SWC Software Construction

RWTH AACHEN UNIVERSITY

FAKULTÄT FÜR MATHEMATIK,
INFORMATIK UND
NATURWISSENSCHAFTEN

LEHR- UND FORSCHUNGSGEBIET
INFORMATIK 3
SOFTWAREKONSTRUKTION

BACHELORARBEIT

# Evaluation von Microservices als Erweiterung etablierter Komponententechnologien am Beispiel von CiBo

## Evaluation of Microservices as an extension to established component technologies in the context of CiBo

vorgelegt von

**Niklas Jörg Scholz**

aus Velbert

am 2. Februar 2015

GUTACHTER

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

BETREUER

Dipl.-Inform. Andreas Steffens

---

Hiermit erkläre ich, Niklas Jörg Scholz, an Eides statt, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keinem anderem Prüfungsamt vorgelegt und auch nicht veröffentlicht.

Aachen, 2. Februar 2015 (Niklas Jörg Scholz)

# Danksagung

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. rer. nat. Horst Lichter bedanken, dass er mir die Möglichkeit gegeben hat an seinem Lehrstuhl, dem Lehr- & Forschungsgebiet 3 Softwarekonstruktion, diese Bachelorarbeit verfassen zu können. Außerdem danke ich ihm und Prof. Dr. rer. nat. Bernhard Rumpe für die Korrektur dieser Arbeit.
Einen ganz besonderen Dank richte ich an meinen Betreuer Dipl.-Inform. Andreas Steffens, der mich während meiner Arbeit großartig unterstützt hat und immer gute Ratschläge für mich parat hatte.
Meinen Eltern möchte ich dafür danken, dass sie mir während meines Studiums nicht nur finanziell sondern auch moralisch immer zur Seite standen. Meinem Vater danke ich außerdem dafür, dass er viel Zeit in Korrekturlesen investiert hat. Abschließend gilt mein Dank allen denen, die mich während der Anfertigung der Bachelorarbeit unterstützt und motiviert haben. Vielen Dank!

*Niklas Jörg Scholz*

# Kurzdarstellung


### Deutsch

Die vorliegende Arbeit beschäftigt sich mit dem neuen Architekturansatz Microservices und der Frage, ob diese eine sinnvolle Erweiterung von Komponententechnologien darstellen. Dazu werden zunächst etablierte Komponententechnologien (Java EE, OSGi, .NET und CORBA) betrachtet und anhand von Software-Qualitätsmerkmalen, die auf Wartbarkeit basieren, bewertet. Anschließend wird das Konzept von Microservices eingeführt und genauer betrachtet. Hierfür wird am Beispiel des CiBo Projekts, eine E-Ticketlösung für Busse, ein Backend auf Basis von Microservices entwickelt. Es wird ein Betreiber-Backend realisiert und die Komponenten Registrierung, Login, Fahrtenspeicherung, Fahrtensuche, Preisberechnung und Haltestellenverwaltung als Microservices implementiert. Auf Basis dieser Implementierung werden Microservices in Verbindung mit dem Spring Framework als Komponententechnologie evaluiert und bewertet.

### English

This thesis deals with the new architectural style microservices and the question whether they are a useful extension to component technologies. Therefore, established component technologies (Java EE, OSGi, .NET and CORBA) will be introduced and rated in terms of software qualities which are based on maintainability. Subsequently, the concept of microservices will be introduced and investigated. Therefore, a backend on the basis of microservices will be implemented in the context of CiBo, an e-ticket system for busses. An operator-backend will be developed and the components registration, login, save trip, trip search, price calculation and bus stop management will be implemented as microservices. On the basis of this implementation microservices combined with the Spring Framework will be evaluated and rated as a component technology.

# Inhaltsverzeichnis

**1 Einleitung** **1**

**2 Einführung** **3**
2.1 Projektbeschreibung CheckIn BeOut . . . 3
2.2 Prototyp und Problemstellung . . . 4

**3 Grundlagen** **7**
3.1 Komponentenbegriff . . . 7
3.1.1 Was ist eine Komponente? . . . 7
3.1.2 Was sind Komponententechnologien? . . . 8
3.2 Etablierte Komponententechnologien . . . 8
3.2.1 Java EE . . . 8
3.2.2 OSGi . . . 11
3.2.3 .NET . . . 15
3.2.4 CORBA . . . 18
3.3 Zusammenfassung . . . 22

**4 Bewertung etablierter Komponententechnologien** **23**
4.1 Software-Qualitätsmerkmale . . . 23
4.1.1 Änderbarkeit . . . 23
4.1.2 Testbarkeit . . . 24
4.1.3 Portabilität . . . 25
4.2 Bewertung . . . 25
4.2.1 Java EE . . . 25
4.2.2 .NET . . . 26
4.2.3 OSGi . . . 27
4.2.4 CORBA . . . 27
4.3 Fazit . . . 28

**5 Microservices** **29**
5.1 Eigenschaften . . . 29
5.2 Vorteile . . . 31
5.3 Nachteile . . . 31
5.4 Zusammenfassung . . . 32

**6 Anforderungen** **33**
6.1 Zentrale Begriffe . . . 33

6.2 Stakeholder . . . 34
6.3 Funktionale Anforderungen . . . 34
6.4 Nicht funktionale Anforderungen . . . 37
6.5 Zusammenfassung . . . 37

**7 Entwurf** **39**
7.1 Schichten . . . 39
7.1.1 Kommunikation mit REST . . . 41
7.2 Komponenten . . . 41
7.2.1 Microservice Architektur . . . 41
7.2.2 Registrierung . . . 42
7.2.3 Login . . . 43
7.2.4 Fahrtenspeicherung . . . 44
7.2.5 Fahrtensuche . . . 44
7.2.6 Preisberechnung . . . 45
7.2.7 Haltestellenverwaltung . . . 46
7.2.8 Web-Interface . . . 46
7.3 Entitäten . . . 48
7.3.1 Fahrten Datenmodell . . . 48
7.3.2 Login Datenmodell . . . 48
7.4 Dynamische Sicht . . . 51
7.4.1 Registrierung . . . 51
7.4.2 Login . . . 52
7.4.3 Passwort ändern . . . 52
7.4.4 Fahrtenspeicherung . . . 52
7.4.5 Darstellung der Fahrten inklusive Preis . . . 53
7.4.6 Erstellung neuer Haltestellen . . . 54
7.5 Zusammenfassung . . . 54

**8 Implementierung** **57**
8.1 Verwendete Technologien . . . 57
8.1.1 Spring & Spring Boot . . . 57
8.1.2 AngularJS . . . 58
8.1.3 Swagger . . . 58
8.1.4 JHipster . . . 58
8.2 Microservices . . . 59
8.2.1 Umsetzung Entitäten . . . 59
8.2.2 Registrierung . . . 60
8.2.3 Login . . . 60
8.2.4 Fahrten speichern . . . 60
8.2.5 Fahrtensuche . . . 62
8.2.6 Ticketpreis . . . 63
8.2.7 Haltestellenverwaltung . . . 64
8.3 Web-Frontend . . . 64

8.4 Gesamtübersicht . . . . . . . . . . 64
8.5 Testen . . . . . . . . . . 65
8.6 Zusammenfassung . . . . . . . . . . 67

**9 Evaluation** **69**
9.1 Microserivces und der Komponentenbegriff . . . . . . . . . . 69
9.1.1 Komponenten . . . . . . . . . . 69
9.1.2 Komponententechnologie . . . . . . . . . . 70
9.2 Beurteilung von Microservices . . . . . . . . . . 70
9.2.1 Bewertung anhand von Software-Qualitätsmerkmalen . . . . . . . . . . 71
9.2.2 Sinnvolle Erweiterung etablierter Komponententechnologien? . . . 71
9.3 Erkannte Herausforderungen und Fragestellungen . . . . . . . . . . 72
9.4 Fazit . . . . . . . . . . 73

**10 Zusammenfassung und Ausblick** **75**
10.1 Zusammenfassung . . . . . . . . . . 75
10.2 Ausblick . . . . . . . . . . 76

**A Anhang** **79**
A.1 Web-Interface Screenshots . . . . . . . . . . 79

**Literaturverzeichnis** **83**

**Abkürzungsverzeichnis** **87**

# Tabellenverzeichnis


7.1 Schnittstellen des Registrierungs-Microservice . . . . . . . . . . . . . . . . . 43
7.2 Schnittstellen des Login-Microservice . . . . . . . . . . . . . . . . . . . . . . . 44
7.3 Schnittstellen des Fahrtenspeicherung-Microservice . . . . . . . . . . . . . . . 44
7.4 Schnittstellen des Fahrtensuche-Microservice . . . . . . . . . . . . . . . . . . 45
7.5 Schnittstellen des Preisberechnung-Microservice . . . . . . . . . . . . . . . . 46
7.6 Schnittstellen des Haltestellen-Microservice . . . . . . . . . . . . . . . . . . . 47
7.7 Übersicht über die Objekte des Datenmodells . . . . . . . . . . . . . . . . . 50

