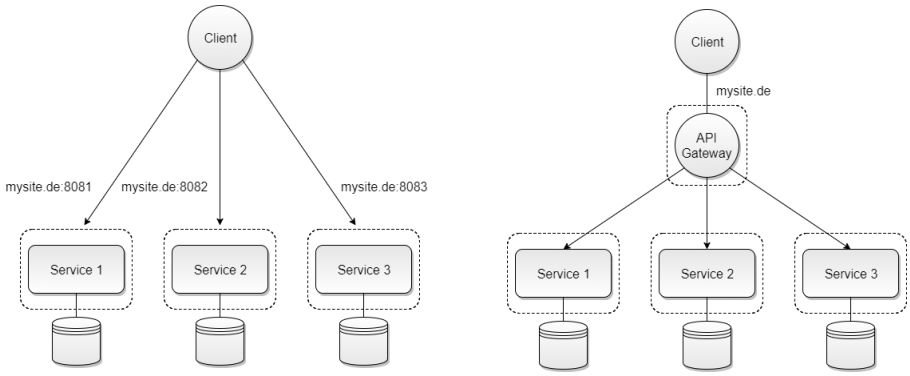

Abbildung 2.9: Services ohne und mit API Gateway

Load Balancing

Unter Load Balancing versteht man die Verteilung von Anfragen auf mehrere Instanzen eines Services. Dies ermöglicht parallele Abarbeitung und Redundanz - Falls ein Knoten ausfällt, werden Anfragen an andere, funktionierende *Nodes* weitergeleitet.

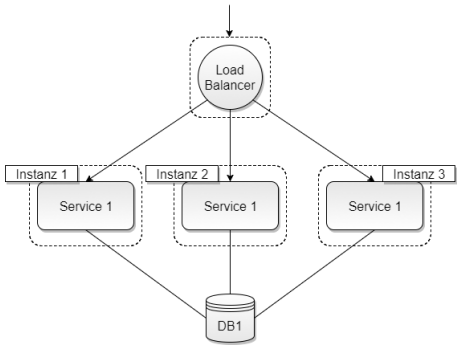

Abbildung 2.10: Load Balancer

Resilience

Es wird zwischen fünf Fehlerarten, welche in verteilten Systemen auftreten können, unterschieden[Ste17]:

Fehlerart	**Beschreibung**
Crash Failure (Absturzfehler)	Das System funktioniert nicht mehr, hat aber bis zum Absturz korrekt funktioniert
Omission Failure (Auslassungsfehler)	Das System kann auf (einzelne) Anfragen nicht mehr reagieren
Timing Failure (Zeitfehler)	Die Antwortzeit liegt außerhalb eines festgelegten Zeitfensters
Response Failure (Antwortfehler)	Das System beantwortet Anfragen nicht korrekt
Arbitrary Failure (Willkürliche Fehler)	Das System produziert willkürliche Antworten zur willkürlichen Zeit

Eine cloudfähige Anwendung muss an eine nicht zu 100% perfekte Infrastruktur angepasst sein und trotz der potenziell auftretenden Fehler hochverfügbar sein.
Die Verfügbarkeit einer Anwendung definiert sich wie folgt[Ste17]:

$$\text{Verfügbarkeit} = \frac{MTTF}{MTTF+MTTR}$$

Die Variablen sind:

- *Mean Time To Failure* (MTTF) - Die durchschnittliche Dauer zwischen dem Starten des Systems und dem ersten Ausfall einer Komponente

- *Mean Time To Recovery* (MTTR) - Die im Durchschnitt benötigte Zeit zum Wiederherstellen der ausgefallenen Komponente

Je kleiner die MTTR, umso näher befindet sich die Verfügbarkeit an der 100%-Marke. Um dies zu erreichen werden *Resilience-Patterns* eingesetzt. Unter *Resilience* versteht man die Fähigkeit einer Software, mit unerwarteten Situationen umzugehen und sich selbstständig von Fehlern zu erholen. Einige dieser Entwurfsmuster werden nachfolgend beschrieben.

Bulkheads

Unter *Bulkhead* versteht man außerhalb von Software-Engineering ein Schott, welches im Falle eines Lecks die Überflutung des gesamten Schiffes abwendet. Dieses Prinzip kann auf Software übertragen werden[Rot14]. Dazu werden die einzelnen Komponenten möglichst lose miteinander gekoppelt. So wird erreicht, dass der Absturz einer einzelnen Komponente nicht den Absturz des gesamten Systems nach sich zieht.

Circuit Breaker

Circuit Breaker bedeutet *Sicherung* und beschreibt eine Schutzeinrichtung, die auslöst, wenn eine festgelegte Menge von Ereignissen einen Kreislauf über eine vorgegebene Zeit hinaus überschreitet[Sim18].

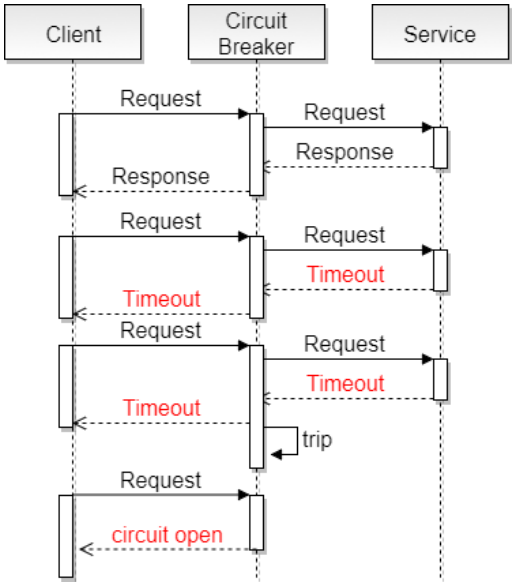

Abbildung 2.11: Circuit Breaker (nach [Fow14])

Wenn innerhalb einer festgelegten Zeit eine Anzahl von Anfragen fehlschlägt oder ihre Verarbeitung zu lange dauert, greift das Schutzmechanismus des Circuit Breakers ein. Dabei werden ankommende Anfragen nicht mehr weitergeleitet, sondern direkt mit einer Fehlermeldung oder einer vordefinierten Rückmeldung(*Fallback*) beantwortet. Nach bestimmter Zeit überprüft der Circuit Breaker, ob der Service sich erholt hat und wechselt wieder in den gewöhnlichen Betriebszustand[Fow14].

Persistenz

Damit die einzelnen Komponenten eines verteilten Systems unabhängig voneinander bleiben, ist es notwendig, dass jeder Service über eine eigene Datenbankinstanz verfügt. Dadurch wird die Verwendung unterschiedlicher Datenbanktypen innerhalb eines Systems (*Polyglot Persistence*) ermöglicht[MF14]. Es wird zwischen relationalen Datenbanken und *Not Only SQL* (NoSQL)-Datenbanken unterschieden. Während klassische relationale Datenbanksysteme bekannter sind, sind ihnen NoSQL-Datenbanken in vielen Belangen überlegen. Es existieren unterschiedliche Arten von NoSQL-Datenbanken mit verschiedenen Datenmodellen, welche für bestimmte Anwendungsfälle angepasst sind. Damit eignen sie sich insbesondere für Microservices, welche in der Regel einen einzigen, vordefinierten Geschäftsbereich abdecken[Lon17].

Caching

Die einzelnen Services sollen zustandslos sein, um clientseitiges Load-Balancing zu ermöglichen und jederzeit ersetzbar zu sein. Dazu wird jegliche Art von Zustand außerhalb der Instanz gespeichert. Neben persistenten Daten existieren sessionbezogene Daten, welche für die Dauer einer Session gespeichert werden müssen. Dies geschieht mithilfe von *Caching*.

Synchrone Kommunikation mit ReST

Eine verbreitete Art der Bereitstellung von Ressourcen sowie der verteilten Kommunikation ist *Representational State Transfer* (ReST). ReST wurde 2000 von Dr. Roy Fiedling als Teil seiner Dissertation vorgestellt[Lon17]. Dabei handelt es sich um einen Architekturstil zur Kommunikation im Internet. Beim ReST wird das *HTTP Request-Response-Modell* verwendet, um den Zugang zu Ressourcen zu ermöglichen. Die Grundprinzipien von ReST sind nach [Wol15a]:

- **Ressourcen mit eindeutiger Identifikation**
  Ressourcen werden in einer abstrahierten Form unter einer eindeutigen Adresse zur Verfügung gestellt. Die Adressen sollen dabei von Menschen lesbar und leicht zu interpretieren sein, um sowohl die browserbasierte als auch die Anwendung-zu-Anwendung-Kommunikation zu vereinheitlichen.
- **Verknüpfungen und Hypermedia**
  Es werden Verknüpfungen verwendet, um Beziehungen zwischen Ressourcen herzustellen.
- **Standardmethoden**
  ReST verwendet die gängigen *HTTP Verbs (GET, POST, PUT, DELETE, HEAD und OPTIONS)* für den Zugriff und Manipulation von Ressourcen.
- **Unterschiedliche Repräsentationen**
  Die selben Ressourcen können auf eine unterschiedliche Art, abhängig von der Anfrage, bereitgestellt werden.

- **Zustandslose Kommunikation**
  Jede Anfrage wird wie eine neue, unabhängige Anfrage betrachtet. Der Zustand wird nicht vom Server, sondern bei Bedarf vom Client verwaltet.

Die Kommunikation über ReST verläuft *synchron* - der Aufrufer versendet eine Anfrage und wartet blockierend auf eine Antwort.

Asynchrone Kommunikation mit Messaging

Die interne Kommunikation zwischen den Services kann synchron in Form von ReST erfolgen. Alternativ kann dazu der asynchrone Austausch von Nachrichten in Form von *Messaging* eingesetzt werden. Nachrichten können dabei an einen oder mehrere Empfänger gehen und zu einer Antwort führen, die wiederum als Nachricht verschickt werden kann[Wol15b]. Messaging kann auf zwei unterschiedliche Arten erfolgen: *Point-To-Point* und *Publish-Subscribe*. Bei der Point-to-Point-Kommunikation werden Nachrichten über eine *Queue* zwischen zwei Instanzen ausgetauscht. Beim Publish-Subscribe Modell werden Nachrichten an einen Topic publiziert. Beliebig viele Instanzen können sich eine Kopie dieser Nachricht abholen, indem sie diesen Topic abonnieren.

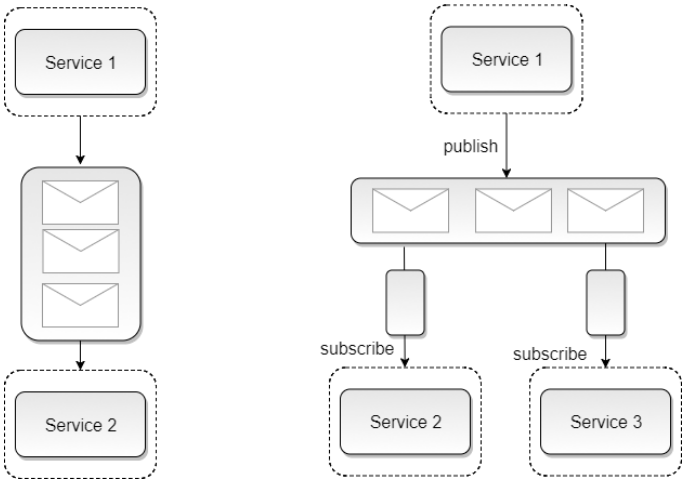

Abbildung 2.12: Messaging mit Queues und Topics

Logging, Tracing und Monitoring

Punkt 11 von *Bestandteile einer Cloud-Native-Anwendung* besagt, dass Logging nicht von den Services selbst behandelt wird, sondern in Form von Ereignisströmen erfolgen soll. Weiterhin muss mit *Tracing* das Zurückverfolgen von Anfragen über Instanzgrenzen hinweg sowie *Monitoring* zur Überwachung von laufenden Instanzen ermöglicht werden.

Security

Damit nur Berechtigten Zugriff auf bestimmte Ressourcen gewährt wird, bedarf es einer Authentifizierung. Ein Benutzer identifiziert sich mithilfe eines eindeutigen Benutzernamens und Passworts. Dies kann über mehrere Wege erfolgen[Wol15a]:

- **HTTP Basic**
  Bei *HTTP Basic* handelt es sich um die einfachste Form der Authentifizierung. Dabei werden die Benutzerdaten im *Header* der ReST-Anfrage in Form eines *base64*-codierten Strings verschickt. Weil ReST zustandslos ist, müssen diese Daten mit jeder Anfrage erneut versendet werden. Da die Benutzerdaten inklusive Password dabei unverschlüsselt übertragen werden, muss der Zugriff auf die API unbedingt über eine verschlüsselte Verbindung stattfinden. Eine Authentifizierung mittels HTTP-Basic über eine unverschlüsselte Verbindung ist riskant, da dabei Abhörungen durch *Man-In-The-Middle*-Attacken in keinster Weise verhindert werden.

- **HTTP Digest Authentication**
  Bei dieser Art der Authentifizierung handelt es sich um eine Erweiterung von HTTP Basic. Dabei werden die Benutzerdaten vor dem Versenden mithilfe eines vom Server mitgeteilten Wertes verschlüsselt und nicht in Klartext übertragen.

- **OAuth2 und Open ID Connect**
  Die nachfolgenden Begriffe sind Bestandteile von OAuth 2.0[Ora18a]:
  - **Resource Owner**
    Der *Resource Owner* ist eine Instanz, welche bestimmte Ressourcen, die ihr gehören, anfordert. In der Regel handelt es sich dabei um den Endnutzer.
  - **Resource Server**
    Auf dem *Resource Server* befinden sich die geschützten Ressourcen. Er validiert Anfragen anhand von *Tokens*.
  - **Client Application**
    Eine *Client Application* macht Anfragen nach den geschützten Ressourcen auf Anweisung und im Namen des *Resource Owners*.
  - **Authorization Server**
    Der *Authorization Server* vergibt nach erfolgreicher Authentifizierung Access Tokens an die *Client Application*. Damit kann sie auf die gewünschten Ressourcen zugreifen.

  OpenID Connect erweitert OAuth 2.0 darüber hinaus um den Einsatz von JWT-Tokens. Diese können vom Resource Server mithilfe eines Public Keys verifiziert werden. Außerdem beinhalten sie Informationen über die Identität des autorisierten Benutzers.

3 Cloud Native Anwendungen mit Spring Cloud

3.1 Spring Boot

> *Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".*[piv18b]

Bei Spring Boot handelt es sich um ein von *Pivotal Software Inc.* entwickeltes Open-Source Framework. Spring Boot basiert auf dem Spring Framework und ermöglicht einfaches Erstellen von lauffähigen Microservices mit minimalem Aufwand. Spring Boot verwendet Technologien wie *Convention over Configuration*, eine ausgeklügelte Abhängigkeitsverwaltung in Form von *Starter-Abhängigkeiten* sowie einen eingebetteten Servlet-Container. Es wird weitestgehend auf die vom klassischem Spring Framework bekannten Konfigurationen in Form von XML-Dateien verzichtet und stattdessen auf Annotationen und Konfigurationsklassen gesetzt[Lar14].
Nachfolgend wird eine minimale lauffähige Webanwendung demonstriert, welche einen ReST-Service beinhaltet. Dieses *Hello World*-Beispiel entstammt der unter [piv18a] verfügbaren Anleitung.

1. **Abhängigkeiten**

   In der *Project Object Model* (POM)-Datei eines mithilfe von *Maven* erstellten Projekts werden nachfolgende Abhängigkeiten definiert:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.5.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

   Die *Parent*-Abhängigkeit beinhaltet Versionsnummern für die nachfolgend definierten Abhängigkeiten, sodass diese nicht manuell eingetragen werden müssen.

   Bei `spring-boot-starter-web` handelt es sich um eine Starter-Abhängigkeit, welche für den gewünschten Anwendungsfall einen Satz von benötigten *Dependencies* mitbringt. So beinhaltet `spring-boot-starter-web` unter anderem eine Abhängigkeit

für das Anbieten von ReST-Schnittstellen sowie den eingebetteten Servlet-Container *Apache Tomcat*[Par18a].

2. **ReST-Controller**

   Der nachfolgend abgebildete ReST-Controller gibt beim Aufruf des *Root-Endpoints* ("/") `Greetings from Spring Boot!` zurück:

```
1 @RestController
2 public class HelloController {
3     @RequestMapping("/")
4     public String index() {
5         return "Greetings from Spring Boot!";
6     }
7 }
```

3. **Anwendungsklasse**

   Die Anwendungsklasse ist der Einstiegspunkt der Anwendung. Sie beinhaltet die Methode `public static void main()` und wird mit der Annotation `@SpringBootApplication` versehen:

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

Diese drei Schritte reichen für die Erstellung einer lauffähigen Anwendung aus. Beim Starten fährt der eingebettete *Tomcat* hoch und die Funktionalität des ReST-Endpoints kann mithilfe von *curl* oder *Postman* überprüft werden.

Spring Initializr

Spring Initializr ist eine benutzerfreundliche Art, neue Spring-Boot-Projekte zu erstellen. Dabei handelt es sich um eine unter `http://start.spring.io` erreichbare Anwendung. In ihrer Benutzeroberfläche kann interaktiv ausgewählt werden, welche Art von Spring-Boot-Projekt erstellt werden soll. Anschließend wird ein Grundgerüst eines Projekts, ähnlich einem *Maven-Archetype*, generiert und zum Herunterladen bereitgestellt. Dieses kann in die lokale Entwicklungsumgebung eingebunden und als Ausgangsbasis benutzt werden.

3.2 Spring Cloud

> *Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.*[Piv18b]

Spring Cloud ist ein Schirmprojekt von Spring Boot, welches verschiedene Unterprojekte beinhaltet, die zur Erstellung von Cloud-Native-Anwendungen eingesetzt werden können.

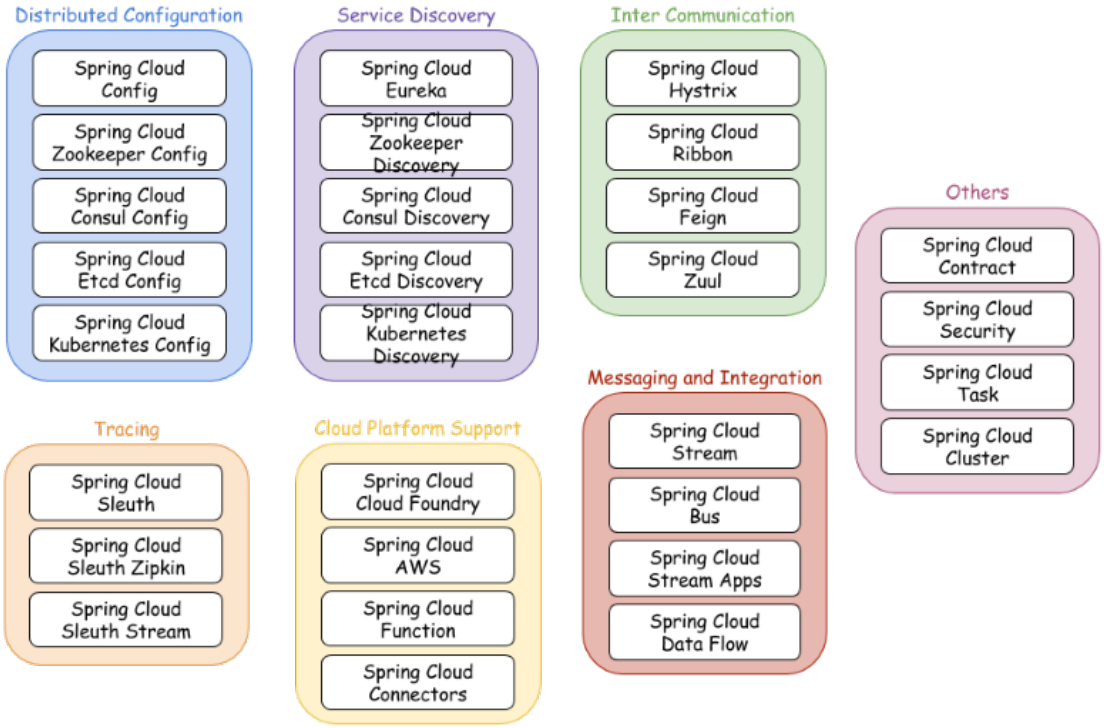

Abbildung 3.1: Bestandteile von Spring Cloud[Min18]

4 Cloud Native Anwendungen mit Java EE

4.1 Java EE

4.1.1 Definition

Bei *Java Platform, Enterprise Edition* (Java EE) handelt es sich um die Spezifikation einer Softwarearchitektur für die transaktionsbasierte Ausführung von in Java programmierten Anwendungen und Web-Anwendungen[Wik18]. Die erste Version des Standards erschien im Jahr 1999 unter dem Namen *Java 2 Enterprise Edition* (J2EE). Der Standard wurde bis September 2017 von *Oracle Corporation* im Rahmen des *Java Community Process* (JCP) weiterentwickelt. Dabei wurden alle Neuerungen und Verbesserungsvorschläge von Mitgliedern des JCP, Vertreten von namhaften Technologieunternehmen wie IBM und Red Hat in Form von *Java Specification Request* (JSR) vorgeschlagen. Deren Übernahme in kommende Java EE-Spezifikationen wurde unter allen Mitgliedern der JCP abgestimmt[Ora18b]. Die aktuelle Version ist seit 31.07.2017 Java EE 8. Die vollständige Spezifikation umfasst 277 Seiten und befindet sich unter [Sha17].
In der nachfolgenden Abbildung werden die Spezifikationen von Java EE 8 dargestellt. Dabei handelt es sich um standardisierte Schnittstellen, welche von den jeweiligen Herstellern von *Application Servern* implementiert werden müssen, um für ihr Produkt eine Java EE-Zertifizierung zu erlangen.