8.1 Übersicht über das Tarifmodell . . . . . . . . . . . . . . . . . . . . . . . . . 63

# Abbildungsverzeichnis


2.1 CheckIn BeOut Comic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Übersicht über den Prototypen des CiBo Projekts . . . . . . . . . . . . . 4

3.1 Übersicht über die Java EE Container [34] . . . . . . . . . . . . . . . . . . . 9
3.2 Übersicht über die OSGi Architektur (Anlehnung an [52]) . . . . . . . . . 12
3.3 Grobarchitektur des .Net-Frameworks (Anlehnung an [2]) . . . . . . . . . 15
3.4 Statischer & Dynamischer Aufruf eines CORBA-Objekts (Anlehnung an [12] und [32]) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4.1 Qualitätenbaum [22] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

5.1 Vergleich von Monolith und Microservices [6] . . . . . . . . . . . . . . . . 30

7.1 Schichtendarstellung des Betreiber-Backends . . . . . . . . . . . . . . . . . 40
7.2 Architekturoptionen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
7.3 Komponentendiagramm des Registrierungs Microservice . . . . . . . . . . 42
7.4 Komponentendiagramm der Web-Interface Komponente . . . . . . . . . . 47
7.5 Übersich über das Fahrten Entity-Relationship (ER)-Modell . . . . . . . . 49
7.6 Übersich über das Login ER-Modell . . . . . . . . . . . . . . . . . . . . . 49
7.7 Sequenzdiagramm Registrierung eines Nutzers . . . . . . . . . . . . . . . . 51
7.8 Sequenzdiagramm einloggen in das Web-Interface . . . . . . . . . . . . . . 52
7.9 Sequenzdiagramm Änderung des Passworts . . . . . . . . . . . . . . . . . 53
7.10 Sequenzdiagramm Speicherung von Fahrten . . . . . . . . . . . . . . . . . 53
7.11 Sequenzdiagramm Darstellung der Fahrten im Web-Interface . . . . . . . 54

8.1 Screenshot eines Swagger Tests . . . . . . . . . . . . . . . . . . . . . . . . 59
8.2 Klassendiagramm Registrierung . . . . . . . . . . . . . . . . . . . . . . . . 61
8.3 Übersicht über das Betreiber Backend . . . . . . . . . . . . . . . . . . . . 65

A.1 Screenshot der Startseite . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
A.2 Screenshot des Logins . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
A.3 Screenshot der Passwortänderung . . . . . . . . . . . . . . . . . . . . . . . 80
A.4 Screenshot der Fahrtensuche . . . . . . . . . . . . . . . . . . . . . . . . . . 81
A.5 Screenshot der Haltestellenübersicht . . . . . . . . . . . . . . . . . . . . . 81
A.6 Screenshot neue Haltestelle hinzufügen . . . . . . . . . . . . . . . . . . . . 82

# Liste der Quelltexte


8.1 Methode des Representational State Transfer (REST)-Controllers TripResource.java zum Speichern von Fahrten . . . . . . . . . . . . . . . . . 61
8.2 JavaScript Object Notation (JSON) Bespiel eines POST . . . . . . . . . . . 62
8.3 JSON Beispiel eines POST inklusive Anfrage-URL . . . . . . . . . . . . . . 62
8.4 JSON Beispiel eines Response Body inklusive Anfrage-URL . . . . . . . . . 64
8.5 JavaScript Service, der auf den Haltestellen-Microservice zugreift . . . . . 64
8.6 JUnit Test der Fahrtenspeicherung . . . . . . . . . . . . . . . . . . . . . . . . 66

# 1 Einleitung

> The cheapest, fastest, and most reliable components are those that aren't there.
>
> GORDON BELL

Software hat sich immer mehr in das alltägliche Leben integriert, wodurch die Software immer relevanter wird und ein hoher Anspruch an Qualität und Stabilität herrscht. Gleichzeitig werden Softwareprojekte immer größer und komplexer, daher ist es schwierig die gesamte Codebasis zu beherrschen. Komponententechnologien haben sich über die Jahre in der Softwareentwicklung etabliert, da sie die Komplexität von Projekten verringern wollen und zum Ziel haben die Wartbarkeit von Code zu erhöhen. Zusätzlich stellen Komponententechnologien eine Infrastruktur bereit für Mechanismen wie Sicherheit oder Versionierung, wodurch sich die Entwickler besser auf die Implementierung des Codes konzentrieren können. Allerdings sind diese Technologien selbst auch umfangreich, daher ist der Aufwand hoch sich in die Komponententechnologien einzuarbeiten, wenn man noch nie zuvor damit entwickelt hat. Es stellt sich die Frage, ob diese wirklich die Wartbarkeit mit ihrem Konzept erhöhen oder ob diese noch verbessert werden kann.

Im Laufe des letzten Jahres ist der Begriff Microservices aufgetaucht. Microservices sollen Software durch übersichtliche Architektur, bessere Wartbarkeit und Wiederverwertbarkeit einzelner Services verbessern. Dieser Ansatz klingt sehr vielversprechend, wie jedoch realisiert man solche Microservices? Da es sich um einen neuen Begriff handelt, gibt es noch keine Literatur zu diesem Thema. Es gibt allerdings diverse Blogeinträge, die sich mit Microservices beschäftigen. In den meisten dieser Einträge werden Microservices nur grob betrachtet und Vorteile sowie eventuell auch deren Nachteile erläutert. Bisher existiert keine eindeutige Definition von Microservices. Daher ist das Ziel dieser Bachelorarbeit Microservices anhand eines konkreten Softwarebeispiels zu entwickeln und herauszuarbeiten, welche Vorzüge Microservices in der praktischen Anwendung haben. Besonders in Bezug auf die Wartbarkeit von Software sollen Microservices auch im Vergleich zu herkömmlichen Komponententechnologien wie Java EE, OSGi, .NET oder CORBA betrachtet werden. Die zentrale Frage dieser Arbeit ist, ob Microservices als Erweiterung etablierter Komponententechnologien geeignet sind. Konkretes Beispiel, an dem die Entwicklung von Microservices überprüft werden soll, ist das Betreiber-Backend des CheckIn BeOut Projekts (vgl. Abschnitt 2.1). Zur Zeit existiert hier ein mit Java EE entwickelter, monolithischer Prototyp, der im Rahmen dieser Arbeit ersetzt werden soll.

In Kapitel 2 erfolgt eine Einführung in das CiBo Projekt und die Beschreibung

der Problemstellung des aktuellen Prototyps. Daraufhin wird erläutert, was sich hinter dem Begriff Komponententechnologie verbirgt und die bekanntesten Komponententechnologien (Java EE, OSGi, .NET und CORBA) werden vorgestellt (Kapitel 3). In Kapitel 4 erfolgt die Bewertung der zuvor vorgestellten Komponententechnologien anhand von Qualitätsmerkmalen. Das fünfte Kapitel definiert die Anforderungen an die Software des Betreiber-Backends und das sechste Kapitel betrachtet den Begriff Microservices genauer. In Kapitel 7 folgt der Softwareentwurf. Anschließend wird in Kapitel 8 die Implementierung der Software genauer beschrieben. Nach der Entwicklung der Software können Microservices evaluiert und die zentrale Frage dieser Arbeit geklärt werden, ob Microservices sich als Erweiterung etablierter Komponententechnologien eignen (Kapitel 9). In Kapitel 10 werden die Ergebnisse dieser Arbeit zusammengefasst und es erfolgt ein Ausblick.

# 2 Einführung

Inhalt

2.1 Projektbeschreibung CheckIn BeOut . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Prototyp und Problemstellung . . . . . . . . . . . . . . . . . . . . . . . . 4

In diesem Kapitel erfolgt eine Einführung in das Projekt, da die im Rahmen dieser Arbeit zu entwickelnde Software Teil eines Projekts ist. Es wird erläutert, wie der aktuelle Prototyp aussieht und warum ein Teil dieses Prototyps neu Implementiert werden soll.

## 2.1 Projektbeschreibung CheckIn BeOut

Das Projekt CheckIn BeOut (CiBo) ist eine E-Ticketlösung für Busse. Über eine Applikation (App) auf dem Smartphone wählt der Benutzer den gewünschten Bus aus (Check in), daraufhin erhält er ein E-Ticket, um dies im Falle einer Kontrolle vorzeigen zu können. Anschließend werden vom Smartphone kontinuierlich Global Positioning System (GPS) Daten an ein Backend System gesendet, um die aktuelle Position des Benutzers zu überprüfen. Beim Verlassen des Busses erkennt das Backend auf Basis der GPS Daten des Benutzers und des Busses, dass der Benutzer ausgestiegen ist. Dieser muss selbst nichts tun, um die Fahrt abzuschließen (BeOut).