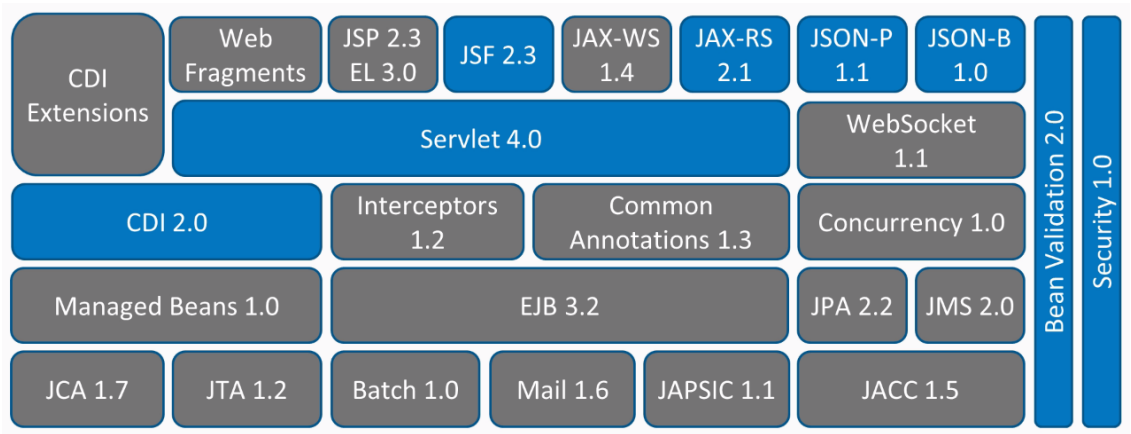

Abbildung 4.1: Spezifikationen von Java EE 8[Eis16]

4.1.2 Laufzeitumgebung

Die nach dem Java EE-Standard entwickelten Anwendungen sind innerhalb eines *Application Servers* lauffähig. Dieser stellt die für die Laufzeit benötigte Umgebung im Form der Implementierung des Java EE - Standards zur Verfügung. Application Server, welche Java EE 8 unterstützen sind *Glassfish, Red Hat WildFly* sowie *IBM Websphere Liberty Profile*[Ora18d].

4.2 Jakarta EE

Am 12.09.2017 wurde bekanntgegeben, dass die Betreuung und Weiterentwicklung des Java EE-Standards an die *Eclipse Foundation* übergeben wird[Del17]. Bei der Eclipse Foundation handelt es sich um eine globale Community, welche die Entwicklung von über 350 Open-Source-Projekten betreut[Joh18]. Trotz Bedenken[Gua18] hat Oracle die Rechte an dem Markenzeichen *Java* im Rahmen der Übergabe nicht abgetreten. Aus diesem Grund wird die kommende Version des Standards nicht mehr diesen Namen tragen dürfen. Als Ergebnis einer mehrmonatigen Abstimmung wurde am 26.02.2018 verkündet, dass der neue Name *Jakarta EE* lauten wird[Mil18]. Die Weiterentwicklung des Standards wird in einer ähnlichen Form wie JCP unter dem Namen *Jakarta EE Working Group* weitergeführt[Joh18].

Abbildung 4.2: Logo Jakarta EE

4.3 Native Cloud Applications mit Java EE

Da es sich bei Java EE um einen sich seit 1999 entwickelnden Standard handelt, ist dieser primär auf die Entwicklung von Monolithen ausgerichtet[Ora14].
Ein großer Wunsch der Community ist, dass sich Java EE in der kommenden und ersten von der Eclipse Community betreuten Version, Jakarta EE, vor allem in Richtung Microservices und Cloud-Native weiterentwickelt[Joh18].
Ein wichtiger Inkubator für die Ausrichtung von Java EE in Richtung Cloud Native ist das nachfolgend vorgestellte Projekt *Eclipse Microprofile*.

4.3.1 Eclipse Microprofile

> *Microprofile ist eine im Jahr 2016 von führenden Application-Server-Herstellern (Red Hat, IBM, Tomitribe und Payara) ins Leben gerufene Initiative. Ihr Ziel ist es, das über Jahre aufgebaute Wissen der Java-EE-Community auch für die Entwicklung und den Betrieb von Microservices zu nutzen, ohne dabei an die starren Restriktionen des Java-Enterprise-Standards gebunden zu sein. Erreicht werden soll dies durch ein schmales Bundle von Java-EE-APIs (JAX-RS + CDI + JSON-P), ergänzt um neue, speziell auf die Anforderungen von Microservices zugeschnittene APIs.*[Rö18]

Microprofile wird von der Eclipse Community betreut und ist nicht an einen bestimmten Hersteller gebunden. Bei Microprofile handelt es sich um den de-facto-Standard zum Erstellen von cloudfähigen Microservices mit Java EE.

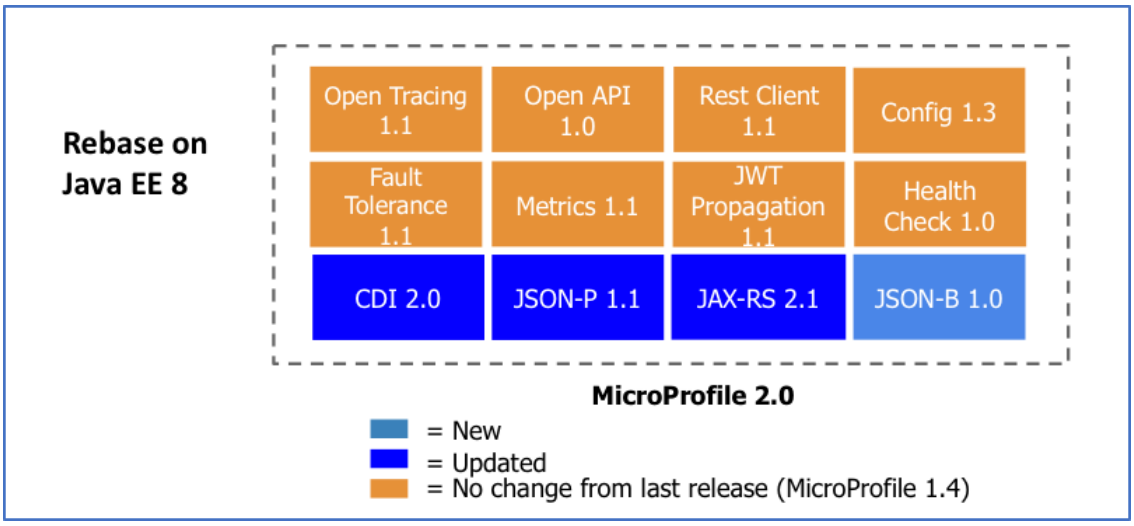

Abbildung 4.3: Bestandteile von Eclipse Microprofile 2.0[Saa18]

4.3.2 Lightweight Application Server

Traditionelle Application Server sind groß und brauchen vergleichsweise lange zum Hoch- und Herunterfahren. Das liegt daran, dass sie die komplette Implementierung der `javax`-API (*Full Profile*) beinhalten. Damit sind sie nicht optimal zum Betrieb von voneinander entkoppelten Microservices geeignet.
Dazu werden leichtgewichtige, einbettbare Application Server benötigt. Sie sollen möglichst kompakt sein sowie schnell hoch- und herunterfahren.
Nachfolgend werden Projekte beschrieben welche diesen Anforderungen genügen sowie den Microprofile-Standard implementieren:

Red Hat Thorntail

Thorntail (Bis 2018 *WildFly Swarm*) ist eine modularisierbare Version des Application Servers *WildFly*. Um ihn zu verwenden, reicht es aus, die entsprechenden Abhängigkeiten in der POM des Projektes festzulegen. Beim Ausführen des Goals `mvn package` wird eine ausführbare *Uberjar* erstellt, welche sowohl den Anwendungscode als auch den Application Server beinhaltet.
Thorntail lässt sich modularisieren - die einzelnen benötigten Implementierungen von Java EE werden explizit angegeben. Alternativ ist es möglich, das Framework über einen *Autodependency-Modus* bestimmen zu lassen, welche Abhängigkeiten zum Ausführen der Anwendung benötigt werden und diese automatisch aufzunehmen[Fro17].

IBM Open Liberty

Die von IBM entwickelte Open-Source-Version des Application-Servers *Websphere Liberty Profile* wird vom Hersteller als *The most flexible server runtime available to Earth's Java developers* bezeichnet[IBM18a]. Open Liberty unterstützt Java EE 8 sowie Eclipse Microprofile in der aktuellen Version[IBM18a]. Um kommerziellen Support für Open Liberty zu erhalten, kann das Produkt *IBM Websphere Liberty Profile* erworben werden, welches auf dem Quellcode von Open Liberty basiert, aber unter einer kommerziellen Lizenz vertrieben wird[Psc17].

KumuluzEE

KumuluzEE ist eine Implementierung des Microprofile-Standards in der Version 1.2. Es bietet eine Reihe von für die Erstellung von NCAs nützlichen Funktionen wie Service Discovery und Configuration Server[Kum18].

Weitere Lösungen

Neben den beschriebenen Produkten existieren mit *Payara Micro* sowie *Apache TomEE* zwei weitere Applicaion Server, welche den Microprofile-Standard implementieren.

5 Kriterienkatalog

In diesem Kapitel werden Kriterien festgelegt, welche von einer Technologie erfüllt werden müssen, damit sie sich für die Entwicklung von cloudfähigen Anwendungen eignet. Es wird versucht, diese möglichst objektiv festzulegen. Selbstverständlich hängt die letztendliche Entscheidung zusätzlich von individuellen Faktoren wie Kenntnisstand des Entwicklerteams ab, welche an dieser Stelle nicht erfasst werden können.
Die Bewertungskriterien sind in die Kategorien *Zuverlässigkeit, Umfang, Entwicklung, Betrieb* und *Sonstige Bewertungskriterien* unterteilt.

5.1 Zuverlässigkeit

Eine Technologie soll ausreichend dokumentiert sein und über eine große Community verfügen. Außerdem ist es wichtig, dass sie regelmäßig aktualisiert wird und abwärtskompatibel bleibt.

Kriterium	**Unterkriterium**	**Bewertungspunkte**
Dokumentation	Qualität und Umfang	Vollständigkeit
		Ausführlichkeit
		Nachvollziehbarkeit
	Anleitungen	Vorhandensein
		Umfang
	Aktualität	Aktualität
		Versionierung
Support	Community Support	Größe der Community
		Stack Overflow
	Enterprise Support	Vorhandensein
		Kosten
Aktualität	Aktivität auf Git	Aktivität
		Contributor
	Updates	Release-Zyklen
		Neuerungen
		Bug Fixes
Kompatibilität	Abwärtskompatibilität	Abwärtskompatibilität
		Breaking Changes

Abbildung 5.1: Bewertungskriterien Zuverlässigkeit

5.1.1 Dokumentation

Wichtige Entscheidungskriterien für den Einsatz einer Technologie sind das Vorhandensein, Vollständigkeit und die Nachvollziehbarkeit ihrer offiziellen Dokumentation.

Qualität und Umfang

- **Vollständigkeit**
  Ist die Technologie vollständig dokumentiert?
- **Ausführlichkeit**
  Beschreibt die Dokumentation alle relevanten Aspekte ausführlich?
- **Nachvollziehbarkeit**
  Sind die Dokumentationen nachvollziehbar und verständlich?

Anleitungen

Tutorials erleichtern den Einstieg in eine Technologie.

- **Vorhandensein**
  Sind Anleitungen vorhanden?
- **Umfang**
  Sind ausreichend Beispiele und Anleitungen vorhanden?

Aktualität

Mit Aktualisierungen des Produkts soll ebenfalls eine aktualisierte Dokumentation erscheinen. Damit wird eine Inkonsistenz zwischen der aktuellen und dokumentierten Version vermieden. Ebenfalls ist zu Begrüßen, wenn vorangegangene Dokumentationen nicht überschrieben, sondern versioniert zur Verfügung gestellt werden. So wird Nutzern, welche nicht die aktuellste Version verwenden, dennoch der Zugriff auf die passende Dokumentation gewährt.

- **Aktualität**
  Ist die aktuelle Version der Technologie dokumentiert?
- **Versionierung**
  Sind Dokumentationen für nicht mehr aktuelle Versionen vorhanden?

5.1.2 Support

Community Support

Es sollte eine ausreichend große Entwicklergemeinschaft (*Community*) vorhanden sein, auf deren Erfahrungswerte und Wissen zurückgegriffen werden kann. Als eine der wichtigsten Anlaufstellen bei Fragen und Problemen im Umfeld der Software-Entwicklung hat sich *Stack Overflow* etabliert. Mailing Listen sowie Internetforen sind eine weitere Möglichkeit, sich bei Problemen Hilfe zu verschaffen.

- **Stack Overflow**
  Wie präsent ist das Projekt auf Stack Overflow?

Enterprise Support

Für den professionellen Einsatz ist es vorteilhaft, wenn kommerzielle Unterstützung geboten wird.

- **Vorhandensein**
  Ist kommerzieller Support vorhanden?
- **Kosten**
  Mit welchen Kosten ist kommerzieller Support für die Technologie verbunden?

5.1.3 Aktualität

Dieses Bewertungskriterium betrachtet die Aktualität der Technologie und konzentriert sich dabei auf die Punkte *Aktivität auf Git* sowie *Updates*.

Aktivität auf Git

Sowohl Spring Cloud als auch Technologien, welche den Java EE-Standard um die Fähigkeit zum Erstellen von NCAs erweitern, werden in Form von Open-Source-Projekten entwickelt. Ihr Quellcode ist auf GitHub verfügbar. Dort lassen sich einzelne Aktivitäten und *Commits* verfolgen. Daraus können Erkenntnisse gewonnen werden, wie aktiv und von wie vielen Personen das Projekt vorangetrieben wird.

- **Aktivität**
  Wird das Projekt aktiv weiterentwickelt oder über längere Zeiträume nicht gepflegt?
- **Anzahl der Contributor**
  Wie viele Entwickler sind aktiv an dem Projekt beteiligt?

Updates

Eine produktiv einzusetzende Technologie muss stets gepflegt und aktuell gehalten werden. Dabei sind insbesondere *Bug Fixes* sowie sicherheitsrelevante Aktualisierungen wichtig. Neue Funktionen sollen ebenfalls regelmäßig Einzug finden, damit die Technologie in der Zukunft konkurrenzfähig bleibt.

- **Release-Zyklen**
  In welchen zeitlichen Abständen werden aktualisierte Versionen veröffentlicht?
- **Neuerungen**
  Finden Neuerungen Einzug in die Technologie?
- **Bug Fixes**
  Werden Probleme schnell und zuverlässig behoben?

5.1.4 Kompatibilität

Abwärtskompatibilität

Zum Vornehmen von Updates sollte es im Idealfall ausreichen, die Technologien zu aktualisieren, ohne Änderungen am Quellcode der Anwendungen vornehmen zu müssen. Das Gegenteil davon sind *Breaking Changes* - Änderungen an der Grundfunktionalität, welche Anpassungen am Anwendungscode benötigen. Das Anpassen der nach dem Aktualisieren nicht mehr funktionierenden Komponenten bedeutet zusätzlichen Zeit- und Entwicklungsaufwand und bringt Risiken mit sich, weil sich mit jeder Anpassung neue Fehler einschleichen können.

- **Abwärtskompatibilität**
  Ist die Verwendung von Anwendungen, welche für eine ältere Version der Technologie erstellt wurden, mit neueren Versionen möglich?
- **Breaking Changes**
  Ist die Einführung von neuen Versionen mit Migrationsaufwand verbunden?

5.2 Umfang

Im Kapitel *Bestandteile einer Cloud-Native-Anwendung* (2.2.6) wurde festgelegt, dass eine NCA folgende Bestandteile beinhalten soll:

- Verteilte und versionierte Konfiguration
- Service Discovery
- Routing
- Load Balancing
- Resilience
- Persistenz
- Caching
- Synchrone Kommunikation mit ReST
- Asynchrone Kommunikation mit Messaging
- Logging, Tracing und Monitoring
- Security

Die untersuchte Technologie soll Lösungen beinhalten, welche eine Umsetzung dieser Punkte ermöglichen. Die dabei zu berücksichtigenden Kriterien sind Vorhandensein, Integration und Aufwand. Im Idealfall ist eine integrierte Lösung der einzelnen Aspekte erwünscht, welche nicht von Drittanbietern abhängt und sich einfach einbinden lässt.

5.3 Entwicklung

Für die Entwicklungstätigkeit sind die Punkte *Aufwand, Testbarkeit* und *Werkzeuge* von Bedeutung.

Kriterium	Unterkriterium	Bewertungspunkte
Aufwand	Lines Of Code	Anzahl
	Konfiguration	Art der Konfiguration
	Convention Over Configuration	Verwendung
Testbarkeit	Unit Tests	Unterstützung bei der Erstellung
		Mocking
	Integrationstests	Unterstützung bei der Erstellung
Werkzeuge	Entwicklungsumgebung	Unterstützung

Abbildung 5.2: Bewertungskriterien Entwicklung

5.3.1 Aufwand

In diesem Abschnitt wird betrachtet, mit wie viel Aufwand die Erstellung einer lauffähigen Anwendung verbunden ist. Bewertungskriterien sind hierbei Anzahl der *Lines Of Code* (LoC), Art der Konfiguration sowie Einsatz von *Covention Over Configuration.*

Lines Of Code

Es ist von Vorteil, wenn zum Erstellen einer Anwendung wenige Codezeilen (LoC) benötigt werden. Einerseits wird dadurch die Effizienz der Arbeit erhöht, andererseits sinkt damit der Wartungsaufwand und die Anzahl der potenziellen Fehlerquellen.

- **Anzahl der LoC**
  Wie viele LoC beinhaltet eine mit der untersuchten Technologie erstellte Anwendung?

Konfiguration

Die Konfiguration kann mithilfe von XML-Dateien, über externe Konfigurationsdateien oder mithilfe von Annotationen erfolgen.

- **Art der Konfiguration**
  In welcher Form findet die primäre Konfiguration statt?
- **Aufwand**
  Wie aufwendig gestaltet sich die Konfiguration der Anwendung?

Convention Over Configuration

Convention Over Configuration ist ein Prinzip der Softwareentwicklung. Dabei werden Konventionen festgelegt, welche angenommen werden, solange vom Programmierer nichts

anderes explizit festgelegt wird. Die Verwendung von Convention Over Configuration erleichtert die Arbeit, da alle Standardeinstellungen und Konventionen automatisch vorgenommen werden. Damit reduziert sich ebenfalls die bereits erwähnte Anzahl an potenziellen Fehlerquellen.

- **Verwendung**
  Wird Convention Over Configuration eingesetzt?

5.3.2 Testbarkeit

Um Fehler in der Produktion zu vermeiden, soll der Anwendungscode ausgiebig getestet werden. In der Softwareentwicklung wird zwischen Unit-, Integrations- und UI-Tests unterschieden. Die Testpyramide von Mike Cohn[Coh09], eine Richtlinie zum Verhältnis der Anzahl von Unit-, Integrations- und UI-Tests zueinander, besagt, dass Unit-Tests die Grundlage von der Testautomatisierung sein sollen, da diese mit geringem Aufwand erstellt werden und die größte Aussagekraft zur Fehlerbestimmung besitzen. Service- bzw. Integrationstests sollen auf Unit-Tests aufbauen sowie einige UI-Tests implementiert werden.

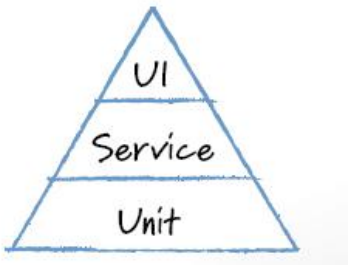

Abbildung 5.3: Testpyramide[Coh09]

Unit Tests

- **Unterstützung bei der Erstellung von Unit-Tests**
  Bietet die Technologie Unterstützung bei der Erstellung von Unit-Tests?

- **Mocking**
  Mocking ist ein Kernaspekt vom Unit-Testing. Beim Mocking werden externe Abhängigkeiten, welche die zu testende Klasse benötigt, weggekapselt, indem sie durch *Mocks* ersetzt werden. Damit wird sichergestellt, dass nur das Verhalten einer Klasse und nicht die Interaktion mehrerer Klassen untereinander getestet wird. Durch das Framework soll eine Unterstützung von Mocking gegeben sein.

Integrationstests

- **Unterstützung bei der Erstellung von Integrationstests**
  Bietet das Framework Unterstützung bei der Erstellung von Integrationstests?

5.3.3 Werkzeuge

Entwicklungsumgebung

Das wichtigste Werkzeug, mit welchem Softwareentwickler arbeiten, ist die *Integrated Development Environment* (IDE). Es ist zu Begrüßen, wenn sie grundlegende Funktionen der Technologie beherrscht und Unterstützung für die Arbeit bietet.

- **Unterstützung**
  Wird IDE-Support geboten?

5.4 Betrieb

In diesem Abschnitt werden Aspekte betrachtet, welche das Laufzeitverhalten der mit der jeweiligen Technologie erstellten Systeme betreffen. Besonderen Augenmerk bekommen dabei die Punkte *Ressourcenverbrauch* und *Performance*. Administrierungs- und Überwachungsmöglichkeiten der Anwendung wurden bereits im Abschnitt *Umfang* (5.2) unter dem Punkt *Logging, Tracing und Monitoring* erfasst.

Kriterium	Unterkriterium	Bewertungspunkte
Ressourcenverbrauch	Speicherbedarf	Größe der Artefakte
		Größe des Gesamtsystems
	Ressourcenverbrauch zur Laufzeit	Arbeitsspeicherverbrauch
		CPU-Last
Performance	Bereitstellung	Kompilieren
		Hochfahren

Abbildung 5.4: Bewertungskriterien Betrieb

5.4.1 Ressourcenverbrauch

Dieser Abschnitt befasst sich mit dem Ressourcenverbrauch des Systems. Betrachtet werden hierbei der Bedarf an Festplattenspeicher sowie der Bedarf an Rechnerressourcen zur Laufzeit.

Speicherbedarf

Eine Anwendung sollte möglichst wenig Speicherplatz für sich beanspruchen. Das begünstigt die Portabilität und Versionierbarkeit und verkürzt die Bereitstellungszeiten.

- **Größe der Artefakte**
  Wie groß sind die mit der jeweiligen Technologie erstellten Artefakte?
- **Größe des Gesamtsystems**
  Wie groß ist das jeweilige Gesamtsystem?

Ressourcenverbrauch zur Laufzeit

Je geringer der Ressourcenbedarf der Anwendung zur Laufzeit, umso kleiner sind die Anforderungen an das Zielsystem. Im Cloud-Bereich bedeutet dies weniger Kosten, da weniger Rechnerressourcen reserviert werden müssen.

5.4.2 Performance

Bei einer verteilten Anwendung, welche in einer Cloud-Infrastruktur betrieben wird, spielt der Durchsatz im Gegensatz zu Monolithen keine essentielle Rolle. Eine dementsprechend eingerichtete Cloud-Infrastruktur reagiert auf Lastspitzen, indem sie die betroffenen Komponenten horizontal skaliert. Somit ist die Bereitstellungszeit sowie die Dauer des Startens einer neuen Instanz bei einer NCA von größerer Bedeutung.

Bereitstellung

- **Dauer Kompilieren**
  Wie lange dauert die Erstellung einer Artefakte aus dem Quellcode?
- **Dauer Hoch- und Herunterfahren**
  Wie viel Zeit benötigt die Anwendung zum Hochfahren?

5.5 Sonstige Bewertungskriterien

5.5.1 Lizenzierung

- Unter welcher Lizenz wird die Technologie zur Verfügung gestellt?
- Kann sie in allen seinen Teilen kostenfrei benutzt werden?

5.5.2 Abhängigkeit zum Hersteller (Vendor Lock-In)

- Ist der Code proprietär oder *Open-Source*?
- Ist ein Migrieren bestehender Anwendungen auf andere Plattformen möglich?

6 Referenzanwendung

Für die Beurteilung der vorgestellten Technologien anhand des im Kapitel 5 beschriebenen Kriterienkatalogs ist es notwendig, eine Referenzanwendung mit den jeweiligen Technologien zu erstellen. Die Anwendung soll möglichst alle technischen Aspekte abdecken. Gleichzeitig sollte sie ausreichend generisch sein, um übertragbare, allgemein gültige Evaluierungsergebnisse zu liefern.

6.1 Fachliche Anforderungen

Die Referenzanwendung wird in Form eines Online-Shops erstellt. Eine der ersten Domänen, in der Microservices sowie Cloud-Native-Anwendungen im Produktivbetrieb eingesetzt wurden, ist der Online-Shop. So begann das deutsche Unternehmen *Otto Group* bereits im Jahr 2011 damit, die Entwicklung ihres Systems in Form von Microservices vorzunehmen[Ste15]. Zur Vermeidung von unnötiger, für die Evaluierung nicht relevanter Komplexität soll die zu erstellende Software im Wesentlichen zwei Anwendungsfälle abdecken: die Registrierung sowie den Bestellprozess. Für den ersten Use-Case - Registrierung - werden folgende *User-Stories* umgesetzt:

Als Administrator möchte ich einen neuen Nutzer anlegen können.
Als neu angemeldeter Nutzer möchte ich über die Neuanmeldung per Email informiert werden.
Als neu angemeldeter Nutzer möchte ich einen Neukundenrabatt erhalten.
Als neu angemeldeter Nutzer möchte ich mein Profil mit weiteren Daten wie Adresse vervollständigen können.

Für den zweiten Anwendungsfall - Bestellung - sollen folgende User-Stories umgesetzt werden:

Als angemeldeter Nutzer möchte ich mir den Warenbestand ansehen können.
Als angemeldeter Nutzer möchte ich mir den Preis der Waren abzüglich meines individuellen Rabatts ansehen können.
Als angemeldeter Nutzer möchte ich einen Artikel kaufen können.

Aus den oben genannten Szenarien lassen sich die fachlichen Komponenten ableiten: *Benutzerverwaltung - Artikelverwaltung - Rabattverwaltung - Nachrichten - Versand.* Obwohl eine produktiv einzusetzende Anwendung durchaus weitere Komponenten wie *Bestellungen* oder *Rechnungen* beinhalten müsste, wird zur Vermeidung unnötiger Komplexität auf diese verzichtet.

6.2 Technische Anforderungen

Die Komponenten werden in Form von Microservices erstellt. Die in 2.2.5 genannten Anforderungen nach *The-Twelve-Factor-App* sind nach Möglichkeit alle umzusetzen. So beinhaltet jeder Service eine eigene Datenbankinstanz, bietet seine Dienste über gesicherte ReST-Schnittstellen an, ist horizontal skalierbar und unterstützt asynchrone Kommunikation. Außerdem sollen in der Gesamtanwendung möglichst alle in 2.2.6 genannten Komponenten einer NCA umgesetzt werden.

6.3 Architektur

Aus den fachlichen und technischen Anforderungen ergeben sich die nachfolgenden Ablaufdiagramme:

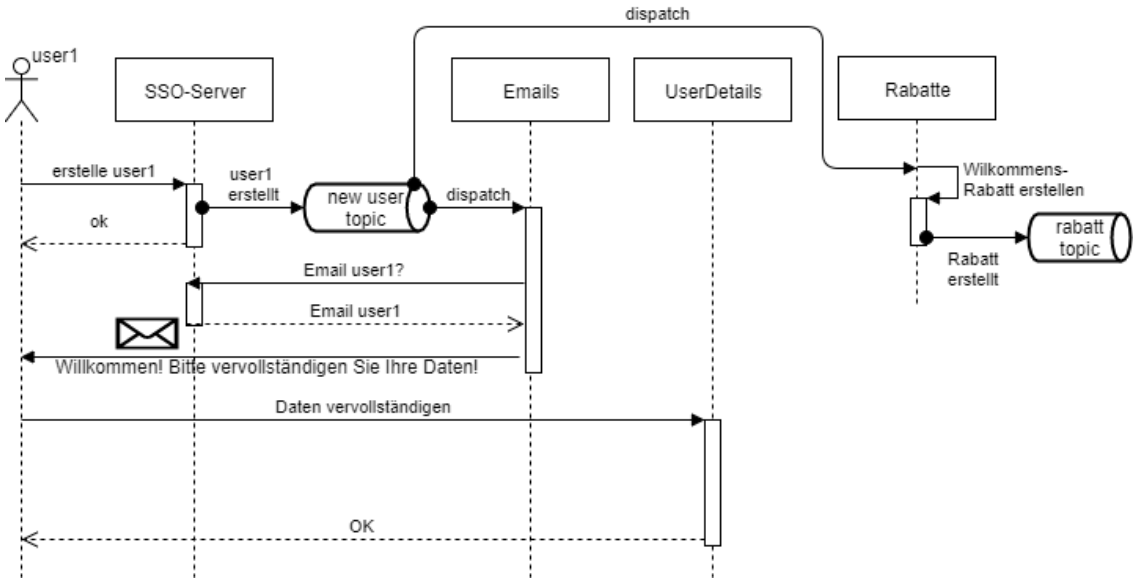

Abbildung 6.1: Ablauf Registrierung

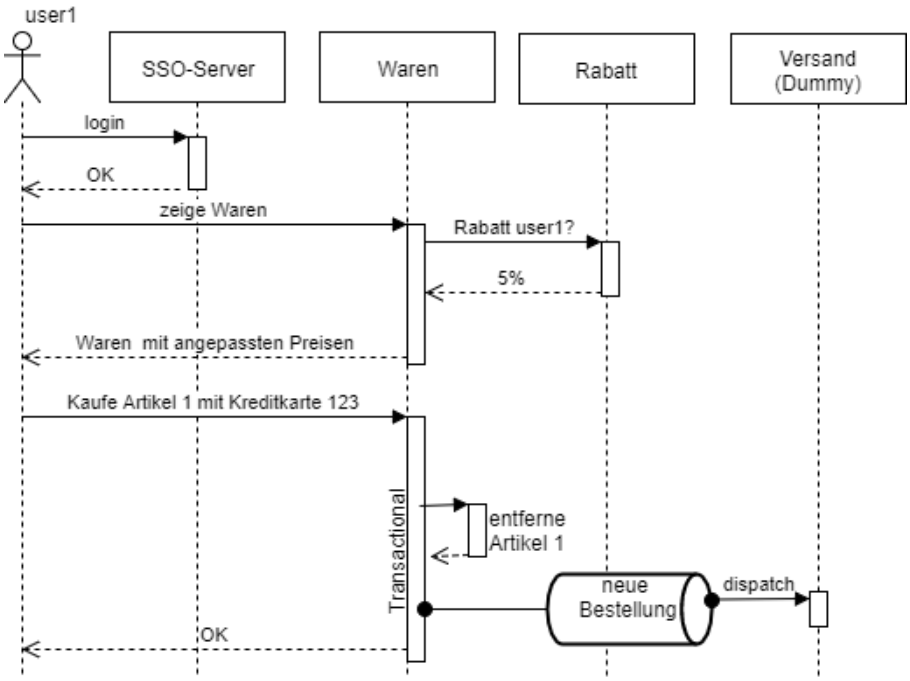

Abbildung 6.2: Ablauf Bestellung

Die einzelnen zu erstellenden Services sind:

Komponente	**Beschreibung**	**ReST-Schnittstellen**	**Messaging**
User-Details	Benutzerdaten für existierende Nutzer anzeigen und bearbeiten	GET /user/{userId} POST /user/{userId}	subscribe: "Neuer Benutzer"
Email	Benutzer via Email über Ereignisse informieren.	-	subscribe: "Neuer Benutzer"
Waren	Verfügbare Waren anzeigen und zum Verkauf anbieten	GET /waren/all POST /waren/buy/{id}	publish to: "Neue Bestellung"
Rabatt	Individuellen Rabatt berechnen und speichern.	GET /rabatt/{userId}	subscribe: "Neuer Benutzer"
Versand	Bestellung annehmen.	-	subscribe: "Neue Bestellung"

Um die Unterstützung von NoSQL-Datenbanken evaluieren zu können, soll mindestens ein Service, falls von der Technologie unterstützt, über eine NoSQL-Datenbank verfügen. Alle Services sollen so implementiert werden, dass sie skalierbar sind und Resilience-Patterns wie Circuit Breaker umsetzen. Die gesamte Anwendung bedarf weiterer *Nicht-operativer Komponenten*, um den in 2.2.6 festgelegten Anforderungen an verteilte Anwendungen zu genügen. Somit sieht die geforderte Gesamtarchitektur wie folgt aus:

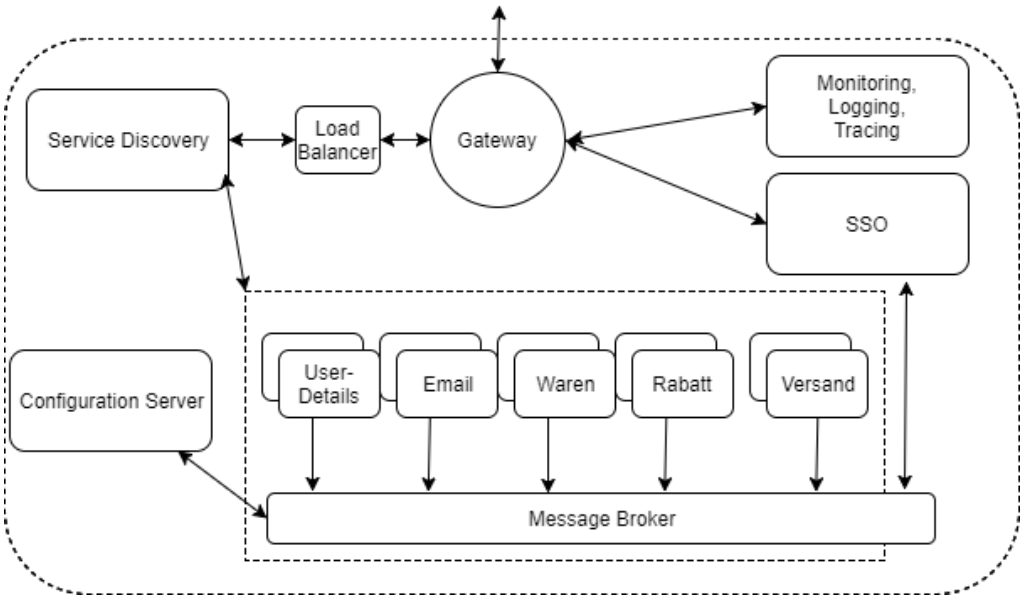

Abbildung 6.3: Makroarchitektur Referenzanwendung

Wie auf dem Bild ersichtlich werden Gateway, Load Balancing, Service Discovery, Single-Sign-On, Configuration Server und der Message Broker allen Komponenten zur Verfügung gestellt.

6.3.1 Anforderungen an das Testen

Um die im Abschnitt *Testbarkeit* genannten Aspekte bewerten zu können, soll die Anwendung ausreichend getestet werden. Für Demonstrationszwecke genügt es, pro Komponente exemplarisch mindestens einen Unit- und Integrationstest zu erstellen.

6.4 Umsetzung mit Spring Cloud

Die mit Spring Cloud erstellte Anwendung ergibt die nachfolgende Gesamtarchitektur:

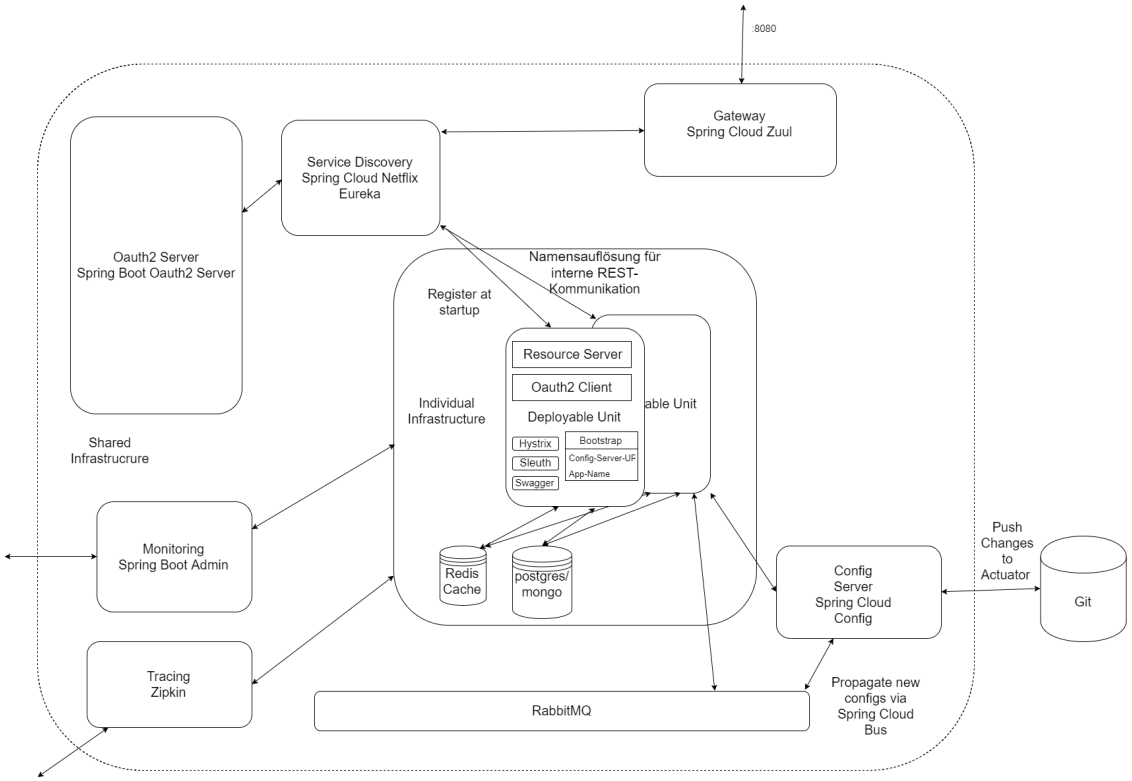

Abbildung 6.4: Architektur Spring Cloud

Als Unterstützung für die Umsetzung der Referenzanwendung wurde auf Anleitungen und Beispiele unter `spring.io` und [Par18a] zurückgegriffen. Durch den Einsatz von Profilen `default` und `cloud` ist die Anwendung in dieser Form sowohl in einer lokalen Umgebung als auch in Cloud Foundry lauffähig.

6.5 Umsetzung mit Java EE

6.5.1 Technologieentscheidung

Weil die Implementierung des Java EE-Standards in Verbindung mit Microprofile von mehreren Herstellern angeboten wird, musste eine bestimmte Technologie für die Umsetzung der Referenzanwendung ausgewählt werden. Die Auswahl betraf dabei *Thorntail, Open Liberty* und *KumuluzEE*. Es wurde gegen KumuluzEE entschieden, da dieses Projekt zum Zeitpunkt der Umsetzung den Microprofile-Standard in der nicht mehr aktuellen Version 1.2 unterstützte und nur den Java EE 7 - Standard implementierte[Kum18]. Red Hat Thorntail schien auf den ersten Blick eine gute Alternative zu sein. Dieser Eindruck wurde von der stellenweise knappen und unvollständigen Dokumentation sowie der komplizierten und uneinheitlichen Konfiguration getrübt[Red18]. Letztendlich wurde entschieden,

auf IBM Open Liberty zu setzen, da es Java EE 8 implementiert, von den drei Produkten am vollständigsten dokumentiert ist und die meisten Anleitungen bietet[IBM18i].

6.5.2 Architektur Referenzanwendung Java EE

Analog zum Kapitel 6.4 ergibt die Umsetzung mit Java EE in Verbindung mit Eclipse Microprofile und Open Liberty das nachfolgende Architekturmodell:

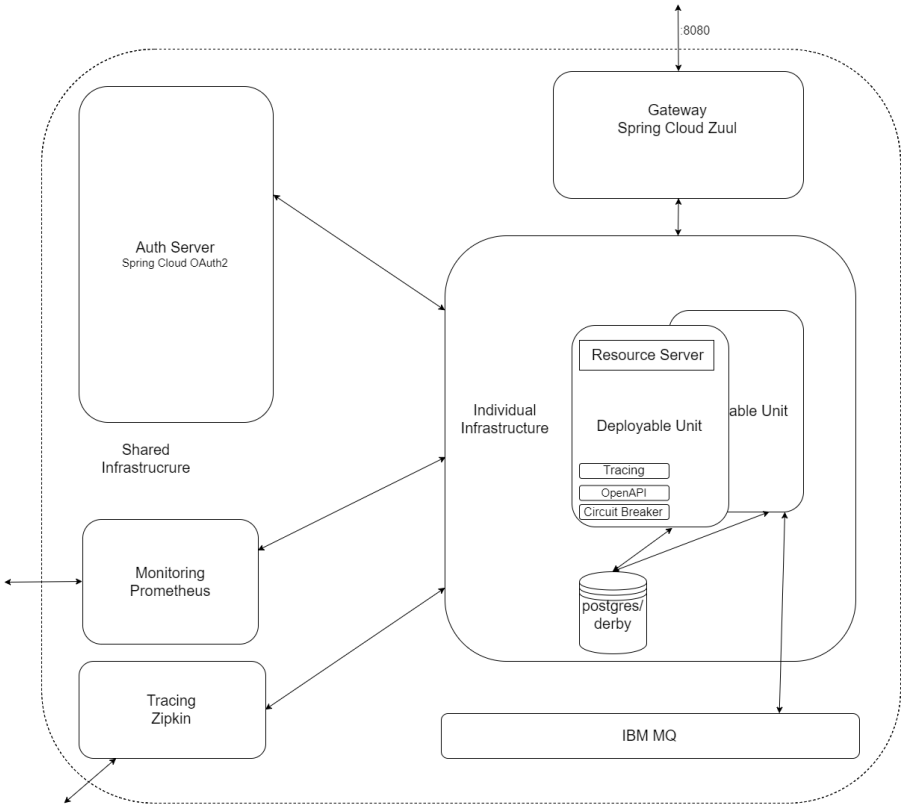

Abbildung 6.5: Architektur Java EE

Als Grundlage für die Implementierung der einzelnen Services wurde auf die unter [IBM18k] verfügbaren Anleitungen und Codebeispiele zurückgegriffen. *Edge Services* wie SSO-Server, Service Discovery, Configuration Server, API Gateway und Load Balancing konnten mit Java EE und Microprofile nicht umgesetzt werden. Um die Funktion des API Gateways und des Authentifizierungsservers zu ermöglichen, wurden die im Rahmen der mit Spring Cloud erstellten Referenzanwendung entstandenen Komponenten wiederverwendet. Weil der Microprofile-Standard keine Implementierung von Discovery-Clients bietet, ist die Funktionalität von Service Discovery auf Anwendungsebene nicht umsetzbar. Sie kann ggf. von der Cloud-Umgebung übernommen werden. Um Abhängigkeiten von einem bestimmten Softwareanbieter zu vermeiden wurden bei der Umsetzung nur die in Java EE und Microprofile enthaltenen Spezifikationen und keine proprietären Lösungen verwendet.

7 Evaluierung Spring Cloud

Nachfolgend wird die Erfüllung der im Kriterienkatalog aufgeführten Punkte durch Spring Cloud dokumentiert. Eine Bewertung der einzelnen Punkte erfolgt im direkten Vergleich mit Java EE im Kapitel 9.

7.1 Zuverlässigkeit

7.1.1 Dokumentation

Qualität und Umfang

Der offizielle Internetauftritt des Projekts, `www.spring.io`, beinhaltet eine umfangreiche Dokumentation. Diese beschreibt ausführlich alle Komponenten des Frameworks. Mit jeder Release-Version wird eine neue Dokumentation veröffentlicht - diese befindet sich unter [Piv18c] für Spring Cloud sowie unter [Bha18c] für Spring Boot. Sie ist übersichtlich, verständlich und umfangreich. In gedruckter Form würde die Dokumentation von Spring Boot 301 DIN-A4 Seiten umfassen. Die Dokumentation der aktuellen Version von Spring Cloud, `Finchley.SR1` würde 190 Seiten umfassen.

Anleitungen

Auf der offiziellen Internetseite existieren zahlreiche Schritt-für Schritt-Anleitungen, welche die Verwendung von Spring Cloud beschreiben.
Die von *Eugen Parasshiv* betreute Webseite `www.baeldung.org` beinhaltet weitere, von mehreren Softwareentwicklern verfasste Dokumentationen und Anleitungen rund um das Spring-Framework. Da die Artikel vor dem Verfassen korrekturgelesen werden sowie bestimmte Kriterien erfüllen müssen, ist ihre Qualität entsprechend hoch[Par18b].

Aktualität

Die offiziellen Dokumentationen sind stets auf dem aktuellen Stand. Dokumentationen nicht mehr aktueller Versionen können ebenfalls von den jeweiligen Projektseiten bezogen werden. Die älteste auf der offiziellen Seite verfügbare Dokumentation von Spring Boot ist für die Version 1.5.16. Bei Spring Cloud ist es die Dokumentation der 2017 veröffentlichten Version *Dalston.SR5*. Anleitungen werden in unregelmäßigen Abständen aktualisiert und basieren oft nicht mehr auf der aktuellsten Version des Frameworks. Da es sich hierbei nicht um einen essentiellen Teil der Dokumentation handelt, ist dieser Umstand zu verkraften. Trotz der vorkommenden Breaking Changes können nicht mehr aktuelle Tutorials mit einigen Anpassungen auf neuere Versionen angewandt werden.