Abbildung 2.1: CheckIn BeOut Comic[1]

Das Projekt ist eine Kooperation zwischen dem Lehr- und Forschungsgebiet Informatik 3 an der RWTH Aachen und der IVU Traffic Technologies AG in Aachen. Die IVU

[1] © 2014 Felix Reidl – http://tcs.rwth-aachen.de/~reidl/

ist ein Software-Dienstleister, der verkehrstechnologische Lösungen anbietet. Außerdem erfolgt in diesem Projekt eine Zusammenarbeit mit der Aachener Straßenbahn und Energieversorgungs-AG (ASEAG), welche die E-Ticketlösung einsetzten will.

## 2.2 Prototyp und Problemstellung

Im Sommersemester 2014 wurden im Rahmen des Software Projektpraktikums bereits drei Prototypen, von drei verschiedenen Entwicklerteams, für die IVU entwickelt. Die Software ist in drei Teilsysteme unterteilt (vgl. Abbildung 2.2). Zum einen in eine Smartphone App, die auf dem Betriebssystem Android [10] basiert. Über diese erfolgt die Wahl der Buslinie und sie stellt dem Nutzer ein E-Ticket bereit. Während einer Fahrt ist die App für die Ortung des Nutzers zuständig und sendet diese Daten an das Tracking & BeOut System. Das zweite Teilsystem ist das Tracking und BeOut System, welches als Backend der App fungiert. Dieses enthält den BeOut Algorithmus, der erkennt, wann der Benutzer aus dem Bus ausgestiegen ist. Außerdem bietet das Tracking und BeOut System eine Schnittstelle zu dem Unified Realtime API (URA)-Service, der Echtzeitdaten der Busse übermittelt. Als drittes gibt es noch das Betreiber-Backend. Es stellt ein Informationssystem dar, welches für die Speicherung von Fahrten zuständig ist und ein Web-Interface bereitstellt. Dieses Web-Interface wurde mit AngularJS realisiert und ermöglicht es dem Betreiber sich dort einzuloggen und alle Fahrdaten der Benutzer einzusehen. Die Fahrten können nach Kriterien wie Buslinie oder Uhrzeit gefiltert werden.

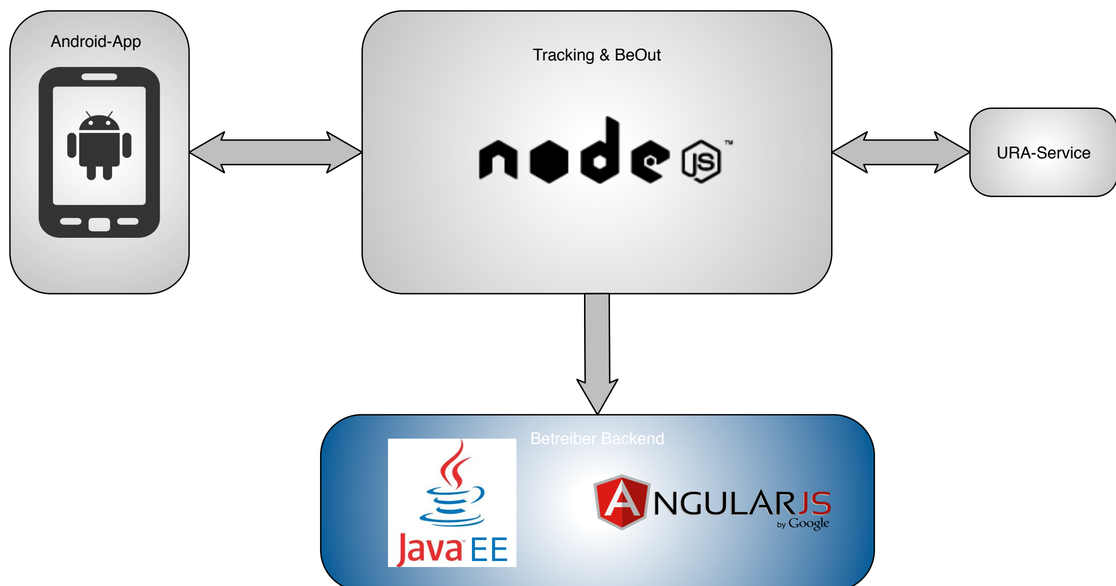


Abbildung 2.2: Übersicht über den Prototypen des CiBo Projekts

Im Fokus dieser Arbeit steht das zuvor genannte Betreiber-Backend, welches

in der aktuellen Lösung mit Java EE realisiert wurde. Das Problem bei diesem Prototypen ist, dass er eine monolithische Struktur aufweist, denn alle Komponenten sind eng miteinander verbunden. Solch ein Aufbau hat negative Auswirkungen auf die Wartbarkeit der Software. Beispielsweise ist die Änderbarkeit beeinträchtigt, da Anpassungen des Codes schnell zu Seiteneffekten führen können, wodurch unerwünschtes Verhalten entsteht. Es soll nun dieses Teilsystem des CiBo Projekts überarbeitet werden, da sich neue Anforderungen ergeben haben. Beispielsweise soll eine Preisberechnung integriert und eine Zahlung über einen Payment-Anbieter ermöglicht werden. Diese Situation wird genutzt, um das Betreiber-Backend mit einem neuen Architekturansatz zu entwickeln, von dem eine bessere Wartbarkeit erwartet wird. Dieser Architekturansatz basiert auf Microservices und knüpft an Komponententechnologien an.
Das nächste Kapitel führt die Begriffe Komponente und Komponententechnologie ein, die als Basis für die Anwendung des Microservices-Ansatzes im neuen Softwareentwurf des Betreiber-Backends dienen.

# 3 Grundlagen

Inhalt

3.1 Komponentenbegriff . . . . . 7
3.2 Etablierte Komponententechnologien . . . . . 8
3.3 Zusammenfassung . . . . . 22

In diesem Kapitel wird erläutert, welche Eigenschaften eine Komponententechnologie hat. Im Anschluss daran erfolgt eine Beschreibung von Komponententechnologien, die sich über die Jahre in der Softwarentwicklung etabliert haben. Dabei werden die Technologien Java EE, OSGi, .NET und CORBA vorgestellt.

## 3.1 Komponentenbegriff

Zunächst muss geklärt werden was sich hinter dem Begriff Komponententechnologie verbirgt. Dabei stellt sich die Frage: Was ist überhaupt eine Komponente?

### 3.1.1 Was ist eine Komponente?

Eine Komponente ist neben der Klasse und dem Objekt ein weiterer Begriff in der Software-Architektur und muss klar von den anderen beiden Begriffen unterschieden werden. Heineman und Councill definieren in ihrem Buch *Component-Based Software Engeneering* [14], eine Komponente als ein Softwareelement, welches unabhängig von anderer Software verwendet und somit ohne Änderung mit anderen Komponenten zusammengesetzt werden kann. Im Grunde lässt sich eine Komponente auf folgende drei Eigenschaften reduzieren:

**Unabhängigkeit** Eine Komponente ist eine Einheit, die unabhängig von anderen Komponenten installiert werden kann. Ludewig und Lichter beschreiben in ihrem Buch *Software Engeneering* [22], dass eine Abgegrenztheit nach außen besteht, da interne Operationen und Daten unzugänglich sind.

**Kompositionseigenschaft** Die zweite Eigenschaft einer Komponente ist, dass sie Teil einer Komposition sein kann. Neubauer, Ritter und Stoinski [32] definieren eine Komposition als einen Zusammenschluss von Komponenten über deren Interfaces, welche wieder als eine Komponente betrachtet werden kann. Sie bietet ihrer Umgebung Dienste an, stellt diese über Interfaces bereit und definiert zudem, welche Dienste anderer Komponenten in Anspruch genommen werden [22].

**Zustandslosigkeit** Als dritte Eigenschaft besitzt eine Komponente keinen dauerhaften Zustand, wodurch es sinnvoll ist, eine Komponente nur einmal in einem System zu haben (ähnlich wie bei einer Klasse). [32]

### 3.1.2 Was sind Komponententechnologien?

Gruhn und Tiel definieren, in ihrem Buch über Komponentenmodelle [12], dass eine Komponente für eine Verwendung innerhalb eines konkreten Komponentenmodells beziehungsweise einer Komponententechnologie ausgelegt ist. Dabei legt die Technologie den Rahmen für die Entwicklung und die Ausführung von Komponenten fest. Des Weiteren stellt sie Anforderungen hinsichtlich Komposition und Kollaboration. Außerdem stellt eine Komponententechnologie eine Infrastruktur bereit um Mechanismen wie Verteilung, Nachrichtenaustausch, Sicherheit oder Versionierung nutzen zu können.