7.1.2 Support

Community Support

Spring, Spring Boot und Spring Cloud sind weit verbreitete Technologien. Dementsprechend existiert eine große weltweite Community. Zu sämtlichen Themen finden sich Blog-Artikel und Anleitungen. Es ist nicht schwer, Hilfestellungen zu finden, sei es im Internet oder in Form von Printmedien.
Auf Stack Overflow existierten am 04.10.2018 44,940 gestellte Fragen zum Thema Spring Boot sowie 2,564 zum Thema Spring Cloud[Ove18d], wobei sich die meisten von ihnen in einer Untermenge von Fragen zu Spring Boot befinden. Weil die Dokumentationen umfangreich sind und insbesondere mögliche Fallstricke beschreiben, könnten die meisten auf Stack Overflow gestellten Fragen durch das Lesen der offiziellen Dokumentation beantwortet werden.

Enterprise Support

Pivotal Software, Inc, das Unternehmen hinter dem Spring Framework, bietet kommerziellen Support und Schulungen für Unternehmen an. Die Kosten werden auf Anfrage mitgeteilt[Piv18b].

7.1.3 Aktualität

Aktivität auf Git

Alle Projekte des Spring Frameworks sind quelloffen. Der Quellcode von Spring Cloud befindet sich unter `https://github.com/spring-cloud` und beinhaltet 92 Unterprojekte. In den meisten von ihnen wird täglich *commitet*, es existieren viele *Forks*. Jeder Freiwillige kann zur Entwicklung des Frameworks beitragen und eingereichte *Pull Requests* werden zeitnah bearbeitet.

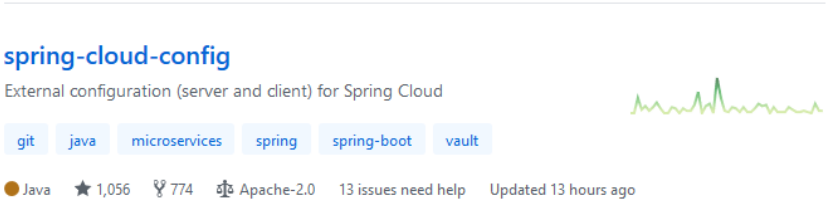

Abbildung 7.1: Aktivität auf GitHub am Beispiel Spring Cloud Config

Updates

Release-Zyklen

Bei Spring Cloud handelt es sich um ein Schirmprojekt, welches eine Untermenge von mehreren Projekten beinhaltet. Zur Vermeidung von Unklarheiten wegen unterschiedlicher Versionsnummern der Unterprojekte, werden die *Release Trains* von Spring Cloud nicht mit Versionsnummern, sondern mit Namen versehen. Dabei handelt es sich um

Namen der Londoner U-Bahn-Stationen. Sie werden in alphabetischer Reihenfolge vergeben[Piv18d]. Nachfolgend werden die Releases von Spring Cloud dargestellt[Piv18h]. Die Tabelle umfasst die Bezeichnung, die zum Betreiben benötigte Version von Spring Boot, das Datum der Veröffentlichung und das *End-Of-Life*-Datum - den Zeitpunkt, ab dem das Projekt nicht weiter betreut wird und keine Updates mehr bekommt.

Name	Spring Boot	Release	End Of Life
Angel	1.2.x	26.06.2015	07.2017
Brixton	1.3.x 1.4.x	11.05.2016	07.2017
Camden	1.4.x 1.5.x	28.06.2016	06.2018
Dalston	1.5.x	12.04.2017	12. 2018
Edgware	1.5.x	27.11.2017	08.2019
Finchley	2.0.x	19.06.2018	nicht festgelegt

Bug-Fixes

Bug Fixes und wichtige Änderungen innerhalb eines *Release-Trains* werden in Form von *Service-Releases* veröffentlicht. Service Releases erkennt man an der Endung *.SRX*, wobei *X* für die Versionsnummer des Releases steht[Piv18d]. Pro Version werden zwischen 4 und 7 Service Releases veröffentlicht[Mav18].

7.1.4 Kompatibilität

Abwärtskompatibilität

Jedes Release von Spring Cloud zieht mehrere Breaking Changes nach sich[Piv18h]. Die Release-Versionen bauen auf einer bestimmten Version von Spring Boot auf und sind nur für diese freigegeben. Ein Update von Spring Cloud zieht damit ein Update der darunterliegenden Version von Spring Boot nach sich.

7.2 Umfang

Sämtliche im Kapitel 2.2.6 genannte Anforderungen werden von Spring Cloud unterstützt und lassen sich mit geringem Aufwand umsetzen. Nachfolgend werden die jeweiligen Lösungen vorgestellt. Im Anhang [9.3] befinden sich Anleitungen, welche die Einrichtung der jeweiligen Komponenten beschreiben sowie auf Besonderheiten aufmerksam machen.

7.2.1 Verteilte und versionierte Konfiguration

Spring Cloud Config

Die Komponente *Spring Cloud Config* ermöglicht die Einrichtung eines Konfigurationsservers in Form einer Spring-Boot-Anwendung. Dieser stellt Konfigurationsdateien der

verwalteten Clients über eine ReST-Schnittstelle zur Verfügung. Sie werden dabei in Versionsverwaltungssystemen wie Git abgelegt. Es besteht die Möglichkeit, die darin enthaltenen Werte zu verschlüsseln. Änderungen in den Werten werden mithilfe des Projekts *Spring Cloud Bus* automatisch an die Clients verteilt[Lon17].
Um Konfigurationsdateien den entsprechenden Anwendungen eindeutig zuzuweisen, werden diese nach dem Muster `{application}-{profile}.{properties/yaml}` benannt.
Die innerhalb eines Profils für alle Anwendungen gültigen Einstellungen wie Zugangsdaten zu zentralen Services müssen dabei nicht redundant für jede Anwendung definiert werden. Sie können in einer allgemeinen Datei unter dem Namen `application-{profile}.properties` abgelegt werden[Piv18a].
Vom Konfigurationsserver bezogene Einstellungen überschreiben die Werte in lokal hinterlegten Konfigurationsdateien. Um eine erhöhte Stabilität zu erreichen, können in den Client-Anwendungen Rückfall-Konfigurationswerte hinterlegt werden, damit sie trotz einer eventuellen Nichterreichbarkeit des Konfigurationsservers ordnungsgemäß hochfahren können.

Automatische Benachrichtigung beim Aktualisieren den Konfigurationsdateien.
Das Projekt *Spring Cloud Bus* ermöglicht es, Änderungen in den Konfigurationsdateien an alle Clients zu verteilen. Die Verteilung erfolgt automatisch und bedarf außer einer Verbindung zu einem *Message-Broker* keiner weiteren Einstellungen[Bae18b]. Das Auslösen des Verteilungsvorgangs erfolgt mit dem Aufruf des Endpoints `/monitor` am Konfigurationsserver[Fen17]:

```
curl -v -X POST "http://localhost:8888/monitor"
    -H "Contepe:application/json"
    -H "X-Event-Key:repo:push"
    -H "X-Hook-UUID:webhook-uuid"
    -d '{"push":{"changes":[]}}'
```

Anbieter von Versionsverwaltungssystemen wie *GitHub* bieten die Funktion von *Webhooks* an - bei jeder Änderung in den verwalteten Dateien wird der oben dargestellte Aufruf ausgelöst.

7.2.2 Service Discovery

Spring Cloud bietet für die Unterstützung von *Service Discovery* eine Integration mit den Projekten *Netflix Eureka*, *Apache Zookeeper* und *Consul*. Aufgrund der größeren Bekanntheit wurde in der Referenzanwendung *Eureka* eingesetzt.

Eureka

Bei Eureka handelt es sich um eine Umsetzung von Service Discovery aus dem *Netflix-Stack*. Eureka verfügt über die im Kapitel 2.2.6 beschriebenen Funktionen und bietet darüber hinaus weitere Features wie *Load-Balancing* und *Failover*[Sim18].

7.2.3 Routing

Spring Cloud Zuul

Spring Cloud Zuul ist ein auf dem Projekt *Netflix Zuul* basierender Baustein von Spring Cloud und bietet unter anderem die Funktionalität eines API-Gateways. Das Gateway wird in Form einer Spring-Boot-Anwendung erstellt. In seinen Konfigurationsdateien werden die Regeln für die Weiterleitung der Anfragen festgelegt. So besagt beispielsweise die Regel `zuul.routes.service1 = service1`, dass Anfragen an die Route `/service1/` an die beim *Discovery Server* unter dem Namen `service1` registrierte Instanz weitergeleitet werden. Wenn in der verteilten Anwendung kein Discovery Server zum Einsatz kommt, können an dieser Stelle die URLs der jeweiligen Anwendungen eingetragen werden[Piv18f].

7.2.4 Load Balancing

Ribbon

Spring Cloud Zuul beinhaltet neben der Funktion des API-Gateways Client-seitiges Load-Balancing in Verbindung mit *Ribbon*. Dabei werden Anfragen nach dem *Round-Robin*-Prinzip an die bei Eureka registrierten Instanzen weitergeleitet. Dies geschieht automatisch und für den Endbenutzer nicht wahrnehmbar.

7.2.5 Resilience

Spring Cloud Netflix Hystrix

Spring Cloud bietet mit dem Modul *Spring Cloud Netflix Hystrix* eine Implementierung der im Kapitel 2.2.6 beschriebenen Patterns *Circuit Breaker, Timeouts und Bulkheads*. Das nachfolgende Beispiel demonstriert den Einsatz von Hystrix:

```
1  @HystrixCommand(fallbackMethod = "dummyUser")
2  public User getUser(String user) {
3          User userEntitiy = restTemplate.getForObject(
4                          HTTP_SSO_SERVER_USER + user, User.class);
5          return userEntitiy;
6  }
7
8  public User dummyUser() {
9          return new User("dummy", "no@where.com");
10 }
```

Dabei wird mithilfe eines ReST-Clients ein Objekt der Klasse `User` abgerufen. Falls der *Circuit Breaker* für die mit `@HystrixCommand(fallbackMethod = "dummyUser")` annotierte Methode aktiviert wurde, wird die *Fallback-Methode* `dummyUser()` aufgerufen, welche eine lokal vorenthaltene Instanz der Klasse `User` zurückgibt.

7.2.6 Persistenz

JPA mit Spring Data JPA

Spring bietet mit *Spring Data JPA* Unterstützung von relationalen Datenbanken mithilfe einer optimierten Form von *Java Persistence API* (JPA) und des ORM-Frameworks *Hibernate*. Der Zugriff auf in Form von *Entities* verwaltete Objekte erfolgt mithilfe von *Repositories*:

- **Entity**
  Die entsprechende Klasse wird mit `@Entity` annotiert. Ein Feld wird mithilfe der Annotation `@Id` als *Primary Key* deklariert.

```
1  @Entity
2  public class UserDetails {
3
4          @Id
5          private String userId;
6
7          private String firstName;
8
9          private String lastName;
10
11         //Getters and Setters...
12 }
```

- **Repository**
  Ein Repository wird in Form eines Interfaces erstellt:

```
1  @Repository
2  public interface UserRepository
3          extends JpaRepository<UserDetails, String> {
4  }
```

  Es enthält die Annotation `@Repository` und erbt von dem Interface `JpaRepository`. Von diesem erhält es zahlreiche Methoden zum Ausführen von geläufigen CRUD-Operationen. Die Argumente `<UserDetails, String>`, bezeichnen die Klasse der verwalteten *Entities* sowie das Format des *Primary Keys*. Die entsprechende Repository-Klasse wird zur Laufzeit erstellt und kann innerhalb der Anwendung injiziert werden.

Unterstützung von NoSQL-Datenbanken

Folgende NoSQL-Technologien werden von Spring Data unterstützt: *MongoDB, Neo4j, Elasticsearch, Solr, Cassandra, Couchbase* und *LDAP*. [Piv18e] Die Benutzung von NoSQL-Datenbanken orientiert sich an *Spring Data JPA*, sodass für die Verwendung von NoSQL in der Regel Kenntnisse von JPA ausreichen und spezielle Kenntnisse der jeweiligen Datenbanktechnologie für den simplen Einsatz nicht zwingend vorausgesetzt werden.

7.2.7 Caching

Spring Redis Cache

Mit *Spring Redis Cache* können Daten und Zustände außerhalb von Microservice-Instanzen gespeichert werden. Bei Redis handelt es sich um eine NoSQL-Datenbank, welche sich laut Hersteller insbesondere für Caching eignet.

7.2.8 Synchrone Kommunikation mit ReST

Bereitstellen von Services mithilfe von RestController

Rest-Schnittstellen lassen sich mithilfe vom Annotationen definieren:

```
1  @RestController
2  public class RabattRestController {
3          @Autowired
4          RabattRepository rabattRepository;
5
6          @RequestMapping("/rabatt/{userId}")
7          public Integer getRabatt(@PathVariable String userId) {
8                  Optional<Rabatt> rabattOptional =
9                          rabattRepository.findById(userId);
10                 return (rabattOptional.isPresent() ?
11                         rabattOptional.get().getPercent() : 0);
12         }
13 }
```

Konsumieren von Services mithilfe von RestTemplate

Mithilfe der Klasse `RestTemplate` werden ReST-Clients definiert:

```
1  User userEntitiy = restTemplate.
2          getForObject(HTTP_SSO_SERVER_USER + user, User.class);
```

7.2.9 Asynchrone Kommunikation mit Messaging

Spring Cloud Stream bietet ein eventgetriebenes Messaging-System mit Unterstützung von RabbitMQ und Apache Kafka.

Spring Cloud Stream mit RabbitMQ

Das Projekt *Spring Cloud Stream RabbitMQ* ermöglicht Verbindungen zu einem *Advanced Message Queuing Protocol* (AMQP)-Broker und bietet Abstraktionsklassen wie `MessageChannel` und `RabbitTemplate` zur Konfiguration des Nachrichtenaustauschs[Lon17].

7.2.10 Logging, Tracing und Monitoring

Logging

Den mithilfe von *Spring-Boot-Starters* erstellten Projekten wird automatisch die Abhängigkeit *Zero Configuration Logging* hinzugefügt. In der Standard-Einstellung wird dabei *Logback* als Logging-Framework verwendet[Lig18].

Tracing mit Spring Cloud Sleuth und Zipkin

Spring Cloud Sleuth protokolliert sämtliche Anfragen, welche in einer Anwendung ausgelöst werden und reichert diese mit weiteren Informationen an. Jeder Eintrag erhält dabei eine eindeutige `SpanID` und `TraceID`. Mit ihrer Hilfe lassen sich Requests über Anwendungsgrenzen hinaus protokollieren und nachverfolgen[Lon17].
Bei Zipkin bzw. *Openzipkin* handelt es sich um eine Anwendung, welche die von Sleuth erstellten Einträge empfängt, auswertet und in einer grafischen Benutzeroberfläche visualisiert. Mithilfe des Projekts *Spring Cloud Zipkin* werden die mit Sleuth generierten Einträge an eine bestehende Zipkin-Instanz weitergeleitet. *Spring Cloud Zipkin* beinhaltet sowohl Sleuth als auch den Client für die Kommunikation mit einem Zipkin-Server.

Monitoring mit Spring Boot Actuator und Spring Boot Admin

Mithilfe der Kombination aus *Spring Boot Actuator* und *Spring Boot Admin* wird Monitoring realisiert. *Spring Boot Actuator* stellt Statusdaten laufender Spring-Boot-Anwendungen über Schnittstellen bereit. Diese liefern Daten im JSON-Format und sind unter `/actuator/` abrufbar. *Spring Boot Admin* ruft die Actuator-Endpoints in regelmäßigen Abständen auf und stellt die Daten in einer grafischen Benutzeroberfläche dar[cod18]. Die benötigten Zugangsdaten der Anwendungen werden vom *Discovery Server* bezogen. Überwachte Anwendungen müssen für die Verwendung mit Spring Boot Admin nicht gesondert konfiguriert werden.

7.2.11 Security

Spring Cloud Security

Spring Cloud Security ermöglicht eine Integration von Sicherheitsaspekten in Microservices. Es bietet Interaktion mit Authentifizierungsservern und Absicherung von ReST-Schnittstellen[Lon17]. Spring Cloud Security kann mit einem bestehenden Authentifizierungsserver verwendet werden, bietet aber auch die Möglichkeit, einen eigenen Authentifizierungsserver zu erstellen.

Absicherung von Ressourcen

Mithilfe der Annotation `@PreAuthorize` werden Zugriffsrechte von Ressourcen verwaltet:

```
1 @PreAuthorize("#oauth2.isClient() or hasRole('ROLE_ADMIN')")
2 @RequestMapping("/rabatt/{userId}")
3 public Integer getRabatt(@PathVariable String userId) {
4     return 0;
5 }
```

7.3 Entwicklung

7.3.1 Aufwand

Lines Of Code

Eine funktionierende Webanwendung kann unter Spring Boot mit unter 30 Lines of Code erstellt werden[Piv18a]. Spring Boot setzt auf Annotationen und *Autoconfiguration*. Dies verringert die Konfiguration der Anwendungen auf ein Minimum. Die nachfolgende Tabelle beinhaltet die Anzahl der Codezeilen pro erstellter Anwendung:

Service	**Lines of Code**
Config-Server	14
Delivery-Service	37
Eureka	11
Gateway	13
Mail-Service	299
Rabatt-Service	247
SSO-Server	365
User-Service	271
Waren-Service	361

Abbildung 7.2: LoC in der Referenzanwendung

Konfiguration

Spring Boot ermöglicht eine externe Verwaltung von Konfigurationen. Damit kann die selbe Anwendung ohne Änderungen am Quellcode in unterschiedlichen Umgebungen betrieben werden. Die Einstellungen können dabei in Form von .property-Dateien, YAML-Dateien, Umgebungsvariablen und Kommandozeilenargumenten übergeben werden. Es existiert eine vordefinierte Konfigurationshierarchie. Nach dieser überschreiben sich die Werte abhängig von ihrer Quelle[Bha18b]

Konfigurationsdateien

Sämtliche Konfigurationen einer Spring-Anwendung lassen sich zentral in einer Konfigurationsdatei verwalten. Die Einstellungen werden in Form von *Key-Value*-Paaren in der Datei `application.properties` oder im .yaml - Format in der Datei `application.yml` gespeichert. Die Beschreibung aller Einstellungsmöglichkeiten befindet sich unter [Bha18a]. Neben den vordefinierten Werten können beliebige eigene Werte deklariert werden. Auf diese wird aus der Anwendung heraus mithilfe der Annotation `@Value`, der Abstraktion `Environment` oder mithilfe von `@ConfigurationProperties` zugegriffen[Bha18b]:

```
1 @Value("${security.oauth2.client.access-token-uri}")
2 private String accessTokenUri;
```

Konfiguration in Verbindung mit Profilen

Die Verwendung von Profilen erlaubt eine individuelle Konfiguration für unterschiedliche Umgebungen. So kann beispielsweise für die lokale Umgebung eine *In-Memory*-Datenbank verwendet werden und im Produktivbetrieb mit einer vorhandenen Datenbankinstanz verbunden werden. Eine Konfigurationsdatei für ein Profil wird nach dem Muster `application-{profile}.properties/yaml` erstellt.

Convention Over Configuration

Convention Over Configuration ist ein zentraler Aspekt von Spring Boot. Bei Verwendung der vom Framework als Standard angenommenen Konfigurationen muss im Idealfall nichts weiter konfiguriert werden. Die Entwickler können sich damit auf die eigentliche Anwendungsentwicklung statt auf die Konfiguration konzentrieren[Mü17].

7.3.2 Testbarkeit

Integrationstests benötigen den Zugang zum `AplicationContext`. Unit Tests werden dagegen so erstellt, dass dieser zum Ausführen nicht benötigt wird[Lon17].

Unit Tests

Da es sich bei Unit-Tests um Tests von einzelnen Java-Klassen handelt, wird keine spezielle Unterstützung vom Framework benötigt. Die zum Durchführen von Unit-Tests benötigten Abhängigkeiten, *JUnit, AssertJ, Hamcrest, Mockito, JSONAssert und JsonPath* werden beim Definieren der Abhängigkeit `spring-boot-starter-test` dem *Classpath* hinzugefügt[Piv18i]. Durch den ausgiebigen Einsatz von *Inversion Of Control* (IoC) wird das Erstellen von Unit Tests erleichtert, weil dadurch Abhängigkeiten leicht durch *Mocks* ersetzt werden können.

Integrationstests

Beim Erstellen eines Spring-Projekts mit *Spring Initializr* werden die zum Schreiben und Ausführen von Tests benötigten Abhängigkeiten mit dem Paket `spring-boot-starter-test` automatisch hinzugefügt und ein Integrationstest in der einfachsten Form automatisch erstellt. Das nachfolgende Beispiel aus [Lon17] zeigt einen solchen Integrationstest:

```
1  package demo;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.boot.test.context.SpringBootTest;
8  import org.springframework.context.ApplicationContext;
9  import org.springframework.test.context.junit4.SpringRunner;
10
11 @SpringBootTest
12 @RunWith(SpringRunner.class)
13 public class ApplicationContextTests {
14
15  @Autowired
16  private ApplicationContext applicationContext;
17
18  @Test
19  public void contextLoads() throws Throwable {
20   Assert.assertNotNull("the application context should have loaded.",
21    this.applicationContext);
22  }
23 }
```

Durch die Annotation `@SpringBootTest` wird die Testklasse zum Integrationstest, welche beim Ausführen den `ApplicationContext` lädt und die zu testende Anwendung startet. Die Annotation `@RunWith(SpringRunner.class)` ist eine Anweisung an das Test-Framework JUnit. Die Klasse `SpringRunner` stellt Testfunktionalitäten bereit, die zum Ausführen von Spring-Tests benötigt werden[Lon17]. Die Testmethode `contextLoads()` überprüft, ob der `ApplicationContext` erfolgreich geladen wurde.

7.3.3 Werkzeuge

Entwicklungsumgebung

Für Spring Boot und Spring Cloud existiert eine um zusätzliche Plugins erweiterte Version von Eclipse - *Spring Tool Siute* (STS). Sie bietet Unterstützung bei der Erstellung neuer Projekte durch die Integration von Spring Initializr. Daneben bietet STS eine Anbindung zu Cloud Foundry, welche es ermöglicht, die Anwendungen direkt aus der IDE heraus zu *pushen* und dort laufende Instanzen zu analysieren. Da es sich bei Spring-Projekten um von Maven bzw. Gradle verwaltete Java-Projekte handelt, ist die Verwendung der STS aber keine zwingende Voraussetzung.

7.4 Betrieb

Die nachfolgenden Messwerte wurden auf einem Laptop mit dem Prozessor Intel Core i5 6440HQ, 20 GB Arbeitsspeicher unter Microsoft Windows 10 festgestellt.

7.4.1 Ressourcenverbrauch

Speicherbedarf

Bei Spring Boot werden die Anwendungen in Form von *Uberjars* bereitgestellt. Sie beinhalten alle benötigte Abhängigkeiten sowie mit *Apache Tomcat* einen eingebetteten Webserver.

Service	**Größe in MB**
Config-Server	34,8
Delivery-Service	23.2
Eureka	42,6
Gateway	40,6
Mail-Service	51,3
Rabatt-Service	61,3
SSO-Server	61,6
User-Service	61.3
Waren-Service	63,3

Abbildung 7.3: Größe der Deployment-Artefakte in MB

Ressourcenverbrauch zur Laufzeit

Auswertung Ressourcenverbrauch

Laut den Auswertungen mithilfe des mit dem *Java SE Development Kit* (JDK) ausgelieferten Metrics-Tool *JConsole* verbrauchen die Komponenten die folgende Menge an Ressourcen zur Laufzeit im Ruhebetrieb:

	Heap in MB	**Non-Heap in MB**	**Threads**	**Klassen**	**CPU**
Config-Server	203	76	52	11152	0.2%
Delivery-Service	163	55	40	8218	0.1%
Eureka	108	76	68	11202	0.3%
Gateway	160	71	46	10798	0.1%
Mail-Service	400	87	53	13541	0.3%
Rabatt-Service	368	115	60	17966	0.5%
SSO-Server	163	105	55	16595	0.1%
User-Service	295	107	53	16576	0.1%
Waren-Service	307	105	65	16913	0.2%

Abbildung 7.4: Ressourcenverbrauch zur Laufzeit im Ruhezustand

7.4.2 Performance

Kompilieren

Nachfolgend wird die Dauer der Ausführung des *Maven-Goals* `mvn clean package` (ohne Testdurchführung) in Sekunden dargestellt:

Service	**Dauer**
Config-Server	3.29
Delivery-Service	3.17
Eureka	3.53
Gateway	3,31
Mail-Service	3,81
Rabatt-Service	4,23
SSO-Server	4,67
User-Service	4,24
Waren-Service	4.03

Abbildung 7.5: Dauer Kompilieren und Verpacken der Anwendungen

Starten

Nachfolgend wird die Dauer des Hochfahrens in Sekunden dargestellt:

Service	**Dauer**
Config-Server	8,5
Delivery-Service	7,7
Eureka	7,7
Gateway	9,48
Mail-Service	15,1
Rabatt-Service	22,5
SSO-Server	20,4
User-Service	27,8
Waren-Service	21,9

Abbildung 7.6: Dauer Starten der Anwendungen

7.5 Sonstige Bewertungskriterien

7.5.1 Lizenzierung

Spring Cloud ist mit der Lizenz *Apache Software License 2.0* lizenziert.

7.5.2 Abhängigkeit zum Hersteller (Vendor Lock-In)

Bei Spring handelt es sich um ein Framework. Eine Migration auf eine andere Technologie ist nicht möglich.

8 Evaluierung Java EE

Nachfolgend wird die Erfüllung der im Kriterienkatalog genannten Punkte beschrieben. Evaluierungsgrundlage ist dabei die auf Basis der Technologien Java EE 8 und Microprofile 2.0 mit Open Liberty erstellte Referenzanwendung[6.5.2].

8.1 Zuverlässigkeit

8.1.1 Dokumentation

Die Entwicklung von Cloud-Native-Microservices mit Java EE basiert auf drei Grundsteinen: den standardisierten Java EE-Schnittstellen, dem de-facto-Standard für Microservices Eclipse Microprofile, sowie der eingesetzten Servertechnologie - IBM Open Liberty[6.5.1]. Somit sind die nachfolgenden Evaluierungspunkte in Abschnitte *Java EE 8*, *Eclipse Microprofile* und *IBM Open Liberty* aufgeteilt. Während die ersten zwei Bausteine nicht austauschbar sind, kann anstelle von Open Liberty auf eine andere Implementierung von Java EE 8 und Microprofile wie *Thorntail* zurückgegriffen werden.

Qualität und Umfang

- **Java EE 8**
  Der offizielle Internetauftritt von *Oracle Software* umfasst eine umfangreiche Dokumentation der Spezifikationen und Schnittstellen. Vor dem Betrachten der im PDF-Format verfügbaren Dateien muss den von Oracle festgelegten Lizenzvereinbarungen zugestimmt werden. Die Dokumentationen sind anspruchsvoll geschrieben, sodass sie sich nicht zum schnellen Nachschlagen eignen[Ora18e].

- **Eclipse Microprofile**
  Die einzelnen Unterprojekte von Microprofile sind in den jeweiligen Repositories spärlich dokumentiert[Mic18]. Ausführlichere Beschreibungen finden sich auf den Portalen der jeweiligen Hersteller, welche Microprofile-konforme Application Server anbieten.

- **IBM Open Liberty**
  Auf der Internetseite von Open Liberty, `openliberty.io` sind die verwendeten Schnittstellen von Java EE und Microprofile dokumentiert. Ebenso sind die grundlegenden Funktionen des Liberty-Servers beschrieben[IBM18i]. Da Open Liberty und Websphere Liberty Profile auf dem selben technischen Kern basieren, kann ebenfalls auf vorhandene Dokumentationen von Websphere Liberty zurückgegriffen werden[Not17]. Ihre Qualität schwankt dabei von detailliert und umfangreich bis nicht vorhanden[IBM18n].

Anleitungen

- **Java EE 8**
  Oracle bietet unter `https://javaee.github.io/firstcup` Anleitungen zum Einstieg in die Programmierung gegen die Java EE 8 - Schnittstellen. In den Anleitungen wird der Application Server *Glassfish* als Referenzimplementierung verwendet.

- **Eclipse Microprofile**
  Wie bei der Dokumentation bietet Microprofile selbst keine Anleitungen zum Einsatz der Technologien, sondern verweist auf die jeweiligen Hersteller der Application-Server.

- **IBM Open Liberty**
  Auf der Internetseite von Open Liberty existierten am 10.10.2018 24 Schritt-für-Schritt-Anleitungen[IBM18k]. Einige davon sind interaktiv. Sie haben eine hohe Qualität und es ist ersichtlich, dass die Autoren sich bemühen, diese verständlich zu gestalten. Jede Anleitung beinhaltet den entsprechenden Quellcode. Weitere Anleitungen sind in Vorbereitung[IBM18j].

Aktualität

- **Java EE 8**
  Die offiziellen Dokumentationen und Anleitungen wurden mit dem Release von Java EE 8 veröffentlicht und wurden seitdem dem Anschein nach nicht weiter gepflegt[Ora18c].

- **IBM Open Liberty**
  Die Dokumentationen und insbesondere die Anleitungen auf `openliberty.io` sind aktuell. Da die Anleitungen in Form von GitHub-Projekten erstellt und verwaltet werden, ist es möglich, nachzuvollziehen, wie oft Anpassungen und Aktualisierungen stattfinden. In viele der Repositories wird nahezu täglich *commitet*[IBM18j].

8.1.2 Support

Community Support

- **Java EE 8**
  Bei Java EE handelt es sich um einen breit verbreiteten und eingesetzten Industriestandard. Dementsprechend groß ist die Anzahl der Fachliteratur, Artikel und Internetblogs. Auf Stack Overflow existierten am 10.10.2018 28187 Fragen zum Thema Java EE[Ove18a].

- **Eclipse Microprofile**
  Gerade im Bereich von Cloud-Native Microservices ist es ungewöhnlich, Java EE zu verwenden[Rö17]. Es gibt zwar durchaus eine Community und einzelne Blogs, dessen Autoren sich mit diesem Thema beschäftigen. Aber da dieser Bereich von Spring dominiert wird, ist hier mit deutlich weniger Unterstützung zu rechnen. Auf Stack Overflow existierten am 10.10.2018 22 Fragen mit dem *Tag* Microprofile[Ove18e].

- **IBM Open Liberty**
  Open Liberty basiert auf dem sich seit 2012 auf dem Markt befindenden Application Server Websphere Liberty Profile[Not17]. Dieser ist in Fachkreisen durchaus bekannt und wird von vielen Unternehmen produktiv eingesetzt. Auf Stack Overflow existierten zum 10.10.2018 1261 Fragen zum Thema *Websphere Liberty*[Ove18c] sowie 41 Fragen zu Open Liberty[Ove18b].

Enterprise Support

Für Open Liberty kann ein Upgrade auf das kostenpflichtige Produkt IBM Websphere Liberty Profile vollzogen werden. Für dieses ist kommerzieller Support durch IBM sowie Partner verfügbar[Psc17].

8.1.3 Aktualität

Aktivität auf Git

- **Java EE 8**
  Bei Java EE handelt es sich um kein Open-Source-Projekt. Damit kann diese Frage für Java EE nicht beantwortet werden.

- **Eclipse Microprofile**
  Microprofile hat keine besonders große Anzahl an aktiven Contributorn. Pro Unterprojekt sind es weniger als 30 Personen, von denen jeweils ca. 5 aktiv sind[Mic18].

- **IBM Open Liberty**
  Das Kernprojekt von Open Liberty hatte am 10.10.2018 154 Contributor. 76 von ihnen hatten mehr als 10 Commits. Es existierten 511 offene und 1732 geschlossene *Issues*[IBM18j]. Pro Woche erfolgen in Durchschnitt über 50 Commits[IBM18j].

Updates

- **Java EE 8**
  Updates im klassischen Sinne existieren bei Java EE nicht.

- **Eclipse Microprofile**
  In der nachfolgenden Tabelle werden die Versionen von Microprofile sowie die in dem jeweiligen Release enthaltenen Bestandteile aufgeführt[Ecl18a][Lit16]

  Die aktuellen Versionen sind 1.4 und 2.0. Microprofile 1.4 basiert auf Java EE 7, die Version 2.0 auf Java EE 8.

Version	Relase-Datum	Bestandteile
1.0	17.09.2016	JAX-RS 2.0, CDI 1.2, JSON-P 1.0
1.1	21.07.2017	Microprofile 1.0 + Config 1.0
1.2	30.09.2017	Microprofile 1.1 + Config 1.1, Fault Tolerance 1.0, Health Check 1.0, Metrics 1.0, JWT Propagation 1.0
1.3	03.01.2018	Microprofile 1.2 + Config 1.2, Metrics 1.1, Open API 1.0, Open Tracing 1.0, Rest Client 1.0
1.4	20.06.2018	Microprofile 1.3 + Config 1.3, Fault Tolerance 1.1, JWT Propagation 1.1, Open Tracing 1.2, Rest Client 1.1
2.0	20.06.2018	Microprofile 1.4 + CDI 2.0, JAX-RS 2.1, JSON-B 1.0, JSON-P 1.1

Abbildung 8.1: Releases von Eclipse Microprofile

- **IBM Open Liberty**
  Open Liberty basierte in der ersten Version auf IBM Websphere Liberty Profile 17.0.0.1 und übernahm dessen Versionsnummer. Seit der Veröffentlichung wird die gemeinsame technische Basis für beide Produkte unter dem Namen Open Liberty weiterentwickelt. Seit dem ersten Release wurden 4 neue Versionen veröffentlicht. Ab der Version 18.0.0.2 wird Java EE 8 unterstützt. Die aktuelle Version ist 18.0.0.3 [IBM18a]

8.1.4 Kompatibilität

Abwärtskompatibilität

- **Java EE 8**
  Java EE ist voll abwärtskompatibel, sodass der Umstieg von Java EE 7 auf Java EE 8 keine Schwierigkeiten mit sich bringen sollte[Bie18].

- **Eclipse Microprofile**
  Von Eclipse Microprofile existieren zwei Versionen - 1.x auf Basis von Java EE 7 und 2.x auf Basis von Java EE 8. Beide Versionen werden betreut und mit Updates versorgt[Saa18].

- **IBM Open Liberty**
  Open Liberty bleibt mit neueren Versionen abwärtskompatibel.

8.2 Umfang

Nachfolgend wird beschrieben, welche der im Kapitel 2.2.6 beschriebenen Komponenten einer NCA mit Microprofile 2.0 und Open Liberty umgesetzt werden können. Anleitungen zum Einsetzen dieser Technologien befinden sich unter [9.3] sowie [IBM18k].

8.2.1 Verteilte und versionierte Konfiguration

Verteilte und versionierte Konfiguration nach dem im Kapitel 2.2.6 beschriebenen Prinzip ist mit den eingesetzten Technologien nicht realisierbar. Es ist durchaus möglich, mithilfe von *Microprofile Config* und eigenen Anpassungen eine entfernte Dateiverwaltung wie *etcd* über einen Java-Client anzusprechen und zur Laufzeit zu nutzen. Das Äquivalent zu der von Spring Cloud gebotenen Funktionalität eines Konfigurationsservers - das Laden der zentralen Konfigurationsdatei `server.xml` aus einer entfernten Quelle beim initialen Starten der Anwendung ist nicht umsetzbar.

8.2.2 Service Discovery

Service Discovery wird von Microprofile nicht geboten[Mot18]. Diese Funktion kann in einer Cloud-Umgebung von der darunterliegenden Orchestrierungsplattform übernommen werden[Kub18].

8.2.3 Routing und Load Balancing

Eine Routing-Instanz kann mithilfe der verwendeten Technologien nicht umgesetzt werden. Bei Bedarf kann sie analog zu *Service Discovery* von der Infrastruktur bereitgestellt werden. In der Referenzanwendung wurde an dieser Stelle das API-Gateway *Spring Cloud Zuul* eingesetzt. Da ein Betrieb mit einer Service-Discovery-Instanz nicht umsetzbar ist, wurden die Weiterleitungsrouten statisch in die Konfigurationsdateien des Gateways eingetragen.

8.2.4 Resilience

Mithilfe von `Microprofile Fault Tolerance` können in Open Liberty Resilience-Patterns wie *Circuit Breaker* und *Fallback* umgesetzt werden:

```
1 @Fallback(fallbackMethod = "fallbackEmail")
2 @CircuitBreaker(requestVolumeThreshold = 20, failureRatio = 0.5, delay = 500)
3 public String getEmailAdressFromUser(String userId) {
4         //some code
5         return userEmail;
6 }
7 public String fallbackEmail(String userId) {
8         return "no@where.com";
9 }
```

In diesem Beispiel wird, wenn von 20 erfolgten Anfragen mindestens die Hälfte fehlschlägt, der Circuit Breaker für 500 Millisekunden geöffnet. Während dieser Zeit werden sämtliche Requests an die in der Annotation `@Fallback` festgelegte Fallback-Methode umgeleitet[IBM18l].

8.2.5 Persistenz

SQL mit JPA

JPA 2.2 ist Bestandteil von Java EE 8. Die Standardimplementierung von JPA bei Open Liberty ist Eclipselink[IBM18a]. Zur Verbindung mit einer Datenbank ist es notwendig, einen entsprechenden JDBC-Treiber herunterzuladen. Leider ist dazu das einfache Definieren der benötigten Abhängigkeit in der POM-Datei nicht ausreichend. Der entsprechende Treiber wird manuell heruntergeladen und dessen Standort in der Konfigurationsdatei angegeben. Es ist zwar möglich, mithilfe von Maven-Plugins die Dateien automatisiert herunterzuladen, dennoch wird dadurch die Abhängigkeitsverwaltung uneinheitlich und die Wartbarkeit des Projekts erschwert.

NoSQL mit MongoDB

Open Liberty unterstützt MongoDB in den Versionen 2.10.0 bis 2.14.2. Der benötigte Datenbanktreiber kann so wie bei JPA nicht über Maven bezogen werden, sondern wird manuell eingebunden[IBM18b]. Im Rahmen der erstellten Referenzanwendung ist es nicht gelungen, eine Verbindung zu einer bestehenden MongoDB-Instanz aufzubauen.

8.2.6 Caching

Open Liberty bietet mit `sessionDatabase-1.0` ein Feature zum Persistieren von Session-Daten in einer relationalen Datenbank. Darüber hinaus besteht mit `sessionCache-1.0` eine auf *JCache* basierende Lösung, welche ein standardisiertes, verteiltes *In-Memory-Caching* ermöglicht[Gui18].

8.2.7 Synchrone Kommunikation mit ReST

Anbieten von Ressourcen mit JAX-RS

JAX-RS sowie JSON-P sind Bestandteile von Microprofile und lassen sich vollständig einsetzen:

```
1  @Path("/{userId}")
2  public class UserDetailsEndpoint {
3
4    @Inject
5    UserService userService;
6
7    @GET
8    @Produces(MediaType.APPLICATION_JSON)
9    public Response getUserDetails(@PathParam("userId") String userId) {
10         UserDetails userDetails = userService.getUserByUserName(userId);
11         return (userDetails == null ? Response.status(404).build() :
12                         Response.ok(userDetails).build());
13   }
14 }
```

Konsumieren von Ressourcen mit RestClient

Mithilfe von *Microprofile Rest Client* kann ein Client zum Abrufen von entfernten Ressourcen erstellt werden[IBM18d].

```
1 @Dependent
2 @RegisterRestClient
3 public interface RabattRestClient {
4   @GET
5   @Produces(MediaType.TEXT_PLAIN)
6   public Integer getRabatt(@PathParam("userId") String userId);
7 }
```

Beschreiben der Schnittstellen mit OpenAPI

Mithilfe von *Microprofile OpenAPI 1.0* werden bestehende ReST-Schnittstellen dokumentiert. Das Projekt basiert auf *Swagger* und ähnelt diesem sowohl im Funktionsumfang als auch in der visuellen Aufmachung[Mag18].

8.2.8 Asynchrone Kommunikation mit Messaging

Java EE 8 bietet mit JMS 2.0 eine Spezifikation für Messaging[Dea13]. Es kann sowohl ein interner(*embedded*) Broker eingesetzt werden, als auch mit einer externen Instanz verbunden werden. Die Benutzung von Messaging erfolgt mittels *Message Driven Bean* (MDB), welche seit Java EE 7 vollständig über Annotationen konfiguriert werden[Dea13].

8.2.9 Logging, Tracing und Monitoring

Logging

In der Standardkonfiguration werden bei Open Liberty sämtliche Log-Einträge in eine zentrale, im Verzeichnis des Application Servers liegende Datei, `console.log`, geschrieben. Dieses Verhalten kann in der zentralen Konfigurationsdatei angepasst werden[IBM18g].

Tracing mit Microprofile Open Tracing und Zipkin

Tracing wird mithilfe der Unterstützung von *Microprofile Open Tracing* realisiert. Die Tracing-Daten lassen sich mit der Anwendung *OpenZipkin* visualisieren[IBM18f].

Monitoring mit Microprofile Metrics

Das Paket *Microprofile Metrics* stattet die Anwendungen mit dem Endpoint `/metrics` aus. Unter diesem lassen sich Status-Informationen abrufen. Dritthersteller-Software wie *Prometheus*[Rup17] oder *Graphana*[IBM18m] stellen diese Daten grafisch dar.

8.2.10 Security

Mithilfe von *Microprofile JWT 1.1* in Verbindung mit dem Liberty-Modul `App Security 3.0` lassen sich wesentliche *Security-Patterns* umsetzen[IBM18c]:

Absichern von Ressourcen

Durch den Einsatz der von Java EE 8 bereitgestellten Annotation `@RolesAllowed` ist es möglich, den Ressourcenzugriff nur für bestimmte Rollen zu erlauben. Aus dem Objekt `SecurityContext`, welcher bei einem Aufruf einer ReST-Schnittstelle ausgelesen wird, kann auf Attribute wie Benutzername und Rollen zugegriffen werden, um Zugriffsrechte anhand dieser Informationen feingranular zu vergeben.

Verifizieren von JWT-Tokens

Microprofile JWT 1.1 ermöglicht das Verifizieren von JWT-Tokens innerhalb einer Anwendung und unterstützt damit OpenID-Connect 1.0. Die Einstellungen zu der Verifikation der Token erfolgt innerhalb der Konfigurationsdatei `server.xml` nach dem folgenden Muster:

```
1  <mpJwt
2      userNameAttribute="user_name"
3      groupNameAttribute="authorities"
4      signatureAlgorithm="HS256"
5      sharedKey="123"
6      issuer="open-liberty"
7      id="jwtUserConsumer"
8      keyName="default"
9      audiences="oauth2-resource"
10 />
```

Es besteht die Möglichkeit, zur Laufzeit über die Klasse
`org.eclipse.microprofile.jwt.JsonWebToken`
auf die im Informationsteil des JWT-Tokens enthaltenen Daten zuzugreifen[IBM18c].

Authentifizierungsserver

Es besteht keine Möglichkeit zur Erstellung eines Authentifizierungsservers mit den verfügbaren Technologien. Bestehende Authentifizierungsserver wie *Red Hat Keycloack* oder *Spring Cloud Oauth2 Server* lassen sich aber zusammen mit *Microprofile JWT* betreiben.

8.3 Entwicklung

8.3.1 Aufwand

Die Entwicklung gegen die `javax`-*Application Programming Interface* (API) ist allgemein bekannt und braucht an dieser Stelle nicht gesondert betrachtet werden. Die von Microprofile bereitgestellten Zusatzfunktionen lassen sich ohne großen Aufwand einbinden und bereiten keine nennenswerten Probleme.

Lines Of Code

Die erstellten Anwendungen beinhalten keine signifikant großen Mengen an LoC. Seit Java EE 6 wird verstärkt auf den Einsatz von Annotationen zur Konfiguration gesetzt. Java EE 8 geht weiter in diese Richtung, sodass die meiste Konfiguration der Anwendung mithilfe von Annotationen erfolgen kann und kaum XML-Dateien benötigt werden.