## 3.2 Etablierte Komponententechnologien

Auf Basis der definierten Begrifflichkeit werden in diesem Abschnitt die bekanntesten Komponententechnologien und ihre wichtigsten Eigenschaften vorgestellt. Dazu wird jeweils die Architektur und die dafür nötige Infrastruktur beschrieben. Anschließend werden Vor- und Nachteile der jeweiligen Technologie betrachtet und zum Schluss werden noch Beispiele aus der Anwendung der Technologie genannt. Zwar erfolgt in diesem Kapitel eine Betrachtung der Vor- und Nachteile der Technologien, jedoch wird die eigentliche Bewertung in Kapitel 4 durchgeführt.

### 3.2.1 Java EE

Java Platform, Enterprise Edition (Java EE) ist eine von Oracle entwickelte Komponententechnologie, die auf der Programmiersprache Java basiert. Java EE ist eine Ansammlung von Application Programming Interfaces (APIs), die benutzt werden um komponentenbasierte mehrschichtige Anwendungen zu entwickeln [9].

#### Architektur

Die Basis der Java EE Architektur bildet der Java EE Application Server. Dieser stellt unter anderem Funktionalitäten wie Sicherheit, Transaktionsmanagement, Kommunikation zwischen Komponenten und Deployment zu Verfügung. Dabei ist der Server in verschiedene Container unterteilt (vgl. Abbildung 3.1) um die Komplexität einer Anwendung zu verringern. Durch verschiedene Protokolle, wie beispielsweise Hypertext Transfer Protocol (HTTP) Secure Sockets Layer (SSL), können die Container untereinander kommunizieren. Außerdem wird somit eine Infrastruktur bereitgestellt, die Funktionalitäten wie Sicherheit und Datenbankzugriff ermöglicht. Die folgenden viert Container stehen bei Java EE zur Verfügung:

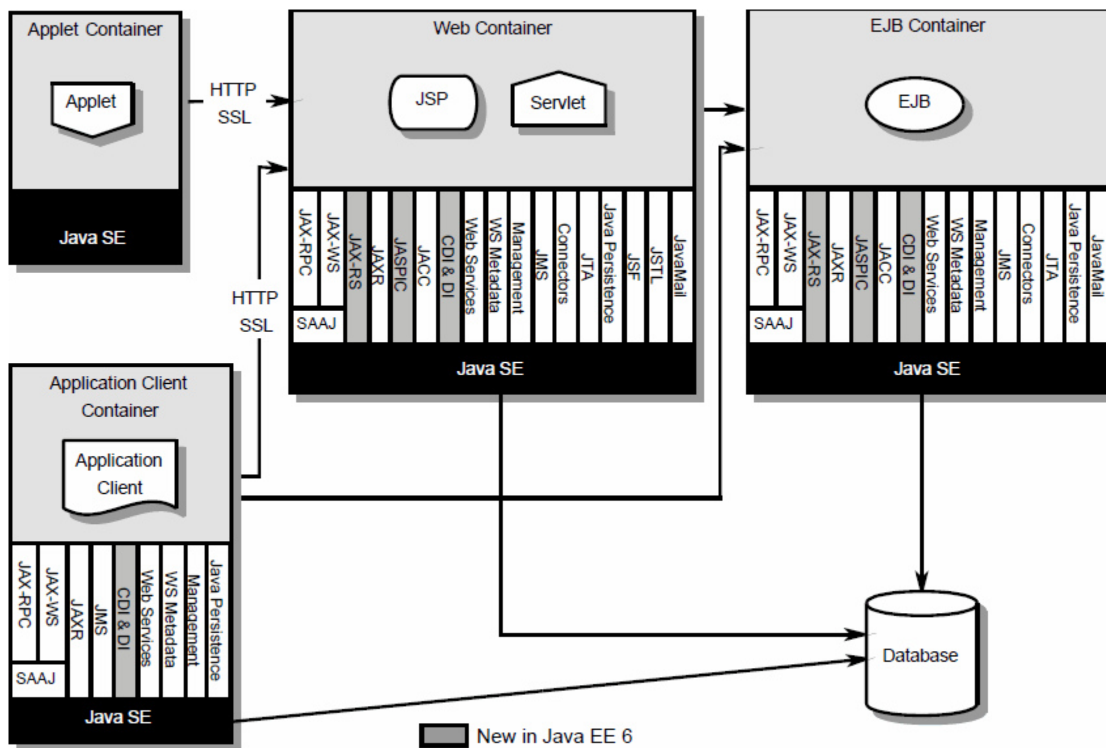


Abbildung 3.1: Übersicht über die Java EE Container [34]

**Applet Container** Der Applet Container wird von den gängigen Web-Browsern unterstützt und führt Applet-Komponenten aus. Applets, das sind Graphical User Interface (GUI) Anwendungen die im Webbrowser ausgeführt werden, also auf der Client Seite [43]. Bei der Entwicklung von Applets kann man sich auf die visuellen Aspekte konzentrieren, während sich der Container um weitere Funktionen kümmert.

**Application Client Container** Der Application Client Container enthält Java Klassen, Bibliotheken und sonstige Dateien um Funktionen wie Sicherheit zu gewährleisten und Abhängigkeiten aufzulösen. Der Application Client Container beinhaltet Komponenten die als Applications bezeichnet werden. Dies sind Programme die auf den Endgeräten ausgeführt werden, so wie GUIs oder Programme zur Datenverarbeitung [9].

**Enterprise JavaBeans (EJB) Container** Der EJB Container, verwaltet und führt EJBs aus. EJBs, das sind serverseitige Komponenten, die Geschäftslogik zusammenfassen und Datenzugriffe durchführen. Außerdem verwaltet der EJB Container den Lebenszyklus der Komponenten und stellt Funktionen wie Transaktions-, Namens- und Sicherheitsdienste zu Verfügung. [9]

**Web Container** Der Web Container hat die Verwaltung und Ausführung von Web-Komponenten als Aufgabe. Dieser Container wird verwendet, um die Webseiten mit Inhalten auszustatten. Der Web Container enthält verschiedene Arten von Komponenten. Zum einen gibt es Servlets, Programme die auf Servern ausgeführt werden, also Komponenten auf Server Seite [43]. Außerdem gibt es die JavaServer Faces (JSF), welche bei der Webentwicklung das Einbinden von Komponenten ermöglichen. Solche Komponenten können verwendet werden um im Webumfeld Benutzerschnittstellen wie beispielsweise Text Boxen zu benutzen. [9]

#### Notwendige Infrastruktur

Um ein Projekt mit Hilfe von Java EE zu realisieren, benötigt man eine spezielle Infrastruktur. Das Programmieren mit Java erfordert ein Java Development Kit. Des Weiteren benötigt man auch ein Java EE Software Development Kit (SDK) und einen Java EE Anwendungsserver. Der Anwendungsserver ist in die zuvor genannten Container unterteilt und stellt damit die Java EE Architektur bereit[34]. Beispielsweise von Sun Microsystems gibt es den open source Anwendungsserver GlassFish [33]. Zur Unterstützung der Entwicklung können beispielsweise Eclipse oder NetBeans verwendet werden, für die es beide extra Java EE Plugins gibt.

#### Vorteile

Java EE beinhaltet einige Vorteile, welche die Technologie zu einer der meist verwendeten in der Web-Entwicklung gemacht haben [45]. Beispielsweise das Konzept, den Anwendungsserver in Container aufzuteilen hat positive Auswirkungen. Hier durch wird die Softwarestruktur unterteilt und die Softwareentwicklung standardisiert, da bereits die Architektur vorgegeben ist. Außerdem ermöglicht die Verwendung von EJBs, dass sich auf die Implementierung der Geschäftslogik konzentriert werden kann. Man muss sich nicht um technische Aspekte wie Persistenz kümmern, da die Container eine Infrastruktur bereitstellen. Des Weiteren werden eine große Anzahl an APIs zu Verfügung gestellt, dessen Funktionen somit nicht neu entwickelt werden müssen. So gibt es in Java EE beispielsweise die JavaMail API zum automatischen Mails versenden, Security Services für Authentifizierung und Zugriffskontrolle und Web Services beispielsweise JAX-RS für die Unterstützung von RESTful Web Services. Java EE ist eine weit verbreitete Technologie, somit ist die *Community* sehr groß, wodurch sich die Technologie zu einer gut dokumentierten Plattform entwickelt hat, mit vielen erfahrenen Entwicklern. [9]

#### Nachteile

Natürlich weißt Java EE auch ein paar Nachteile auf, die bei der Wahl einer Komponententechnologie nicht vernachlässigt werden sollten. Wenn man ein Projekt mit Java EE realisieren will, dann benötigt man immer einen Anwendungsserver. Zwar

ist mit Java EE 7 dieser Anwendungsserver weniger komplex, als in den früheren Java EE Versionen, jedoch bringt er immer noch eine gewisse Größe mit sich. Java EE bietet viele Möglichkeiten, durch diverse Services und APIs. Allerdings hat dies nicht nur Vorteile, denn dadurch wird die Technologie natürlich auch komplexer. Das kann gerade bei kleineren Projekten unnötig unübersichtlich werden, da die meisten Funktionen nicht benötigt werden. So entstehen selbst bei kleineren Softwareprojekten Monolithen, also Software bei der sich Komponenten nur schlecht trennen lassen.