Service	**Lines of Code**
Delivery-Service	34
Mail-Service	287
Rabatt-Service	222
User-Service	197
Waren-Service	265

Abbildung 8.2: LoC in der Referenzanwendung

Konfiguration

Microprofile Config

Das Projekt *Microprofile Config* bietet Funktionen für die Verwaltung von Konfigurationsdateien. Es ermöglicht das Beziehen von Konfigurationen aus unterschiedlichen Quellen wie Property-Dateien und Umgebungsvariablen. Bei Vorhandensein der selben Konfigurationen mit unterschiedlichen Werten wird eine Überschreibungshierarchie angewendet. Zusätzlich können Anwendungen bei Änderungen in den Konfigurationsdaten dynamisch benachrichtigt werden[IBM18h]. Microprofile Config wird voraussichtlich in Form von JSR 382 in die kommende Version von Jakarta EE aufgenommen[Ora18f].

Konfiguration der Anwendungsserver

Bei der Konfiguration der Application Server bietet jeder Hersteller eine eigene Lösung, welche je nach Produkt mit unterschiedlich großem Aufwand umzusetzen ist und unterschiedlich gut dokumentiert ist. Das Vornehmen der Konfigurationen kostet teilweise mehr Zeit als die eigentliche Anwendungsentwicklung. An dieser Stelle wäre ein gemeinsamer Standard wünschenswert.
Bei Open Liberty werden alle Konfigurationen des Anwendungsservers in einer zentralen Konfigurationsdatei, `server.xml` vorgenommen. Für diese Datei existiert ein über 30000 Zeilen umfassendes XML-Schema (XSD), welches alle vorgesehenen Konfigurati-

onsmöglichkeiten beinhaltet und beschreibt. Dies erleichtert das Arbeiten, weil dadurch Autovervollständigung und *Schema Validation* ermöglicht werden.

Convention Over Configuration

Einzelne Konfigurationsparameter besitzen *Default-Werte*, welche angenommen werden, wenn nichts anderes definiert wird. Das bereits beschriebene XML-Schema enthält die Beschreibung der jeweiligen Default-Werte.

8.3.2 Testbarkeit

Unit Tests

Unit Tests für die einzelnen Klassen können problemlos durchgeführt werden. Um Abhängigkeiten im Laufe der Tests durch Mocks ersetzen zu können, ist es empfehlenswert, Constructor- und Setter-Injection statt der Field-Injection zu verwenden.

Integrationstests

Integrationstests müssen mit vorhandenen Mitteln implementiert werden, da Open Liberty an dieser Stelle keine besondere Unterstützung anbietet.

8.3.3 Werkzeuge

Entwicklungsumgebung

Mit *Open Liberty Tools* bietet IBM eine Sammlung von Plugins für die Entwicklungsumgebung Eclipse. Sie ermöglichen unter anderem das Ausführen und *Debuggen* von Anwendungen aus der Entwicklungsumgebung heraus[IBM18a].

8.4 Betrieb

Die nachfolgenden Messwerte wurden auf einem Laptop mit dem Prozessor Intel Core i5 6440HQ, 20 GB Arbeitsspeicher unter Microsoft Windows 10 festgestellt.

8.4.1 Ressourcenverbrauch

Speicherbedarf

Bei der Ausführung von `mvn package` wird eine Anwendung erstellt, welche eine externe Instanz von Liberty benötigt. Der *Goal* `mvn install -P minify-runnable-package` erstellt dagegen eine unabhängige ausführbare *Uberjar*, welche sämtliche zur Laufzeit benötigte Komponenten beinhaltet[IBM18e]. Nachfolgend sind die Größen der jeweiligen auf diese Weise erstellten Artefakte dargestellt:

Service	**Speicherbedarf in MB**
Delivery-Service	54.6
Mail-Service	73.4
Rabatt-Service	86.8
User-Service	84.8
Waren-Service	88.6

Abbildung 8.3: Größe der Deployment-Artefakte in MB

Ressourcenverbrauch zur Laufzeit

Laut den Auswertungen mithilfe des mit dem JDK ausgelieferten Metrics-Tool *JConsole* verbrauchen die Komponenten folgende Mengen an Ressourcen zur Laufzeit im Ruhebetrieb:

	Heap in MB	**Non-Heap in MB**	**Threads**	**Klassen**	**CPU**
Delivery-Service	22	49	39	9673	0.2%
Mail-Service	182	25	48	11707	0.3%
Rabatt-Service	214	23	43	12528	0.2%
User-Service	104	25	45	13804	0.2%
Waren-Service	258	40	44	13584	0.6%

Abbildung 8.4: Ressourcenverbrauch zur Laufzeit im Ruhezustand

8.4.2 Performance

Bereitstellung

Kompilieren

Nachfolgend wird die Dauer der Ausführung des *Maven-Goals*
`mvn install -P minify-runnable-package` (ohne Testdurchführung) in Sekunden pro Anwendung dargestellt:

Service	**Dauer**
Delivery-Service	21.1
Mail-Service	25.3
Rabatt-Service	23.13
User-Service	22.2
Waren-Service	24.1

Abbildung 8.5: Dauer Kompilieren und Verpacken der Anwendungen

Starten

Nachfolgend wird die Dauer des Hochfahren in Sekunden pro Anwendung in einer lokalen Umgebung dargestellt:

Service	**Dauer**
Delivery-Service	9.47
Mail-Service	11.73
Rabatt-Service	11.06
User-Service	11.33
Waren-Service	11.27

Abbildung 8.6: Dauer Starten der Anwendungen

8.5 Sonstige Bewertungskriterien

8.5.1 Lizenzierung

- **JavaEE 8**
  Bei Java EE handelt es sich um eine Spezialisierung und keine konkrete Implementierung. Die einzelnen Implementierungen der Technologie unterliegen unterschiedlichen, von den jeweiligen Anbietern gewählten Lizenzarten.

- **Eclipse Microprofile**
  Eclipse Microprofile unterliegt mit *Apache Software Licence 2.0* einer weit verbreiteten und akzeptierten Lizenz für quelloffene Software[Ecl18b].

- **IBM Open Liberty**
  IBM Open Liberty ist mit der *Eclpise Public License 1.0* lizenziert. Durch diese Lizenzierung, welche ein schwaches *Copyleft* beinhaltet wird ermöglicht, dass das kostenpflichtige proprietäre Produkt IBM Websphere Liberty Profile auf dem Quellcode von Open Liberty basiert[GNU18].

8.5.2 Abhängigkeit zum Hersteller (Vendor Lock-In)

Eine mithilfe von Java EE und Microprofile erstellte Anwendung ist auf allen Implementierungen dieser Standards lauffähig. Die Konfiguration der Application Server und des Dependency-Managements unterscheidet sich von Anbieter zu Anbieter grundlegend und bedarf bei einer Migration einer Neuerstellung. Das kann sich insbesondere im fortgeschrittenen Projektstadium als aufwendig erweisen.

9 Vergleich

In diesem Kapitel wird die Erfüllung der Kriterien durch beide Technologien gegenübergestellt und die Vor- und Nachteile der jeweiligen Lösungen erläutert.

9.1 Vergleich nach Kriterienkatalog

Kriterium	Unterkriterium	Spring	Java EE	vergleichbar
Zuverlässigkeit	Dokumentation	x		
	Support	x		
	Aktualität			x
	Kompatibilität		x	
Umfang	Umfang	x		
Entwicklung	Aufwand	x		
	Testbarkeit	x		
	Werkzeuge			x
Betrieb	Ressourcenverbrauch			x
	Performance			x
Sonstiges	Lizenzierung			x
	Vendor-Lock-In		x	

Abbildung 9.1: Vergleich nach Kriterienkatalog

Zuverlässigkeit

Dokumentation

Die Projekte von Spring sind mit vielen Beispielen und Anleitungen dokumentiert.
Die Dokumentation von Microprofile ist weniger umfangreich und weist Lücken auf.
IBM bietet viele ansprechende und verständliche Anleitungen für Open Liberty, die Dokumentation ist dennoch nicht komplett vollständig.

Support

Das Spring Framework verfügt über eine große Community und es existieren viele Anlaufstellen bei Fragestellungen und Problemen. Es gibt Anleitungen zu nahezu jedem Bereich von Spring. Cloud Native Java EE ist dagegen ein weniger verbreitetes Thema, sodass mit kaum Community-Unterstützung in diesem Bereich zu rechnen ist. Die primäre Anlaufstelle für Anleitungen und Hilfestellungen ist hier die offizielle Dokumentation. Wenn ein Teilbereich in den offiziellen Dokumentationen und Anleitungen nicht ausreichend

beschrieben ist, gibt es im Gegensatz zu Spring kaum Möglichkeiten, sich bei anderen Quellen zu informieren. Kommerzieller Support kann für beide Technologien in Anspruch genommen werden.

Aktualität

In Spring Cloud finden neue Technologien schnell Einzug. Im Gegensatz zum Java EE-Standard, welcher behutsam und wohlüberlegt weiterentwickelt wird, entwickelt sich Spring sehr schnell. Die Anzahl der aktiven Entwickler bei Spring Cloud ist höher als bei Microprofile und Open Liberty. Dies ist insbesondere der größeren Bekanntheit, aber auch dem stattlichen Umfang der angebotenen Unterprojekte zu verdanken.
Eclipse Microprofile wird in regelmäßigen, vergleichsweise kurzen Abständen aktualisiert. So wurden in zwei Jahren nach dem ersten Release fünf neue Versionen des Standards herausgebracht und der Umfang der Technologie kontinuierlich erweitert.

Kompatibilität

Bei Java EE und Microprofile ist die Abwärtskompatibilität zu nahezu 100% gegeben. Laut [Bie18] dauert die Migration von Java EE 7 auf Java EE 8 drei Sekunden.
Bei Spring Cloud wird dagegen jedes neue Release für bestimmte (aktuellste) Versionen von Spring Boot freigegeben und von älteren Versionen nicht mehr unterstützt. Breaking Changes kommen vermehrt vor, sodass ein Update auf eine neuere Version kein triviales Unterfangen ist.

Umfang

Beim Umfang bietet Spring Cloud mit insgesamt 24 Unterprojekten[Piv18b] deutlich mehr als Microprofile. Es ist eine große Auswahl an Technologien vorhanden, welche sich mit minimalem Aufwand in bestehende Anwendungen einbinden lassen. Weil damit sämtliche Teile einer NCA abgedeckt werden, wird die Gesamtanwendung homogen und weniger von der Bereitstellungsplattform und Drittanbietern abhängig.
Microprofile bietet in der Version 2.0 12 Komponenten, welche die nötigen Bestandteile einer NCA umfassen[Saa18]. Obwohl damit viele wichtige Bereiche abgedeckt werden, lassen sich im Vergleich zu Spring Cloud Funktionen vermissen. So ist die Technologie auf das Erstellen von Clients spezialisiert. Server-Anwendungen wie Authentifizierungsserver können mit Microprofile nicht erstellt werden, es muss auf Lösungen von Drittanbietern zurückgegriffen werden. Die Gesamtanwendung wird dadurch uneinheitlich und erfordert mehr Konfigurations- und Wartungsaufwand. Da in der Vergangenheit neue Releases von Microprofile mit der Einführung neuer Komponenten verbunden waren[Saa18], ist damit zu rechnen, dass in der Zukunft weitere Bestandteile vorgestellt werden.

Verteilte und versionierte Konfiguration

Eine Erstellung eines Konfigurationsservers ist nur mithilfe von Spring Cloud möglich.

Service Discovery

Service Discovery ist nicht im Microprofile-Standard vorhanden. Weder ist es möglich, einen Discovery Server zu erstellen, noch die einzelnen Anwendungen um die Funktionalität eines Discovery Clients zu erweitern. Spring Cloud bietet dagegen mit *Eureka, Zookeeper und Consul* drei Technologien zur Auswahl[Piv18c], welche sich mitsamt ohne großen Aufwand integrieren lassen.

Routing und Load Balancing

Spring Cloud bietet mit *Spring Cloud Zuul* und *Spring Cloud Gateway* zwei Technologien zum Erstellen eines API-Gateways, welche sowohl Load Balancing als auch eine Interaktion mit Service Discovery beinhalten. Bei Java EE mit Microprofile existiert kein Äquivalent dazu.

Resilience

Beide Technologien bieten umfangreiche Möglichkeiten zum Einsetzen von Resilience-Patterns wie Circuit Breaker und Failover. Die Umsetzung gestaltet sich bei beiden Technologien unkompliziert und bietet viele Einstellungsmöglichkeiten. Die Einrichtung ist in beiden Fällen gut dokumentiert.

Persistenz

Spring erweitert mit *Spring Data JPA* die JPA-API von Java EE um viele nützliche Funktionen, welche das Arbeiten mit dieser Technologie erleichtern. So werden für Repositories Methoden bereitgestellt, welche die gängigsten vordefinierten CRUD-Operationen anbieten. Das ist bei reinem JPA in dieser Form nicht gegeben und muss manuell nachimplementiert werden, was einen zusätzlichen Zeitaufwand bedeutet sowie eine zusätzliche mögliche Fehlerquelle mit sich bringt.
Spring unterstützt mit Spring Data die gängigsten Arten von NoSQL-Technologien. Open Liberty unterstützt lediglich MongoDB in den mittlerweile nicht mehr aktuellen Versionen 2.10.0 bis 2.14.2. Wie im Kapitel 8 erwähnt, ist es im Rahmen der Erstellung der Referenzanwendung nicht gelungen, einen Verbindung zu einer MongoDB-Instanz nach dokumentierten Methoden herzustellen.

Caching

Beide Technologien bieten Caching an. Die Umsetzung ist gut dokumentiert und lässt sich problemlos einsetzen.

Synchrone Kommunikation mit ReST

Das Anbieten von Ressourcen über ReST-Schnittstellen funktioniert bei beiden Technologien problemlos. Die Schnittstellen werden mithilfe von Annotationen definiert und der Funktionsumfang ist in beiden Fällen vergleichbar.
Das Konsumieren von entfernten Ressourcen mittels eines Rest-Templates verläuft bei Spring einfacher und intuitiver. Dank einer möglichen Integration mit einem *Discovery*

Server können die Adressen der angefragten Ressourcen dynamisch aufgelöst werden. Die von Open Liberty angebotene Lösung, ein Rest-Template in Form eines Interfaces zu erstellen, funktioniert lediglich, wenn sich die angefragte Ressource auf dem selben Host wie der Client befindet. Trifft das nicht zu, muss das Template mithilfe eines `Builders` innerhalb der aufrufenden Klasse erstellt werden. Dadurch geht der Vorteil von Dependency Injection - *Inversion Of Control* verloren.

Asynchrone Kommunikation mit Messaging

Asynchrone Kommunikation in Form von Messaging wird von beiden Technologien zufriedenstellend unterstützt. Java EE 8 bietet mit *Message Driven Bean* (MDB) die elegantere Lösung zum Empfangen von Nachrichten. Diese lassen sich über Annotationen konfigurieren und sind auf Anwendungsebene technologieunabhängig. Die Einrichtung von Topics bedarf keiner zusätzlichen Konfigurationen im Gegensatz zu Spring Cloud und `RabbitTemplate`.

Logging, Tracing und Monitoring

Logging

Logging wird von beiden Technologien auf eine vergleichbare Art und Weise unterstützt.

Tracing

Die von IBM dokumentierte Einrichtung des von Open Liberty unterstützten Projekts Microprofile-Open-Tracing weicht vom gewöhnlichem Muster ab. So sind die benötigten Abhängigkeiten nicht in Form von Maven-Dependencies verfügbar, sondern müssen aus einer externen Quelle manuell heruntergeladen werden. Die Einrichtung von Tracing bei Spring Cloud ist dagegen auf die übliche Art und Weise vorzunehmen.

Monitoring

Beide Technologien lassen sich um einen Endpoint zum Abrufen von Statusdaten erweitern. Die unter den jeweiligen Endpoints bereitgestellten Daten sind umfangreich und geben viel Aufschluss über die Software und das darunterliegende System. Die Umsetzung ist in beiden Fällen gut gelungen. Die Daten lassen sich mit entsprechender Software visualisieren und auswerten.

Security

Spring Cloud Security bietet Lösungen zum Erstellen sowohl von Authentifizierungsservern als auch von dazugehörigen Clients. Wie sonst bei Spring üblich, ist die Einrichtung eines *OAuth2-Servers* unkompliziert und umfangreich dokumentiert. Zugriffsrechte werden mithilfe von Annotationen konfiguriert und es existieren zahlreiche Konfigurationsmöglichkeiten zum Erstellen von JWT-Tokens.
Bei Java EE ist lediglich die Erstellung von *Resource Servern* möglich - die Einrichtung eines Authentifizierungsservers ist nicht vorgesehen - hier muss auf externe Lösungen zurückgegriffen werden. Die Konfiguration der Verifizierung der JWT-Tokens erfolgt in der Konfigurationsdatei `server.xml` und erweist sich im Fall von Open Liberty einfacher

als bei Spring. Zu Beachten ist der Unterschied, dass bei Spring, falls *Spring Cloud Security* im Classpath vorhanden ist, sämtliche Schnittstellen automatisch abgesichert werden. Bei Java EE muss dagegen jede Ressource explizit abgesichert werden.

Entwicklung

Aufwand

Die Anzahl an Codezeilen in den beiden Projekten ist nahezu identisch. Das überrascht, da Spring Boot im Gegensatz zu Java EE dafür bekannt ist, mit sehr wenigen LoC auszukommen. Die komplette Konfiguration von Spring Cloud lässt sich in einer einzelnen zentralen Datei vornehmen. Die Konfigurationsmöglichkeiten sind alle in den entsprechenden Dokumentationen beschrieben. Es wird stark auf Convention Over Configuration gesetzt, sodass nur Werte angegeben werden müssen, welche von den Standards abweichen.
Open Liberty bietet mit der Datei `server.xml` ebenfalls eine zentrale Konfigurationsdatei für die Einstellungen des Application Servers. Diese Art der Configuration ist ein großer Schritt nach vorne im Vergleich zum *IBM Websphere Application Server*, welcher als Vorgänger von Liberty betrachtet werden kann. Dennoch müssen einzelne Konfigurationen in externen Konfigurationsdateien wie beispielsweise `persistence.xml` für die Konfiguration des JPA-Providers vorgenommen werden. Die Art der Konfiguration bei Liberty ist gut, die von Spring Cloud ist aber insgesamt einfacher und besser.

Testbarkeit

Dank dem Einsatz von Dependency Injection sind Unit Tests bei beiden Technologien gut durchführbar. Spring bietet sämtliche zur Testdurchführung benötigte Werkzeuge innerhalb einer Abhängigkeit an. Bei Open Liberty müssen diese einzeln hinzugefügt werden. Für die Durchführung von Integrationstests bietet Spring Unterstützung an. Bei Open Liberty gibt es keine speziellen Tools zur Erleichterung der Erstellung von Integrationstests.

Werkzeuge

Spring bietet mit der auf Eclipse basierenden IDE Spring Tool Suite nützliche Plugins zum Erstellen und verwalten von Spring-Boot- und Spring-Cloud-Projekten. Es wird eine Integration mit Spring Initialzr sowie mit Cloud Foundry geboten.
IBM liefert mit Open Liberty Tools ebenfalls Plugins für Eclipse, welche das Arbeiten mit der Technologie erleichtern. In beiden Fällen bieten diese Tools hilfreiche Unterstützung an. Die Plugins sind stabil und lassen keine Funktionen vermissen.

Betrieb

Ressourcenverbrauch

Die mit Open Liberty erstellten Artefakte sind minimal größer. Beide bewegen sich aber mit durchschnittlich 60 bzw. 70 MB in einem niedrigen Bereich, sodass dieser

Größenunterschied in der Praxis keinerlei Auswirkungen nach sich ziehen wird. Der Ressourcenverbrauch der Anwendungen zur Laufzeit befindet sich auf einem ähnlichem Niveau ohne signifikante Unterschiede.

Performance

Das Erstellen der fertigen Artefakte aus dem Quellcode ist bei Spring schneller. Während der Ausführung des entsprechenden Maven-Goals wird der eingebettete Open-Liberty-Server zum Installieren von Features mehrfach gestartet und heruntergefahren, sodass dieser Zeitunterschied auf diesem Umstand beruht. Das Starten der Anwendungen dauert bei Spring dagegen allesamt länger. Es gilt aber zu beachten, dass bei den im Rahmen der Referenzanwendung erstellten Anwendungen beim Startvorgang zwei zusätzliche Anfragen über das Netzwerk erfolgen - das Beziehen der Konfigurationsdateien vom Konfigurationsserver und das Anmelden der Instanz bei Eureka.

Sonstige Bewertungskriterien

Beide Technologien sind quelloffen und kostenfrei einsetzbar. Bei Spring ist man von der Technologie und ihrem Anbieter abhängig. Java EE und Microprofile sind dagegen Standards. Solange bei der Implementierung an diese gehalten wird und keine zusätzlichen Funktionen von Server-Herstellern und Drittanbietern eingesetzt werden, begibt man sich in keine Abhängigkeit von einem bestimmten Anbieter. Lediglich die Konfiguration der einzelnen Anwendungsserver ist proprietär und nicht ohne weiteres migrierbar.

9.2 Bewertung

Spring Cloud ist die bekanntere und breiter verbreitete Technologie. Es existiert eine beachtliche Community und eine große Menge an Fachbeiträgen und Anleitungen zu allen Bereichen des Frameworks. Die offizielle Dokumentation hat eine hohe Qualität und ist umfangreich. Neueste Technologien finden schnell Einzug und Updates sowie neue Releases werden in kurzen zeitlichen Abständen herausgebracht. Das Einsetzen der Technologie ist unkompliziert und die Konfiguration der mit Spring Cloud erstellten Anwendungen ist einheitlich und gut dokumentiert. Dadurch dass bei Spring Cloud eingeschränkt abwärtskompatible Versionsänderungen innerhalb kurzer Zeit veröffentlicht werden, ist es mitunter aufwendig, mit Spring Cloud erstellte Anwendungen aktuell zu halten. Außerdem begibt man sich beim Einsatz von Spring in Abhängigkeit vom Technologieanbieter, da es sich um ein Framework und keinen Standard handelt.

Die Erstellung von cloudfähigen Microservices mit Java EE und Microprofile ist ein nicht sonderlich verbreitetes Thema. Außer der offiziellen Dokumentationen und Anleitungen finden sich kaum Hilfestellungen bei evtl. auftretenden Problemen und Unklarheiten. Die Anzahl von angebotenen Technologien ist überschaubar und vielen von Spring Cloud bekannten Komponenten fehlt das äquivalente Gegenstück bei Java EE. Da aber regelmäßig Releases von Microprofile herausgebracht werden, welche neue Komponenten beinhalten, ist damit zu rechnen, dass mit der Zeit weitere Technologien Einzug in diesen Standard

halten werden. Die verfügbaren Komponenten sind mit dem untersuchten Anwendungsserver IBM Open Liberty unkompliziert umsetzbar. Der Application Server lässt sich zentral konfigurieren. Der Implementierungsaufwand ist dank Java EE 8 nicht viel größer als bei Spring Cloud. Die mit Open Liberty erstellten Anwendungen sind bezüglich Performance und Ressourcenverbrauch gleichauf mit Spring Cloud, teilweise übertreffen sie diesen. Der größte Vorteil von Java EE und Microprofile ist, dass es sich dabei nicht um Frameworks, sondern um Standards handelt. Diese werden von mehreren Herstellern von leichtgewichtigen Anwendungsservern unterstützt, sodass die Möglichkeit gegeben wird, die selbe Anwendung ohne Anpassungen am Quellcode innerhalb unterschiedlicher Servertechnologien zu betreiben. Lediglich die Konfiguration der einzelnen Application Server ist nicht einheitlich und verhindert eine vollständige Unabhängigkeit. Java EE ist voll abwärtskompatibel, sodass der Umstieg auf neuere Versionen mit keinem nennenswerten Migrationsaufwand verbunden ist.

9.3 Fazit und Handlungsempfehlungen

Bei Java EE und Microprofile überwiegen die Nachteile, sodass Spring Cloud allen voran wegen der großen Verbreitung und der Fülle an Komponenten das Mittel erster Wahl für das Cloud-Native Application Development bleibt. Insbesondere beim Einsatz von agilen Methoden, verbunden mit vielen zeitnahen Releases ist der Einsatz von Spring Cloud zu empfehlen. Falls man sich aber der Einschränkungen von Cloud Native Java EE bewusst ist, kann diese Technologie durchaus als eine mögliche Alternative betrachtet werden, insbesondere bei der Migration von bestehenden, nach dem Java EE-Standard erstellten Anwendungen und wenn das Hauptaugenmerk auf Herstellerunabhängigkeit und Abwärtskompatibilität liegt. Java EE punktet dagegen in anderen Bereichen abseits von Cloud-Native - so ist diese Technologie nach wie vor bei großen Geschäftsanwendungen dank der strikten Trennung von Anwendungscode und Serverkonfigurationen das Mittel der Wahl. Während bei Spring das Einführen eines Sicherheitsupdates der integrierten Servertechnologie ein Rekompilieren sämtlicher Anwendungen bedeutet, bedarf es bei Java EE nur einer Anpassung des Application Servers und keiner Rekonfiguration der einzelnen Anwendungen.

Literatur

[Aug17] Stephan Augusten. *Was sind Container?* 19. Jan. 2017. URL: https://www.dev-insider.de/was-sind-container-a-573872/.

[Bae18a] Baeldung. *Quick Intro to Spring Cloud Configuration*. 2018. URL: https://www.baeldung.com/spring-cloud-configuration.

[Bae18b] Baeldung. *Spring Cloud Bus*. 2018. URL: https://www.baeldung.com/spring-cloud-bus.

[Bha18a] Phillip Webb; Dave Syer; Josh Long; Stéphane Nicoll; Rob Winch; Andy Wilkinson; Marcel Overdijk; Christian Dupuis; Sébastien Deleuze; Michael Simons; Vedran Pavić; Jay Bryant; Madhura Bhave. *Spring Boot Common Application Properties*. 2018. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

[Bha18b] Phillip Webb; Dave Syer; Josh Long; Stéphane Nicoll; Rob Winch; Andy Wilkinson; Marcel Overdijk; Christian Dupuis; Sébastien Deleuze; Michael Simons; Vedran Pavić; Jay Bryant; Madhura Bhave. *Spring Boot Externalized Configuration*. 2018. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html.

[Bha18c] Phillip Webb; Dave Syer; Josh Long; Stéphane Nicoll; Rob Winch; Andy Wilkinson; Marcel Overdijk; Christian Dupuis; Sébastien Deleuze; Michael Simons; Vedran Pavić; Jay Bryant; Madhura Bhave. *Spring Boot Reference Guide*. 2018. URL: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/.

[Bie18] Adam Bien. *Microservices auf Java EE 8 migrieren*. 1. Apr. 2018. URL: https://entwickler.de/leseproben/microservices-java-ee-8-579830912.html.

[CNC18] CNCF. *CNCF Charter*. 15. Mai 2018. URL: https://www.cncf.io/about/charter/.

[cod18] codecentric. *Spring Boot Admin Project Readme*. 2018. URL: https://github.com/codecentric/spring-boot-admin.

[Coh09] Mike Cohn. *The Forgotten Layer of the Test Automation Pyramid*. 17. Dez. 2009. URL: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

[Dea13] Nigel Deakin. *What's New in JMS 2.0, Part One: Ease of Use*. 1. Mai 2013. URL: https://www.oracle.com/technetwork/articles/java/jms20-1947669.html.

[Del17] David Delabassee. *Opening Up Java EE - An Update*. 12. Sep. 2017. URL: https://blogs.oracle.com/theaquarium/opening-up-ee-update.

[Deu04] Alan Deutschman. *Inside the Mind of Jeff Bezos*. 8. Jan. 2004. URL: https://www.fastcompany.com/50106/inside-mind-jeff-bezos-5.

[Ecl18a] Eclipse. *Eclipse MicroProfile*. 2018. URL: https://projects.eclipse.org/projects/technology.microprofile/governance.

[Ecl18b] Eclipse. *Microprofile.io*. 2018. URL: https://microprofile.io/project/eclipse/microprofile.

[Eis16] Markus Eisele. *Modern Java EE Design Patterns*. 15. Jan. 2016.

[Fen17] Leo Feng. *Question: How to broadcast refresh through all clients without webhook*. 2017. URL: https://github.com/spring-cloud/spring-cloud-config/issues/733\#issuecomment-312424916.

[Fow14] Martin Fowler. *Circuit Breaker*. 2014. URL: https://martinfowler.com/bliki/CircuitBreaker.html.

[Fow15] Martin Fowler. *MicroservicePremium*. 13. Mai 2015. URL: https://martinfowler.com/bliki/MicroservicePremium.html.

[Fro17] Thilo Frotscher. *Java EE Microservices: Es geht auch ohne Spring Boot*. 2017. URL: https://jaxenter.de/java-ee-microservices-frotscher-60969.

[Fu17] Arron Fu. *7 Different Types of Cloud Computing Structures*. 3. Mai 2017. URL: https://www.uniprint.net/en/7-types-cloud-computing-structures/.

[GNU18] GNU. *Verschiedene Lizenzen und Kommentare*. 2018. URL: https://www.gnu.org/licenses/license-list.de.html.

[Gua18] Java EE Guardians. *Joint Community Open Letter on Java EE Naming and Packaging*. 2. Jan. 2018. URL: https://javaee-guardians.io/2018/01/02/joint-community-open-letter-on-java-ee-naming-and-packaging/.

[Gui18] Andy Guibert. *JCache session persistence*. 22. Mai 2018. URL: https://openliberty.io/blog/2018/03/22/distributed-in-memory-session-caching.html.

[IBM18a] IBM. *About Open Liberty*. 2018. URL: https://openliberty.io/about/.

[IBM18b] IBM. *Configuring MongoDB connectivity in Liberty*. 8. Okt. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SSAW57_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/twlp_mongodb_create.html.

[IBM18c] IBM. *Configuring the MicroProfile JSON Web Token*. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SSAW57_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/twlp_sec_json.html.

[IBM18d] IBM. *Consuming RESTful services with template interfaces*. 2018. URL: https://openliberty.io/guides/microprofile-rest-client.html.

[IBM18e] IBM. *Deploying and packaging applications*. 2018. URL: https://openliberty.io/guides/getting-started.html.

[IBM18f] IBM. *Enabling distributed tracing in microservices*. 2018. URL: https://openliberty.io/guides/microprofile-opentracing.html.

[IBM18g] IBM. *Logging and Trace*. 8. Okt. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/rwlp_logging.html.

[IBM18h] IBM. *MicroProfile Config API*. 8. Okt. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/cwlp_microprofile_overview.html.

[IBM18i] IBM. *Open Liberty Documentation*. 2018. URL: https://openliberty.io/docs/.

[IBM18j] IBM. *Open Liberty Github Projects*. 2018. URL: https://github.com/OpenLiberty.

[IBM18k] IBM. *Open Liberty Guides*. 2018. URL: https://openliberty.io/guides/.

[IBM18l] IBM. *Preventing repeated failed calls to microservices*. 2018. URL: https://openliberty.io/guides/circuit-breaker.html.

[IBM18m] IBM. *Providing metrics from a microservice*. 2018. URL: https://openliberty.io/guides/microprofile-metrics.html.

[IBM18n] IBM. *Websphere Liberty Documentation*. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SSAW57_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/cwlp_about.html.

[Joh18] Lonn Johnston. *Eclipse Foundation Unveils New Cloud Native Java Future with Jakarta EE*. 24. Apr. 2018. URL: https://jakarta.ee/news/2018/04/24/eclipse-foundation-unveils-new-cloud-native-java-future-with-jakarta-ee/.

[Koc18] Parminder Singh Kocher. *Microservices and Containers, First edition*. Addison-Wesley Professional, 2018, S. 304. ISBN: 978-0-13-459838-3.

[Kub18] Kubernetes. *Discovering services*. 27. Sep. 2018. URL: https://kubernetes.io/docs/concepts/services-networking/service/#discovering-services.

[Kum18] Kumuluz. *Kumuluz EE Online-Auftitt*. 2018. URL: https://ee.kumuluz.com/.

[Kö16] Simon Kölsch. "Consul: Service Discovery für den Microservices-Stack". In: *Microservices* (2016). URL: https://www.innoq.com/de/articles/2016/12/devops-service-discovery-with-consul/.

[Lar14] Magnus Larsson. *A first look at Spring Boot, is it time to leave XML based configuration behind?* 15. Apr. 2014. URL: http://callistaenterprise.se/blogg/teknik/2014/04/15/a-first-look-at-spring-boot/.

[Lig18] Andrea Ligios. *Logging in Spring Boot*. 4. Juli 2018. URL: https://www.baeldung.com/spring-boot-logging.

[Lit16] Mark Little. *To MicroProfile 1.0 and Beyond*. 17. Sep. 2016. URL: https://developer.jboss.org/blogs/mark.little/2016/09/17/to-microprofile-10-and-beyond.

[Lon17] Kenny Bastani; Josh Long. *Cloud Native Java*. O'Reilly Media, Inc., 25. Aug. 2017. Kap. 15. 645 S. ISBN: 978-1-4493-7464-8.

[Mag18] Arthur De Magalhaes. *Introducing MicroProfile OpenAPI 1.0*. 22. Mai 2018. URL: https://openliberty.io/blog/2018/05/22/microprofile-openapi-intro.html.

[Mav18] Maven. *Spring Cloud Dependencies*. 2018. URL: https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dependencies.

[MF14] James Lewis Martin Fowler. *Microservices - a definition of this new architectural term*. 25. März 2014. URL: https://martinfowler.com/articles/microservices.html.

[Mic18] Microprofile. *Microprofile Projekt auf GitHub*. 2018. URL: https://github.com/eclipse/microprofile.

[Mil18] Mike Milinkovich. *And The Name Is...* 26. Feb. 2018. URL: https://mmilinkov.wordpress.com/2018/02/26/and-the-name-is/.

[Min18] Piotr Minkowski. *Mastering Spring Cloud*. Packt Publishing, 26. Apr. 2018, S. 432. ISBN: 978-1-78847-543-3.

[Mot18] Michael Hofmann; Dominik Mohilo; Gabriela Motroc. *Jakarta EE sollte die Zusammenarbeit mit der Cloud Native Computing Foundation intensivieren*. 27. Juli 2018. URL: https://jaxenter.de/jakarta-ee-klartext-michael-hofmann-2-73351.

[Mü17] Melanie Müller. *Convention over Configuration in Spring Boot*. 2017. URL: https://blog.doubleslash.de/convention-over-configuration-in-spring-boot/.

[NIS11] NIST. *The NIST Definition of Cloud Computing*. Hrsg. von Peter Mell; Timothy Grance. 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[Not17] Alasdair Nottingham. *Open Sourcing Liberty*. 19. Sep. 2017. URL: https://www.openliberty.io/blog/2017/09/19/open-sourcing-liberty.html.

[Ora14] Oracle. *Distributed Multitiered Applications*. 2014. URL: https://docs.oracle.com/javaee/7/tutorial/overview003.htm.

[Ora18a] Oracle. *Introduction to API Gateway OAuth 2.0*. 2018. URL: https://docs.oracle.com/cd/E50612_01/doc.11122/oauth_guide/content/oauth_intro.html.

[Ora18b] Oracle. *Introduction to JCP FAQ*. 2018. URL: https://jcp.org/en/introduction/faq.

[Ora18c] Oracle. *Java EE at a Glance*. 2018. URL: https://www.oracle.com/technetwork/java/javaee/overview/index.html.

[Ora18d] Oracle. *Java EE Compatibility*. 2018. URL: https://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html.

[Ora18e] Oracle. *Java^{TM} EE Documentation*. 2018. URL: https://www.oracle.com/technetwork/java/javaee/documentation/index.html.

[Ora18f] Oracle. *JSR 382: Configuration API 1.0*. 2018. URL: https://www.jcp.org/en/jsr/detail?id=382.

[Ove18a] Stack Overflow. *Questions Tagged JavaEE*. 10. Okt. 2018. URL: https://stackoverflow.com/questions/tagged/java-ee.

[Ove18b] Stack Overflow. *Questions Tagged Open Liberty*. 2018. URL: https://stackoverflow.com/questions/tagged/open-liberty.

[Ove18c] Stack Overflow. *Questions Tagged Websphere Liberty*. 2018. URL: https://stackoverflow.com/questions/tagged/websphere-liberty.

[Ove18d] Stack Overflow. *Questions Tagged with Spring Boot*. 4. Okt. 2018. URL: https://stackoverflow.com/questions/tagged/spring-boot.

[Ove18e] Staclk Overflow. *Questions tagges Microprofle*. 10. Okt. 2018. URL: https://stackoverflow.com/questions/tagged/microprofile.

[Par18a] Eugen Paraschiv. *Spring Boot Starters*. 31. Mai 2018. URL: https://www.baeldung.com/spring-boot-starters.

[Par18b] Eugen Paraschiv. *Write for Baeldung*. 2018. URL: https://www.baeldung.com/contribution-guidelines.

[Piv18a] Pivotal. *Building a RESTful Web Service*. 2018. URL: https://spring.io/guides/gs/rest-service/.

[piv18a] pivotal. *Spring Boot Getting Started*. 2018. URL: https://spring.io/guides/gs/spring-boot/.

[Piv18a] Pivotal. *Spring Cloud Config Dokumentation*. 2018. URL: https://cloud.spring.io/spring-cloud-config/multi/multi_spring_cloud_config_server.html.

[Piv18b] Pivotal. *Commertial Support For Spring*. 2018. URL: https://pivotal.io/contact/spring-support.

[piv18b] pivotal. *Spring Boot Project Page*. 2018. URL: https://spring.io/projects/spring-boot.

[Piv18b] Pivotal. *Spring Cloud Getting Started*. 2018. URL: http://projects.spring.io/spring-cloud.

[Piv18c] Pivotal. *RabbitMQ Documentation*. 2018. URL: https://www.rabbitmq.com/documentation.html.

[Piv18c] Pivotal. *Spring Cloud Offizielle Dokumentation*. 2018. URL: http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html.

[Piv18d] Pivotal. *RabbitMQ Tutorial One Spring AMQP*. 2018. URL: https://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html.

[Piv18d] Pivotal. *Spring Cloud Release Trains*. 2018. URL: http://projects.spring.io/spring-cloud/\#release-trains.

[Piv18e] Pivotal. *RabbitMQ Tutorial Three Spring AMPQ*. 2018. URL: https://www.rabbitmq.com/tutorials/tutorial-three-spring-amqp.html.

[Piv18e] Pivotal. *Working with NoSQL Technologies*. 2018. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-nosql.html.

[Piv18f] Pivotal. *Router and Filter: Zuul*. 2018. URL: http://cloud.spring.io/spring-cloud-netflix/multi/multi__router_and_filter_zuul.html.

[Piv18g] Pivotal. *Service Discovery Eureka Clients*. 2018. URL: http://cloud.spring.io/spring-cloud-netflix/multi/multi__service_discovery_eureka_clients.html.

[Piv18h] Pivotal. *Spring Blog Releases*. 2018. URL: https://spring.io/blog/category/releases.

[Piv18i] Pivotal. *Spring Boot Features: Testing*. 2018. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html.

[Psc17] Pavel Pscheidl. *OpenLiberty.io: Java EE Microservices Done Right*. 22. Okt. 2017. URL: https://dzone.com/articles/openlibertyio-java-ee-microservices-done-right.

[Red18] RedHat. *Thorntail 2.0 Documentation*. 2018. URL: https://docs.thorntail.io/2.2.0.Final/.

[Rot14] Gregor Roth. *Stability patterns applied in a RESTful architecture*. 13. Okt. 2014. URL: https://www.javaworld.com/article/2824163/application-performance/stability-patterns-applied-in-a-restful-architecture.html?page=3.

[Rou16] Margaret Rouse. *Container as a Service (CaaS)*. 1. Dez. 2016. URL: https://www.searchdatacenter.de/definition/Containers-as-a-Service-CaaS.

[Rup17] Heiko Rupp. *Monitoring an Eclipse MicroProfile 1.2 Server With Prometheus*. 1. Nov. 2017. URL: https://dzone.com/articles/monitoring-an-eclipse-microprofile-12-server-with.

[Rö17] Lars Röwekamp. *Microservices mit Java EE: Alptraum oder Dreamteam?* 14. Feb. 2017. URL: https://www.informatik-aktuell.de/entwicklung/programmiersprachen/microservices-mit-java-ee-alptraum-oder-dreamteam.html.

[Rö18] Lars Röwekamp. *Enterprise Tales: MicroProfile – der alternative Standard*. 2018. URL: https://jaxenter.de/enterprise-tales-microprofile-66411.

[Saa18] Cesar Saavedra. *Eclipse MicroProfile 1.4 and 2.0 are Now Available*. 28. Juni 2018. URL: https://microprofile.io/2018/06/28/eclipse-microprofile-1-4-and-2-0-are-now-available/.

[Sha17] Linda DeMichiel; Bill Shanon. *Java EE 8 Specifications*. 31. Juli 2017. URL: https://github.com/javaee/javaee-spec/blob/master/download/JavaEE8_Platform_Spec_FinalRelease.pdf.

[Sim18] Michael Simons. *Spring Boot 2*. dpunkt, 16. Mai 2018, S. 460. ISBN: 978-3-86490-525-4.

[Ste15] Guido Steinacker. *Von Monolithen und Microservices*. 2. Juni 2015. URL: https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html.

[Ste17] Andrew S. Tannenbaum;Maarten van Steen. *Distributed Systems - Priciples and Paradigms*. 2017.

[Sti15] Matt Stine. *Migrating To Cloud-Native Application Architectures*. OReilly, 2015.

[Til15] Stefan Tilkov. *Don't Start With A Monolith*. 9. Juni 2015. URL: https://martinfowler.com/articles/dont-start-monolith.html.

[Wig17] Adam Wiggins. *The-Twelve-Factor-App*. 2017. URL: https://12factor.net/de/.

[Wig18] Adam Wiggins. *About Adam Wiggins*. 2018. URL: http://about.adamwiggins.com.

[Wik18] Wikipedia. *Java EE*. Hrsg. von Wikipedia. 2018. URL: https://de.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition.

[Wol15a] Stefan Tilkov; Martin Eigenbrodt; Silvia Schreier; Oliver Wolf. *REST und HTTP, 3rd Edition*. 2015.

[Wol15b] Eberhard Wolff. *Microservices*. dpunkt, 2015, S. 386. ISBN: 978-3-86490-313-7.

Abbildungsverzeichnis

Anhang 1: Einrichtung Spring Cloud

Nachfolgend wird die Einrichtung von Komponenten einer NCA mithilfe von Spring Cloud beschrieben.

Spring Cloud Config

Einrichtung Konfigurationsserver

Der Konfigurationsserver wird in Form einer Spring-Boot-Anwendung erstellt. Um diesen in einer minimalen, lauffähigen Form zu erstellen, bedarf es der nachfolgenden Schritte:

Abhängigkeiten	`org.springframework.cloud:` `  spring-cloud-config-server`

application.properties	`spring.application.name = service1`
	`spring.cloud.config.server.git.uri =`
	`  http://github.com/path/to/repo.git`

Annotationen	`@EnableConfigServer`

In der Konfigurationsdatei `application.properties` muss unter `spring.cloud.config.server.git.uri` die Adresse des Repositories mit den Konfigurationsdateien angegeben werden.

Einrichtung Konfigurationsclient

Um einer Anwendung das Beziehen der Konfigurationsdaten vom Konfigurationsserver zu ermöglichen, muss diese entsprechend eingerichtet werden:

Dependencies	`org.springframework.cloud:` `  spring-cloud-starter-config`
bootstrap.properties	`spring.application.name = service1` `spring.cloud.config.uri = http://localhost:8888`
Annotationen	`@EnableAutoConfiguration`

Die Konfigurationsdateien beinhalten Informationen, welche zum ordnungsgemäßen Hochfahren der Anwendung notwendig sind. Aus diesem Grund müssen Konfigurationsdateien beim *Bootstrapping* geladen werden. Die URL des Konfigurationsservers wird dazu in der Datei `bootstrap.properties` hinterlegt, welche vor der Datei `application.properties` ausgelesen wird. Damit die Anwendung die für sie bestimmte Konfiguration abholen kann, muss hier ebenfalls der Anwendungsname hinterlegt sein. [Bae18a]

Spring Cloud Eureka

Einrichtung Eureka-Server

Eine Eureka-Server-Instanz wird in Form einer Spring-Boot-Anwendung erstellt. Um diese in einer minimalen, lauffähigen Form zu erstellen, bedarf es der nachfolgenden Konfiguration:

Dependencies	`org.springframework.cloud:` `spring-cloud-netflix-eureka-server`
application.properties	`eureka.client.register-with-eureka = false` `eureka.client.fetch-registry = false`
Annotationen	`@EnableEurekaServer`

Einrichtung Eureka-Clients

Damit eine Anwendung mit Eureka interagieren kann, muss diese als *Discovery Client* registriert werden:

Dependencies	`org.springframework.cloud:` `spring-cloud-netflix-eureka-client`
application.properties	`eureka.client.serviceUrl.defaultZone =` http://localhost:8761/eureka
Annotationen	`@EnableDiscoveryClient`

Standardmäßig versucht ein Discovery-Client, eine Verbindung mit dem Eureka-Server unter der Adresse `localhost:8761` zu verbinden. Sollte der Standort des Eureka-Servers von diesem *Default-Wert* abweichen, kann dieser in der Konfigurationsdatei der Anwendung entsprechend angepasst werden.