#### Anwendung

Viele große Softwareentwicklungs-Firmen entwickelt mit Java EE oder haben dies in der Vergangenheit getan. Besonders im Bereich Web-Entwicklung wird Java EE häufig verwendet. Es gibt auch viele Firmen die Java EE unterstützen wie Beispielsweise IBM oder Sun, die also überwiegend selbst Java EE nutzen. Auch SAP verwendet diese Komponententechnologie. Beispielsweise hat SAP einen eigenen Anwendungsserver, den SAP NetWeaver Application Server[1].

### 3.2.2 OSGi

OSGi ist eine Komponententechnologie, die auf der Programmiersprache Java basiert. Um Softwareprojekte mit Hilfe von OSGi zu entwickeln, benötigt man die OSGi Service Platform. Zunächst wurden mit OSGi sogenannte *residatial internet gateways* entwickelt, die der Vernetzung von technischen Geräten in einer Wohnumgebung und dem Internet dienen. Durch die Zunahme an Bekanntheit und die dadurch resultierende Standardisierung wurde die Softwareplattform auch für andere Bereiche interessant. Somit findet sie auch Verwendung in Smartphones, in der Automobilbranche, für Desktop-Applikationen und für Serverseitige Anwendungen.

#### Architektur

Die OSGi Softwareplattform ermöglicht es, die Software zu modularisieren, indem sie in Softwarekomponenten, sogenannte *Bundles* und in Dienste, auch als *Services* bezeichnet, unterteilt. Die Architektur von OSGi beinhaltet verschiedene Bereiche (vgl. Abbildung 3.2), die im Folgenden vorgestellt werden.

**Bundles** Wütherich, Hartmann, Kolb und Lübken definieren Bundles in ihrem Buch *Die OSGi Service Platform* [52] als eine fachlich oder technisch zusammenhängende Einheit aus Klassen und Ressourcen, die eigenständig installiert und deinstalliert werden kann. Dort werden die Abhängigkeiten zu anderen Bundles oder Services definiert und in einer Manifest-Datei festgehalten. Von diesen Abhängigkeiten können nur die Schnittstellen betrachtet werden und nicht die Implementationen der Klassen oder weitere Abhängigkeiten. Um einen Einblick zu erhalten müssen Packages explizit exportiert und vom benutzenden Bundle importiert werden.

[1] `http://de.wikipedia.org/wiki/SAP_NetWeaver_Application_Server`

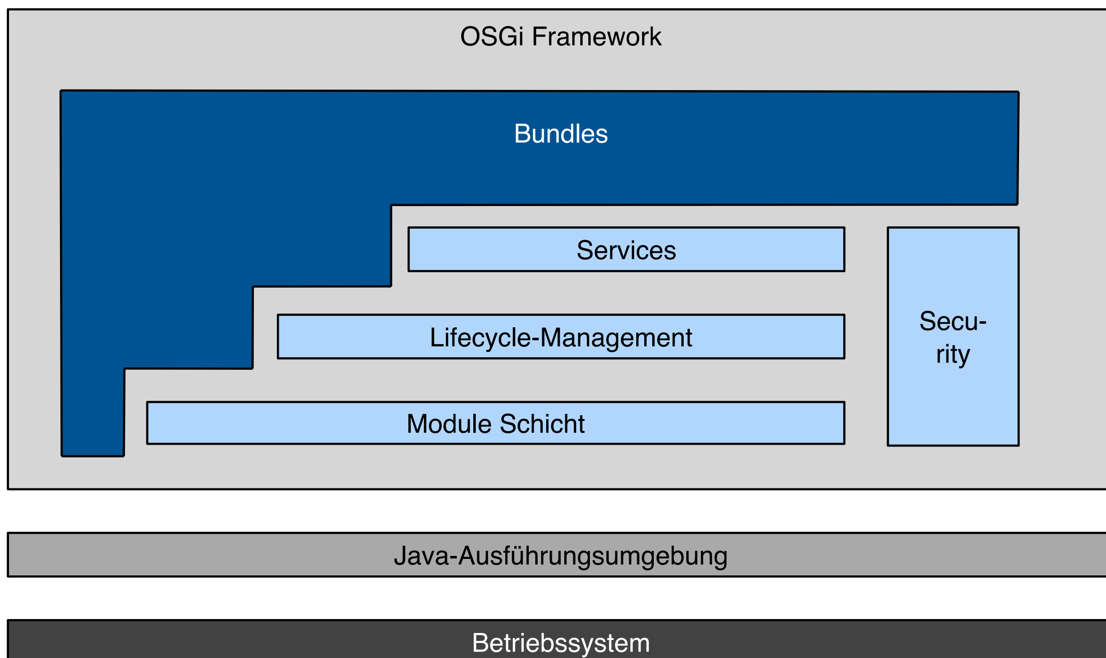


Abbildung 3.2: Übersicht über die OSGi Architektur (Anlehnung an [52])

Bundles werden als .jar-Dateien verwaltet, die zusätzlich noch Informationen über andere Bundles enthalten, von denen sie abhängig sind. Außerdem ist jedes Bundle in OSGi mit einem Bundle Objekt assoziiert und muss eindeutig identifizierbar sein [35].

**Services** Services werden durch ein Java Interface definiert und über eine sogenannte *Service Registry* verwaltet. Durch die Registrierung in der *Service Registry* stehen die Services systemweit zur Verfügung. Es existieren auch vordefinierte Services, die als *Standard Services* bezeichnet werden, wie beispielsweise der Log Service, der das protokollieren und auslesen von Systeminformationen ermöglicht [52]. Diese Standard Services können in eigenen Anwendungen genutzt werden ohne sie vorher selbst implementieren zu müssen. Sie dienen außerdem der Vereinheitlichung und Standardisierung von OSGi Applikationen.

**Lifecycle-Management** In der Lifecycle-Management Schicht wird festgelegt, welchen Zustand ein Bundle während seines Lebenszyklus haben kann und durch welche Aktionen dieser verändert wird. Ein Bundle kann folgende Zustände annehmen:

- INSTALLED - Bundle erfolgreich installiert.
- RESOLVED - Alle Abhängigkeiten des Bundles stehen zur Verfügung.
- STARTING - Bundle wird gestartet.
- RUNNING - Bundle läuft.

- STOPPING - Bundle wird gestoppt.
- DEINSTALLED - Bundle wird deinstalliert.

Der sogenannte Management Agent kann diese Zustände, die das Bundle einnehmen kann, von außen manipulieren, da er extern auf die OSGi Softwareplatform zugreifen kann. [1]

**Module Schicht** In der Module Schicht werden die Abhängigkeiten der Bundles aufgelöst, die in der zuvor erwähnten Manifest-Datei definiert wurden. Dadurch kann das Bundle nun als gesamte Einheit, inklusive der Abhängigkeiten verwendet werden.

**Security** Diese Schicht ist für die Sicherheit zuständig. Hier durch werden die Handlungsspielräume der einzelnen Bundles eingeschränkt und können individuell konfiguriert werden.

Die vorgestellten Schichten stellen das OSGi Architektur dar. Diese läuft auf Basis einer Java-Ausführungsumgebung, welche wiederum auf dem Betriebssystem aufgesetzt ist.

#### Notwenige Infrastruktur

Um die OSGi Serviceplattform zu verwenden, benötigt man zunächst ein Java Development Kit (JDK). Des Weiteren bedarf es eines OSGi-Frameworks, allerdings ist die Referenzimplementierung der OSGi Alliance lediglich eine Vorlage für andere Implementierungen. Man kann sich für ein kommerzielles oder quelloffenes Framework entscheiden, beispielsweise gibt es von Eclipse das Framework Equinox. Ansonsten wird noch ein Application-Server benötigt. Auch hier gibt es für die OSGi-Technologie verschiedene Möglichkeiten, beispielsweise den Gemini Blueprint Server von Eclipse oder Glassfish von Oracle [51].

#### Vorteile

Die OSGi Serviceplattform bietet eine Reihe von Vorteilen, die dazu führten, dass sie sich als Komponententechnologie etabliert hat. Durch die Verwendung von Bundles werden, wie zuvor erwähnt, fachliche oder technisch zusammenhängende Einheiten zusammengefasst. Macht ein Bundle Gebrauch von einem anderen, so werden diese Abhängigkeiten in der Manifest-Datei festgehalten, die Implementierung des verwendeten Bundles, kann jedoch nicht eingesehen werden. Dadurch entsteht eine Trennung zwischen der öffentlichen Schnittstelle und einer internen Implementierung [52]. Durch diese Aufteilung unterstützt OSGi den Entwickler dahingehend, dass ein Bundle nur inklusive der Abhängigkeiten betrachtet wird, jedoch ohne deren Implementierung zu sehen. Dennoch führt diese Trennung nicht dazu, dass Bundles nicht miteinander interagieren, da sie über die öffentliche Schnittstelle immer noch verbunden sind. Da die Komposition automatisch über die Module-Schicht erfolgt, erhöht dies nicht die Komplexität für den Entwickler.

Die OSGi Serviceplattform ermöglicht, dass man Bundles während der Laufzeit der gesamten Anwendung, einfach installieren oder bearbeiten kann. Das sogenannte *Hot Deployment* hat der Vorteil, dass man den Server weder stoppen noch neu starten muss. Dies ist vor allem von großem Nutzen, wenn man Software für Systeme entwickelt, die ständig Verfügbar sein müssen. OSGi bietet auch Übersicht durch die Service Registry, denn diese dient der Vereinheitlichung und Standardisierung. Dies erfolgt nicht nur durch vordefinierten Services, sondern auch durch vom Entwickler implementierte Services. Die Service Registry stellt diese in der kompletten Anwendung zur Verfügung, wodurch Komponenten nur gegen die Schnittstelle eines Services arbeiten müssen ohne dessen Implementierung zu kennen [52].

#### Nachteile

OSGi hat mit ihren Bundles ein interessantes Konzept, jedoch sind diese nicht nur von Vorteil. Wenn das Framework gestartet wird, dann werden zunächst alle Bundles geladen und registriert. Besitzt die Software jedoch eine Vielzahl an Bundles, so kann es zur Verzögerung beim Start kommen, da das Laden der Bundles umso länger dauert, je mehr Bundles es gibt. Auch Services weisen Probleme auf. Wird ein Service in der Service-Registry registriert, werden alle vom Service benötigten Klassen und Objekte geladen. Dabei wird allerdings nicht berücksichtigt, ob dieser Service überhaupt verwendet wird oder nicht. So kommt es zu Verschwendung von Speicherressourcen, wenn mehrere Services zwar in der Service-Registry eingetragen sind, jedoch nicht verwendet werden [52]. Ein weiterer Nachteil ist, dass OSGi ursprünglich auf die Entwicklung von *residential internet gateways* ausgelegt wurde. Da diese die Maschine zu Maschine Kommunikation innerhalb eines Gebäudes ermöglichen, ist OSGi nicht für einen Endnutzer ausgelegt. Für den eigentlichen Zweck von OSGi war dies auch nicht notwendig, da aber mittlerweile auch Applikationen für Endnutzer entwickelt werden, stellt dies ein Problem dar [23].

#### Anwendung

Wie bereits erwähnt wurde OSGi für *residential internet gateways* entwickelt, also für die Gebäudeverwaltung durch Vernetzung von technischen Geräten. Jedoch gibt es auch noch andere Bereiche, in denen OSGi verwendet wird. So ist zum Beispiel Eclipse seit Version 3.0 mit OSGi realisiert. Dadurch, dass die Technologie die Grundlage von Eclipse bildet, ist sie in den Bereich der Desktop-Applikationen gekommen. Ein weiterer Bereich, in dem OSGi zum Einsatz kommt, sind Serverseitige-Anwendungen. Ein bekanntes Beispiele ist hier die Versionsverwaltungssoftware Version Cue von Adobe, welche zum Austausch von Artefakten innerhalb der Abobe Creative Suite dient. Außerdem wird OSGi von IBM für den WebSphere Application Server verwendet. Da sich OSGi auch für eingebettete Systeme eignet findet die Technologie auch Anwendung in der Automobilbranche. So bildet die Service Platform die Grundlage der Telematik- und Informationssyteme in dem 5er Modell von BMW. [52]

### 3.2.3 .NET

Die Softwareplattform .NET wurde von Microsoft für Windows entwickelt. Dennoch ist es durch das quelloffene Projekt Mono [31] möglich die Plattform von anderen Betriebssystemen aus zu nutzen. Für .NET realisierte Microsoft neben Visual Studio, das als Entwicklungsumgebung dient, auch C#, welche die meist verwendete Programmiersprache für .NET ist. Allerdings ist es auch möglich eine andere Programmiersprachen zu verwenden, denn .NET ist programmiersprachenunabhängig. Die Plattform ist nicht für einen bestimmten Bereich der Softwareentwicklung realisiert worden, sondern beabsichtigte von vornherein die Entwicklung von Internet-Anwendungen sowie lokalen PC-Anwendungen und Eingebetteten Systemen zu vereinen.

#### Architektur

So wie es für eine Komponententechnologie typisch ist, bildet auch hier das .NET-Framework das Herzstück der .NET-Technologie. Die Abbildung 3.3 stellt die Architektur des .NET-Frameworks dar, dessen Komponenten im folgen erläutert werden.

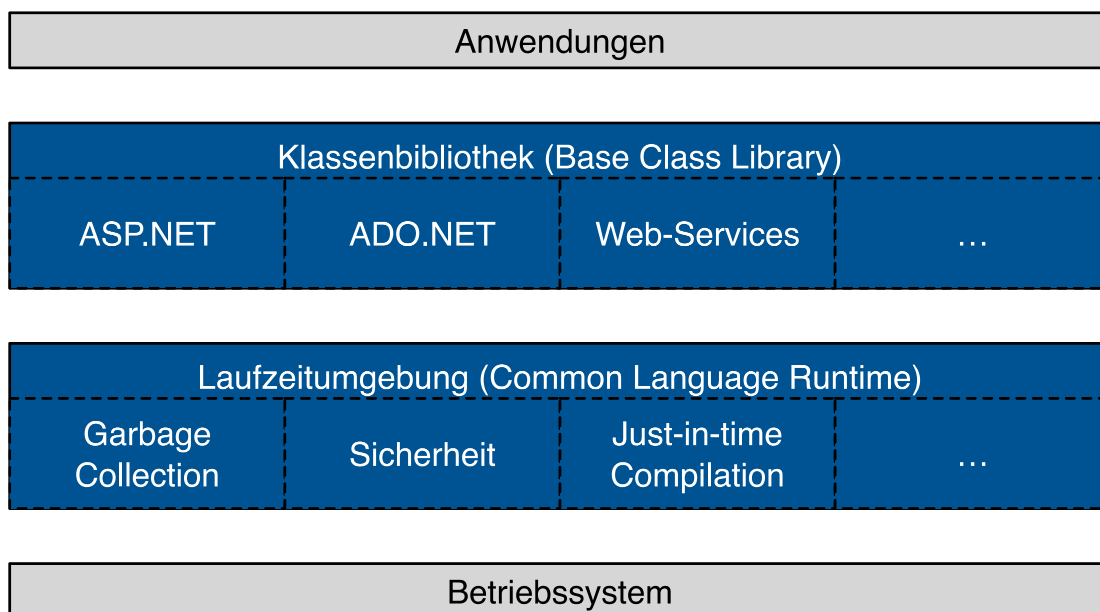


Abbildung 3.3: Grobarchitektur des .Net-Frameworks (Anlehnung an [2])

**Common Language Runtime (CLR)** Beer, Birngruber, Mössenböck, Prähofer und Wöss erläutern die CLR in ihrem Buch *Die .NET-Technologie* [2] als Laufzeitumgebung, die für die Ausführung der .NET-Programme zuständig ist.

Die CLR stellt die Common Intermediate Language (CIL) und die Just-in-time (JIT)-Kompilierung dar.

**Common Intermediate Language** Damit die Funktionalität des .NET-Framworks trotz der Möglichkeit verschiedene Programmiersprachen zu verwenden, gewährleistet wird, gibt es die sogenannte CIL. Diese übersetzt alle eingesetzten Programmiersprachen und um dies zu realisieren, wird durch die CIL außerdem ein gemeinsames Typsystem (Common Type System (CTS)) definiert. Das CTS stellt Anforderungen bezüglich der Darstellung von Klassen, Interfaces und elementaren Typen [2]. Damit eine Programmiersprache für die .NET-Technologie verwendet werden kann, muss diese die Common Language Specification (CLS), die ein Teil des CTS ist, unterstützen.

**Just-in-time-Kompilierung** Die JIT-Kompilierung kompiliert Programme bereits während der Laufzeit, um die Ausführungsgeschwindigkeit zu erhöhen. Sie ist somit für die effiziente Ausführung der Programme zuständig.