Besonderheiten im lokalen Betrieb

Für den lokalen Einsatz von Eureka sind die nachfolgenden Besonderheiten zu beachten: Für die horizontale Skalierbarkeit der Anwendungen, welche auf dem selben *Host* betrieben werden, ist es notwendig, dass ihre Instanzen auf unterschiedlichen Ports laufen. Dies wird durch das Setzen des Arguments `server.port` in der `application.properties` auf 0 erreicht - mit diesem Argument starten die Instanzen auf einem beliebigen freien Port. Dies zieht allerdings den Effekt nach sich, dass die IDs der Instanzen beim Anmelden überschrieben werden und Eureka immer nur den zuletzt angemeldeten *Node* kennt. Das liegt daran dass die Instanzen mit der nach dem Muster
{`spring.cloud.client.hostname`}:
{`spring.application.name`}:{`spring.application.instance_id`}:{`server.port`}
erstellten ID registriert werden[Piv18g], wobei der Wert `server.port` aus der Konfigurationsdatei der Anwendung entnommen wird und bei jeder Instanz in diesem Falle `0` beträgt. Um dies zu verhindern kann der nachfolgende unter [Piv18g] beschriebene Trick eingesetzt werden: Dem Instanznamen wird ein Zufallswert hinzugefügt und damit sichergestellt, dass die Namen eindeutig bleiben:

```
1 eureka.instance.instanceId=
2 ${spring.application.name\}:${spring.application.instance_id:${random.value}}
```

Besonderheiten Betrieb in Cloud Foundry

Für den Betrieb in Cloud Foundry müssen in den Konfigurationsdateien der *Clients* die nachfolgenden Attribute definiert werden:

```
1 eureka.instance.hostname=${vcap.application.uris[0]}
2 eureka.instance.nonSecurePort=80
```

Damit wird erreicht, dass die *Clients* unter der in der Variablen `vcap.application.uris[0]` gespeicherten globalen Adresse auf dem von Cloud Foundry standardmäßig freigegebenen Port `80` aufgelöst werden.[Piv18g]

Spring Cloud Zuul

Einrichtung Gateway

Der Gateway wird in Form einer Spring-Boot-Anwendung erstellt. Um diesen in einer minimalen, lauffähigen Form zu erstellen, bedarf es der nachfolgenden Schritte:

Abhängigkeiten	`org.springframework.cloud:` `spring-cloud-starter-netflix-zuul`
application.properties	`zuul.routes.service1 = service1`
Annotationen	`@EnableZuulProxy`

Diese Beispielkonfiguration legt fest, dass alle Anfragen an die Route `/service1/` an die bei dem *Discovery-Server* unter dem Namen `service1` gespeicherte Instanz weitergeleitet werden sollen.
Wenn in der verteilten Anwendung kein *Discovery-Server* eingesetzt wird, kann an dieser Stelle direkt die URL der jeweiligen Anwendung eingetragen werden.
Über die Funktion des Gateway hinaus bietet *Spring Cloud Zuul* weitere, für den Betrieb von NCAs nützliche Funktionen. Eine ausführliche Dokumentation des Projekts befindet sich unter [Piv18f]

Spring Cloud Netflix Hystrix

Einrichtung Hystrix-Client

Um die Funktionen von *Spring Cloud Netflix Hystrix* nutzen zu können, müssen die nachfolgenden Schritte gemacht werden:

Abhängigkeiten	`org.springframework.cloud:` `spring-cloud-starter-netflix-hystrix`

Spring Data JPA

Um die Unterstützung von JPA mithilfe von Spring Data umzusetzen bedarf es der nachfolgenden Schritte:

- **Abhängigkeiten**

Abhängigkeiten	`org.springframework.boot:` `  spring-boot-starter-jpa`

application.properties	`spring.datasource.url=jdbc:postgresql://`
	`  localhost:5432/service1`
	`spring.datasource.username=service11`
	`spring.datasource.password=password`

Zusätzlich muss je nach Datenbanktyp der entsprechende JDBC-Treiber als Abhängigkeit definiert werden.

Spring Data MongoDB

- **Abhängigkeiten**

Abhängigkeiten	`org.springframework.boot:` `  spring-boot-starter-mongo`

application.properties	`spring.data.mongodb.host = localhost`
	`spring.data.mongodb.port = 27017`
	`spring.data.mongodb.database = database1`
	`spring.data.mongodb.username = user`
	`spring.data.mongodb.password=password`

Spring Redis Cache

Um die Funktionen von *Spring Redis Cache* nutzen zu können, müssen die nachfolgenden Schritte gemacht werden:

Abhängigkeiten	`org.springframework.session:` `  spring-session-data-redis`

application.properties	`spring.redis.host = localhost`
	`spring.rabbitmq.port = 45542`
	`spring.rabbitmq.password = password`

Annotationen	`@EnableRedisRepositories` `@EnableCaching`

RabbitMQ-Clients

Um eine Anwendung als RabbitMQ-Client einzurichten, bedarf es folgender Konfiguration:

Dependencies	`org.springframework.cloud:` `  spring-cloud-starter-stream-rabbit`

application.properties	`spring.rabbitmq.host = localhost`
	`spring.rabbitmq.port = 5672`
	`spring.rabbitmq.username = user`
	`spring.rabbitmq.password = password`

Einrichtung von Point-to-Point Kommunikation

[Piv18d]:

- **Allgemeine Konfiguration**
  Sowohl beim Publisher als auch beim Listener muss eine Queue mit dem gewünschtem Namen in Form einer Bean registriert werden:

```
1  @Configuration
2  public class MQConfiguration {
3          @Bean
4          Queue exampleQueue() {
5                  return new Queue("example-queue");
6          }
7  }
```

- **Publisher**
  Beim Versender (*Publisher*) wird zusätzlich eine Klasse zum Versenden von Nachrichten in Form einer Bean registriert. Der Message Publisher kann an jeder Stelle der Anwendung injiziert werden. Die einfachste Konfiguration eines Publishers erfordert keinerlei Konfigurationen und benötigt dem zufolge keine Konstruktor-Argumente. Für weitere Konfigurationsmöglichkeiten sei an dieser Stelle an die offizielle Dokumentation unter [Piv18c] verwiesen.

```
1  public class MessagePublisher {
2
3          @Autowired
4          RabbitTemplate rabbitTemplate;
5
6          @Autowired
7          Queue queue;
8
9          public void publishDeliveryEvent(String userId) {
10                 rabbitTemplate.convertAndSend(queue.getName(),
11                         "", "New delivery for Customer: " + userId);
12         }
13 }
```

```
@Bean
MessagePublisher messagePublisher() {
        return new MessagePublisher();
}
```

- **Listener**
  Um den Empfänger (*Listener*) einzurichten, bedarf es einer Klasse mit einer Methode, welche mit der Annotation `@RabbitListener(queues = #queue.name)` versehen wird. Bei `#queue.name` handelt es sich um den Namen der abonnierten Queue.

```
@Component
public class MessageListener {
        @RabbitListener(queues = "#{queue.name}")
        public int processDelivery(String user) {
                //do something
                return 0;
        }
}
```

Falls mehrere Listener auf die selbe Queue lauschen, werden die Nachrichten nach dem *Round-Robin-Prinzip* an jeweils einen Client zugestellt. Dieser Anwendungsfall tritt beispielsweise im falle einer horizontalen Skalierung auf.

Einrichtung von Publish-Subscribe

Um eine Publish-Subscribe-Kommunikation zu ermöglichen, müssen Exchange-Topics deklariert werden, welche mittels Binding mit einer Queue verbunden werden. Der Publisher sendet Nachrichten nicht an eine Queue, sondern an einen Topic und die Listener müssen jeweils ihre eigene Queue definieren, welche die Nachrichten von diesem Topic abholt [Piv18e]:

- **Allgemeine Konfiguration**
  Neben der Queue muss nun eine Bean vom Supertyp `Exchange` deklariert werden. Durch das `Binding` wird die Queue mit dem Exchange verbunden:

```
@Configuration
public class MQConfiguration {

        @Bean
        Queue queue() {
                return new Queue("example-queue");
        }
        @Bean
        public FanoutExchange fanout() {
                return new FanoutExchange("example-topic");
        }
        @Bean
        public Binding binding1(FanoutExchange fanout, Queue queue) {
                return BindingBuilder.bind(queue).to(fanout);
        }
}
```

- **Publisher**
  Der `Publisher` wird analog zum oberen Beispiel erstellt, lediglich mit dem Unterschied, dass dieser die Nachrichten nicht mehr direkt an eine Queue, sondern an einen Topic versendet:

```
1  public class MessagePublisher {
2
3          @Autowired
4          RabbitTemplate rabbitTemplate;
5
6          @Autowired
7          FanoutExchange fanout;
8
9          public void publishRabattCreated(Rabatt rabatt) {
10         rabbitTemplate.convertAndSend(fanout.getName(), "",
11                 "New Rabatt Created for " + rabatt.getUserId());
12         }
13 }
```

- **Listener**
  Die Implementierung der Listener-Klasse bleibt unverändert, da die Nachrichten nicht direkt aus dem Topic, sondern von der mit dem Topic verbundenen Queue abgeholt werden.

Entgegen der unter [Piv18e] verfügbaren offiziellen Dokumentation sollen zum Abholen der Topics von einem Exchange keine Queues mit einem zufälligen Namen - `AnonymousQueue()` verwendet werden, sondern Queues mit einem fest vergebenen Namen. Sonst wird bei einer skalierten Anwendung die selbe Message von jedem Node abgeholt und damit der Vorgang, welcher eigentlich durch den Empfang der Nachricht einmal ausgelöst werden soll, bei `n` Nodes `n` mal abgearbeitet.

Spring Cloud Zipkin

Spring Cloud Zipkin beinhaltet sowohl *Sleuth* als auch den Client für die Kommunikation mit dem Zipkin-Server:

Abhängigkeiten	`org.springframework.cloud:` `  spring-cloud-starter-zipkin`
application.properties	`spring.zipkin.sender.type = web` `spring.zipkin.baseUrl = http://localhost:9411`

Spring Boot Actuator

Um *Spring Boot Actuator* nutzen zu können, müssen folgende Konfigurationen vorgenommen werden:

Dependencies	`org.springframework.boot:` `spring-boot-starter-actuator`
application.properties	`management.endpoints.web.exposure.include=*`

Unter dem Wert `management.endpoints.web.exposure.include` werden die Endpoints angegeben, welche bereitgestellt werden sollen. `*` steht dabei für die Freigabe sämtlicher Schnittstellen.

Spring Boot Admin

Spring Boot Admin wird in Form einer Spring-Boot-Anwendung erstellt. Um ihn nutzen zu können, müssen folgende Konfigurationen vorgenommen werden:

Dependencies	`de.codecentric :` `spring-boot-admin-starter-server`
application.properties	`management.endpoints.web.exposure.include=*`

An den zu beobachtenden Anwendungen muss nichts angepasst werden, die benötigten Informationen (Adresse des Actuator-Endpoints `/health`)werden vom *Eureka*-Server bezogen.

OAuth2-Server

Abhängigkeiten	`org.springframework.boot:` `spring-boot-starter-security`
Annotationen	`@EnableAuthorizationServer`

Um die Authentifizierung zu konfigurieren muss eine von der Klasse `AuthorizationServerConfigurerAdapter` abgeleitete Konfigurationsklasse erstellt werden. Durch das Überschreiben der bereitgestellten Methode
`configure(AuthorizationServerSecurityConfigurer security)`
werden die Zugriffsrechte für den Token-Endpoint (`/oauth/token`) des Servers konfiguriert. Mit der Methode
`configure(ClientDetailsServiceConfigurer clients)`
können Ressourcen-Server für den `Client Credentials Grant` registriert werden. Der Methode
`configure(AuthorizationServerEndpointsConfigurer endpoints)`
wird der JWT-Tokenstore sowie eine Implementierung der Klasse `UserDetailService` in Form von `Beans` übergeben.

Die vorgestellte Konfiguration dient lediglich den Vorführzwecken und sollte in dieser Form nicht in einer produktiven Umgebung eingesetzt werden. Insbesondere sollten die Clients nicht im Arbeitsspeicher des Programms mit einem in Klartext geschriebenen Passwort hinterlegt werden sondern in einer Datenbank oder einem LDAP-Server unter Einbezugnahme entsprechender Sicherheitsvorkehrungen abgelegt werden.

```
1  public class SsoServerConfiguration
2  extends AuthorizationServerConfigurerAdapter{
3
4      @Autowired
5      private AuthenticationManager authenticationManager;
6
7      @Autowired
8          @Qualifier("userDetailsService")
9          private UserDetailsService userDetailsService;
10
11     @Override
12     public void configure(AuthorizationServerSecurityConfigurer security)
13         throws Exception {
14         security
15                 .tokenKeyAccess("permitAll()")
16                 .checkTokenAccess("isAuthenticated()")
17                 .allowFormAuthenticationForClients();
18     }
19
20     @Override
21     public void configure(ClientDetailsServiceConfigurer clients)
22         throws Exception {
23         clients.inMemory().withClient("acme")
24                 .authorizedGrantTypes("client_credentials",
25                                 "password","refresh_token")
26                 .authorities("CLIENT")
27                 .scopes("read", "write", "trust")
28                 .resourceIds("oauth2-resource")
29                 .accessTokenValiditySeconds(5000)
30                 .secret("{noop}secret").refreshTokenValiditySeconds(50000);
31     }
32
33     @Override
34     public void configure(AuthorizationServerEndpointsConfigurer endpoints)
35         throws Exception {
36         endpoints.tokenStore(tokenStore())
37                  .accessTokenConverter(accessTokenConverter())
38                  .userDetailsService(userDetailsService)
39                  .authenticationManager(authenticationManager);
40         }
41 }
```

Anhang 2: Einrichtung Java EE 8 mit Microprofile und Open Liberty

Um Funktionen in Open Liberty zu aktivieren, müssen diese in der Datei `server.xml` aktiviert werden sowie in Form von Abhängigkeiten mit dem Scope *provided* dem Projekt hinzugefügt werden. Nachfolgend wird die Einrichtung der wesentlichen für eine Cloud-Infrastruktur benötigten Komponente dokumentiert. Diese Anleitungen basieren mitsamt auf den unter [IBM18k] verfügbaren *Guides*.

Circuit Breaker mit Microprofile Fault Tolerance

In der Datei `server.xml` muss das nachfolgende Feature angegeben werden:

```
<feature>mpFaultTolerance-1.1</feature>
```

Das Einrichten des Circuit Breakers erfolgt im Anwendungscode über Annotationen:

```
@Fallback(fallbackMethod = "fallbackEmail")
@CircuitBreaker(requestVolumeThreshold = 20, failureRatio = 0.5, delay = 500)
public String getEmailAdressFromUser(String userId) {
        //some code
        return userEmail;
}
public String fallbackEmail(String userId) {
        return "no@where.com";
}
```

Relationale Datenbanken mit JPA

In der Datei `server.xml` muss das Feature für JPA aktiviert werden. Der Pfad für den entsprechenden Datenbanktreiber, im Beispiel der Treiber für eine eingebettete Derby-Instanz, muss angegeben werden, sowie die Datenbank mit dem Verweis auf den Treiber definiert werden:

```
 <feature>jpa-2.2</feature>

 <library id="derbyJDBCLib">
      <fileset dir="path/to/some/directory" includes="derby*.jar"/>
</library>
<dataSource
        id="jpadatasource"
        jndiName="jdbc/jpadatasource">
        <jdbcDriver libraryRef="derbyJDBCLib" />
</dataSource>
```

Um aus der Anwendung heraus auf die Datenbankinstanz zugreifen zu können, wird der Entity Manager benötigt:

```
@PersistenceContext
private EntityManager em;
```

Dieser kann in eine Service-Klasse eingeschlossen werden, welche diesen anschließend zum Ausführen von Datenbank-Operationen aufrufen kann:

```
List<Item> itemList =
        em.createNamedQuery("Item.findAll", Item.class).getResultList();
```

Die vom Persistence Context verwalteten Entities werden wie folgt erstellt:

```
@Entity
@NamedQuery(name = "Item.findAll", query = "SELECT i FROM Item i")
public class Item implements Serializable {

        private static final long serialVersionUID = -7603248252310880384L;

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        private String title;

        private String description;

        private Integer price;

        //Getters and Setters
```

Unter der Annotation `@NamedQuery` können Queries definiert werden, welche vom Entity Manager mithilfe des definierten Namens aufgerufen werden können.

Synchrone Kommunikation mit ReST

zum Aktivieren des Features sind demnach die nachfolgenden Schritte notwendig:

```
<feature>jaxrs-2.1</feature>
<feature>jsonp-1.1</feature>
<feature>mpRestClient-1.1</feature>
```

Das Anbieten und Konsumieren von ReST-Schnittstellen wurde bereits im Hauptteil der Ausarbeitung beschrieben:

```
@Path("/{userId}")
public class UserDetailsEndpoint {

   @Inject
   UserService userService;

  @GET
   @Produces(MediaType.APPLICATION_JSON)
   public Response getUserDetails(@PathParam("userId") String userId) {
         UserDetails userDetails = userService.getUserByUserName(userId);
         return (userDetails == null ? Response.status(404).build() :
                        Response.ok(userDetails).build());
   }
}
```

```
@Dependent
@RegisterRestClient
public interface RabattRestClient {
  @GET
   @Produces(MediaType.TEXT_PLAIN)
   public Integer getRabatt(@PathParam("userId") String userId);
}
```

JMS 2.0

Messaging Server erstellen

Um eine instanz von Open Liberty in Form eines Message Brokers zu verwenden, müssen neben dem benötigten Feature Ports definiert sowie die benötigten *Queues* und *Topics* erstellt werden:

```
<feature>wasJmsServer-1.0</feature>

<wasJmsEndpoint
         wasJmsPort="7276"
         wasJmsSSLPort="9127"
/>
<messagingEngine>
         <topicSpace id="USER_CREATED_TOPIC_SPACE" />
         <queue id="ORDER_CREATED" />
</messagingEngine>
```

Messaging Client erstellen

Um eine Open-Liberty-Instanz als Messaging Client zu konfigurieren, bedarf es der nachfolgenden Einstellungen in der Datei `server.xml`:

```
<feature>wasJmsClient-2.0</feature>

<jmsQueueConnectionFactory jndiName="jndi_JMS_BASE_QCF">
      <properties.wasJms
     remoteServerAddress="localhost:7276:BootStrapBasicMessaging" />
</jmsQueueConnectionFactory>

 <jmsActivationSpec id="rabatt-service/UserCreatedListener">
         <properties.wasJms destinationRef="jndi_TOPIC_USER_CREATED" />
</jmsActivationSpec>

<jmsTopic id= "jndi_TOPIC_USER_CREATED" jndiName="jndi_TOPIC_USER_CREATED">
          <properties.wasJms topicName="USER_CREATED_TOPIC_SPACE" />
</jmsTopic>

<jmsQueue id="jndi_ORDER_CREATED" jndiName="jndi_ORDER_CREATED">
     <properties.wasJms queueName="ORDER_CREATED" />
</jmsQueue>
```

Dabei wird das Feature für die Funktionalität des JMS-Clients definiert, Verbindungsdaten zum Messaging-Broker angegeben, Die *Activation Spec* für die MDB angegeben, sowie die benötigten, auf dem Messaging-Server verhandenen Queues und Topics registriert.

MDB erstellen

Eine MDB wird wie folgt erstellt:

```
@MessageDriven(
name = "OrderCreatedListener", activationConfig = {
@ActivationConfigProperty(
  propertyName = "connectionFactoryLookup",propertyValue="jndi_JMS_BASE_QCF"),
@ActivationConfigProperty(
  propertyName = "destination",propertyValue="jndi_ORDER_CREATED"),
@ActivationConfigProperty(
  propertyName = "destinationType",propertyValue="javax.jms.Queue"), })
public class NewOrderListener implements MessageListener {
  @Override
  public void onMessage(Message message) {
 // Do something
  }
}
```

In der Methode `onMessage` wird definiert, was beim Empfang einer Message ausgelöst werden soll.

Messages versenden

Um Messages versenden zu können, muss nachfolgendes definiert werden:

```
@ApplicationScoped
public class NewOrderPublisher {
        @Resource(lookup = "jndi_JMS_BASE_QCF")
        QueueConnectionFactory cf;

        @Resource(lookup = "jndi_ORDER_CREATED")
        Queue queue;

        public void sendMessage(String message) throws Exception {
                QueueConnection con = cf.createQueueConnection();
                con.start();
                QueueSession queueSession = con.createQueueSession(
                        false, javax.jms.Session.AUTO_ACKNOWLEDGE);
                QueueSender queueSender = queueSession.createSender(queue);
                TextMessage m = queueSession.createTextMessage(message);
                queueSender.send(m);
                con.close();
        }
```

Microprofile Metrics

Um bei einer Anwendung den `/metrics`-Endpoint einzuschalten, sind die folgenden Schritte ausreichend:

```
<feature>mpMetrics-1.1</feature>
 <feature>mpHealth-1.0</feature>
```

Security

Um eine Anwendung um die Funktion von Microprofile JWT zu erweitern, bedarf es der nachfolgenden Konfiguration:

```
<feature>appSecurity-3.0</feature>
<feature>mpJwt-1.1</feature>
<feature>mpConfig-1.3</feature>

<mpJwt
        userNameAttribute="user_name"
        groupNameAttribute="authorities"
        signatureAlgorithm="HS256"
        sharedKey="123"
        issuer="open-liberty"
        id="jwtUserConsumer"
        keyName="default"
                audiences="oauth2-resource"
        />
```