**Assemblies** In .NET werden Komponenten, als Assemblies bezeichnet und Klassen und Resources umfassen (ähnlich zu Bundles in OSGi). Schnittstellenbeschreibungen der zu den Assemblies gehörigen Klassen, Methoden und ähnlichen, sind als Metadaten festgehalten. Um zu gewährleisten, dass Assemblies nicht unbeabsichtigt überschrieben werden, sind sie über Versionsnummern eindeutig identifizierbar.

**Global Assembly Cache** Microsoft [29] definiert den Global Assembly Cache als Zwischenspeicher von Assemblies, damit diese im System von verschiedenen Anwendungen verwendet werden können. Falls dies aber nicht notwendig ist, da sie nicht von mehreren Anwendungen benutzt werden, ist es ausreichend diese in das Anwendungsverzeichnis zu kopieren.

**Base Class Library (BCL)** Die Klassenbibliothek, wird in .NET als BCL bezeichnet. Alle Klassentypen sind in .Net in Namensräume (namespaces) gegliedert, beispielsweise enthalten System.Collections unter anderem Objekte wie Bäume, Listen und Arrays. Um Dateizugriff in .NET zu gewährleisten, gibt es ADO.NET. Dies ist eine Sammlung von Klassen, Strukturen und Interfaces, welche für den Zugriff auf verschiedene Datenbanken zuständig ist [26]. ADO.NET ist eine Objektrelationele Abbildung und ist somit zwar unabhängig von der Datenbanktechnologie, beschränkt sich jedoch auf das relationale Datenmodell. Des weiteren gibt es noch das Framework ASP.NET, welches auf die Web-Entwicklung spezialisiert ist. Um Web-Seiten und mobile Seiten zu erstellen, können die Standard Web-Sprachen wie HTML, CSS und JavaScript verwendet werden, aber auch Programmiersprachen wie C# [28].

### Notwenige Infrastruktur

Um Software mit der .NET-Technologie zu entwickeln bedarf es einer technologischen Infrastruktur. Man benötigt ein .NET-SDK, welches die Funktionalitäten, Werkzeuge und Anwendungen des .NET-Frameworks beinhaltet. Die aktuellste Version des .NET-SDKs ist zur Zeit 4.5[2]. Wenn man .NET für die Web-Entwicklung verwenden möchte, so wird auch ein Webserver benötigt, um die .NET-Anwendung an Browser zu übertragen.

### Vorteile

Ein großer Vorteil ist die Programmiersprachenunabhängigkeit, die .NET von anderen Technologien wie JavaEE und OSGi unterscheidet. Ein weitere Vorteile sind Sicherheit und Robustheit [2]. Der zuvor genannte CIL-Code ermöglicht die Programmiersprachnunabhängigkeit in dem er für Typprüfung und Verifikation zuständig ist. Dadurch werden beispielsweise unerlaubte Zugriffe auf fremde Speicherbereiche oder Zeigermanipulation unterbunden. Ein Administrator kann festlegen, welche Benutzergruppen welchen Code ausführen dürfen und kann außerdem Zugriffsrechte für einzelne Benutzer einschränken. Auch bei den Assemblies ist das Thema Sicherheit beachtet worden. Zum Beispiel hat die zuvor erwähnte Versionierung den Effekt, dass keine Assemblies versehentlich überschrieben werden können. Des Weiteren können diese über ein Public-Key Verfahren signiert werden, wodurch die Wahrscheinlichkeit, dass sich Viren verbreiten verringert wird. Somit können Entwickler verifizieren, dass der Code tatsächlich von ihnen stammt und nicht durch unbefugte Personen verändert wurde. Durch das CTS ist ein Typsystem vordefiniert, an welches sich die Programmiersprachen halten müssen, um in .NET verwendet zu werden. Somit liefert das CTS einen Vorteil im Bezug auf die Robustheit und überlässt dies nicht dem Programmierer. Ein weiterer Vorteil ist, dass .NET Desktop- und Web-Entwicklung mit einander vereint und somit einheitlicher gestaltet. Es besteht beispielsweise die Möglichkeit für Web-Anwendungen C# zu verwenden und nicht notwendigerweise die herkömmliche Skriptsprache JavaScript. Dadurch besteht die Möglichkeit auf die gesamte Klassenbibliothek (BCL) zuzugreifen.

### Nachteile

Bei den Nachteilen von .NET steht besonders im Fokus, dass .NET von Microsoft entwickelt wurde und somit ursprünglich für Windows ausgelegt ist. Obwohl auch andere Betriebssysteme die Technologie verwenden können, ist die volle Funktionalität nur für Windows verfügbar [47]. Diese Betriebssystembindung spielt auch bei der Wahl eines Webservers eine Rolle, denn auch hier ist man an die Server von Microsoft gebunden. Es gibt Alternativen zu den Servern von Microsoft, jedoch sind diese in ihrer Entwicklung noch nicht so ausgereift, wie die Server von Microsoft [48]. Mit Visual Studio liefert Microsoft eine Entwicklungsumgebung für .NET, die es auch als kostenlose

[2]Stand Oktober 2014

Version gibt. Möchte man jedoch den vollen Funktionsumfang nutzen, benötigt man die kostenpflichtige Version von Visual Studio. Ein weiterer Punkt, den man bei der Wahl von .NET betrachten sollte ist, dass PHP bei Web-Anwendungen die meist verwendete Skriptsprache ist [36], .NET dies jedoch nicht unterstützt. Durch die Bekanntheit von PHP sind die meisten Web-Hoster PHP-Basiert, wodurch man auch bei der Wahl eines solchen Hosters eingeschränkt ist [48].

#### Anwendung

Im Folgenden werden ein paar Projektbeispiele vorgestellt, in denen .NET verwendet wurde.
Die Autoversicherungsgesellschaft *GEICO*[3] entwickelte mit .NET eine App für das Windowsphone. Diese App sollte den Kunden Service verbessern, indem über die App beispielsweise Rechnungen bezahlt werden können oder bei einem Unfall schnell Hilfe angefordert werden kann. Laut eines *Case Studies* Berichts von Microsoft [25] war die Entwicklung der Windows Phone App vierzig Prozent schneller, als die Entwicklung der iPhone App.
Ein weiteres Beispiel liefert das Telekommunikationsunternehmen *Telefonica*, welches die Entwicklungsplattform BlueVita mit Hilfe der .NET-Technologie entwickelte. Diese Platform soll Entwicklern die Möglichkeit bieten Telekommunikationsfunktionen für ihre Anwendungen hinzufügen zu können [27]. In Kooperation mit Microsoft wurde das BlueVita SDK entwickelt, welches für die .NET-Sotwareplattform genutzt werden kann.
Es gibt noch zahlreiche weitere Beispiele von Firmen, die Softwarelösungen mit der .NET-Technologie realisiert haben. Einige dieser Beispiele lassen sich auf der Seite von Microsoft .NET unter der Rubrik *Case Studies*[4] finden.

### 3.2.4 CORBA

Die Komponententechnologie Common Object Request Broker Architecture (CORBA) wurde durch die Object Management Group (OMG) realisiert. CORBA kombiniert das Client-Server-Modell mit der objektorientierten Programmierung und ermöglicht dabei die Zusammenarbeit von Softwarekomponenten und Objekten der Anwendungen [12]. Außerdem handelt es sich bei CORBA um eine betriebssystemunabhängige Technologie, somit können die mit CORBA realisierten Anwendungen auf beliebigen Plattformen programmiert und ausgeführt werden.

#### Architektur

Die Architektur von CORBA baut auf dem Client-Server Modell auf. Der Server beinhaltet das CORBA-Objekt, welches das zentrale Konzept von CORBA darstellt. Das CORBA-Objekt unterscheidet sich kaum vom herkömmlichen Objekt-Begriff, jedoch werden hier auch Typen, Schnittstellen, Operationen und Attribute genauer

[3] Government Employees Insurance Company
[4] `http://www.microsoft.com/net/casestudies`

qualifiziert [49]. Die Implementation eines CORBA-Objekts wird als Servant bezeichnet. Ein Servant ist für ein oder mehrere Objekte zuständig und implementiert die Funktionalität des Interfaces eines Objekts. Der Servant kann in verschiedenen Programmiersprachen implementiert sein, wie C, C++, Java oder andere. Damit der Klient die Dienste eines CORBA-Objekts nutzen kann, stellt das Objekt seine Dienste über ein Medium zu Verfügung, den Object Request Broker (ORB). Diese Dienste werden vom Klienten genutzt, in dem Operationsaufrufe über den ORB gesendet werden. Der ORB, ist somit die zentrale Vermittlungsstelle zwischen Klient und Server. Er ist ein Softwarebus, auf dem sich CORBA-Objekte registrieren können, um so ihre Dienste den restlichen mit dem Bus verbundenen zur Verfügung stellen können. Bei dem ORB handelt es sich um Message Oriented Middleware (MOM), da dieser für die Codierung und Decodierung von Netzwerknachrichten zur Übermittlung von Operationsaufrufen und Operationsantworten verantwortlich ist und grundlegende unterstützende Laufzeitdienste für CORBA-Objekte bereit stellt. [32]
Der Zugriff über den ORB wird auf zwei Arten ermöglicht (vgl. Abbildung 3.4), eine statische Anfrage und eine dynamische Anfrage.

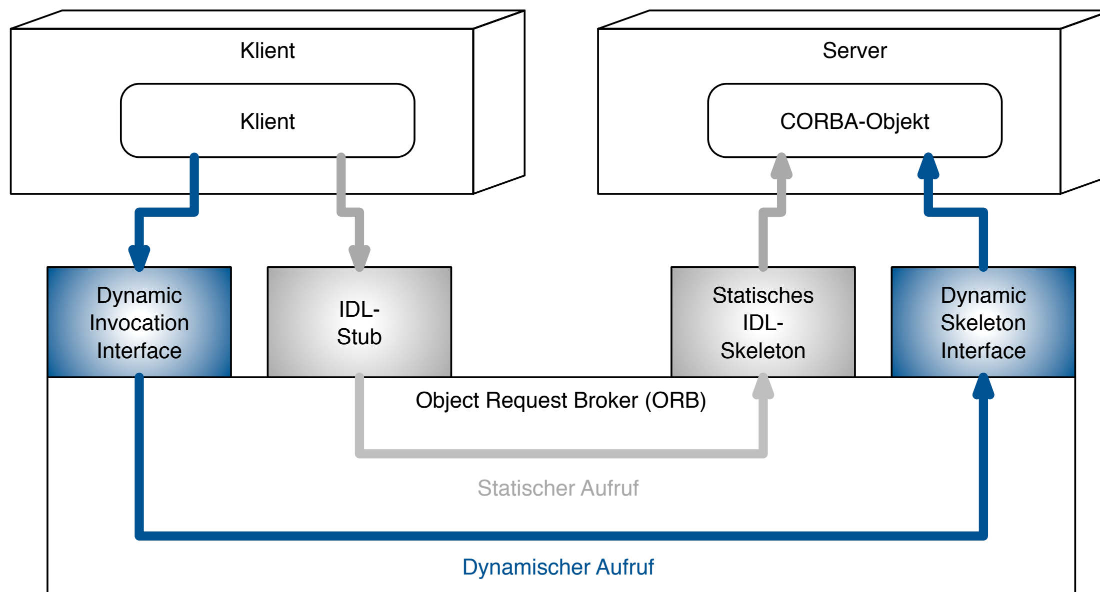


Abbildung 3.4: Statischer & Dynamischer Aufruf eines CORBA-Objekts (Anlehnung an [12] und [32])

**Statischer Aufruf** Der statische Aufruf erfolgt mit Hilfe der Interface Definition Language (IDL). Die IDL beschreibt Datentypen, Interfaces und zugehörige Optionen und Parameter, um den Zugriff über das Interface zu standardisieren. Sie verdeckt die Implementierung für Dienstnehmer, wodurch es auch keine Rolle spielt, welche Programmiersprache verwendet wird oder auf welchem

Betriebssystem der Server läuft. Für den Operationsaufruf auf der Klientenseite sind die IDL-Stubs zuständig, die streng typisiert sind, somit also der Typ des aufgerufenen Objekts bekannt sein muss. Auf der Serverseite ist der IDL-Skeleton verantwortlich, welcher analog zum IDL-Stub funktioniert. [12]

**Dynamischer Aufruf** Ist der Typ jedoch nicht bekannt, so kann auch ein dynamischer Aufruf über ein vom Zielobjekt unabhängiges Interface geschehen. Hierbei wird von der IDL kein Gebrauch gemacht, der Aufruf erfolgt über das Dynamic Invocation Interface (DII). Auf der Serverseite ist das Dynamic Skeleton Interface verantwortlich. [32]

Des Weiteren existiert eine Reihe von so genannten Common Object Services, die vom CORBA Standard definiert werden. Diese Services sind Dienste die häufig benötigte Aufgaben in verteilten Anwendungsumgebungen realisieren. [32]
Um in CORBA Komponenten zu Verwenden wurde auf CORBA 3.0 ein Komponentenmodell aufgesetzt, das CORBA Component Model (CCM). Hier mit können CORBA-Komponenten erstellt werden, die die Eigenschaften Unabhängigkeit, Komposition und Zustandslosigkeit erfüllen. Die Schnittstellen, mit denen eine Komponente nach außen kommunizieren kann, sind in fünf verschiedene Portarten unterteilt [32]:

**Komponenteninterface** Das Komponenteninterface ist ein von der CORBA-Komponente bereitgestellte Schnittstelle mit grundlegenden Operationen. Dieses Interface wird von jeder Komponente angeboten.

**Facets** Facets sind Schnittstellen, die der CORBA-Komponente ermöglichen, ihren Clients verschieden Informationen bereit zu stellen [14].

**Receptacles** Hier werden die Komponenten definiert, die für die Operationen dieser Komponente benötigt werden. Es handelt sich somit um die benötigten Schnittstellen.

**Event sources** Durch die Event sources können Ereignisse nach außen übertragen werden, die aufgetreten sind, währen die Komponente ihre Operationen ausgeführt hat. Solche Ereignisse können beispielsweise Zustandsänderungen von Objekten sein.

**Event sinks** Bei den event sinks handelt es sich um die definition derer event sources von anderen Komponenten, die diese Komponente beobachtet.

Der sogenannte CORBA-Container ist die Laufzeitumgebung einer CORBA-Komponente und verwaltet somit die Komponenten während ihrer Ausführung [32].

### Notwendige Infrastruktur

Um ein Softwareprojekt mit Hilfe von CORBA zu realisieren benötigt man zunächst einen ORB, also eine CORBA Implementation. Hier gibt es kostenlose Varianten wie Orbacus [24] oder TAO [39]. Da CORBA programmiersprachenunabhängig ist, muss man sich noch für eine Programmiersprache entscheiden. Des Weiteren benötigt man einen Server für die Serverseite, auf dem das Servant-Objekt liegt. Als Klienten-Seite muss das Endgerät eingerichtet werden, damit dies auf den Server zugreifen kann. Außerdem muss die IDL spezifiziert werden, welches mit Hilfe des IDL-Files geschieht, damit sowohl für Klienten als auch für den Server die Objekteigenschaften bekannt sind. [8] Um das CCM nutzen zu können und somit Komponenten deployen zu können benötigt man einen component server. Dieser verwaltet die Komponenten verwaltet, in dem er deren assembly files lädt [14].

### Vorteile

Um nachzuvollziehen, warum sich CORBA als Komponententechnologie etabliert hat, muss man den Stand der Technik betrachten zu dem CORBA bekannt wurde. Anfang der 90er Jahre, zu der Zeit als CORBA veröffentlicht wurde, war es extrem schwierig zu realisieren, dass Programme, die auf verschiedenen Systemen laufen, miteinander kommunizieren. Dies war gerade dann ein Problem, wenn verschiedene Hardware, Betriebssysteme und Programmiersprachen involviert waren. Um dies zu realisieren mussten die Programmierer Sockets benutzen und den Protokoll-Stack selbst implementieren, da existierende Technologien nicht auf heterogene Umgebungen ausgelegt waren. Außerdem knüpfte CORBA eine Verbindung zwischen dem Client-Server-Modell und der objektorientierten Programmierung. CORBA, war die erste Technologie, die dies ermöglichte und setzte sich dadurch schnell durch. [15]
CORBA vereint also Software die mit verschiedenen Programmiersprachen realisiert wurde und auf verschiedenen Systemen läuft. Somit ist die Entwicklung mit CORBA betriebssystem- und programmiersprachenunabhängig, wodurch Anwendungen auf einer Vielzahl von Plattformen entwickelt und eingesetzt werden [32].

### Nachteile

Heutzutage ist CORBA keine gängige Technologie mehr, da sie Nachteile aufweist, die dazu führten durch andere Technologien ersetzt zu werden. Durch die Entstehung Anfang der 90er Jahre, war CORBA nicht auf die Web-Entwicklung ausgelegt, da zu diesem Zeitpunkt die Notwenigkeit dessen, nicht absehbar war. Da dies auch später nicht behoben wurde, ist CORBA nicht für die Web-Entwicklung geeignet, da diese einfach nicht unterstützt wird.
Ein weiterer Nachteil von CORBA beschreibt Henning in seinem Artikel *The Rise and Fall of CORBA* [15]. Er sagt, dass die APIs von CORBA mit viel mehr Code entwickelt worden sind als nötig wäre und verkomplizieren dadurch die Interaktion mit Programm und CORBA Laufzeit. Außerdem ist das Typsystem zu komplex, da

die IDL sehr viele verschiedene Typen hat. Es gibt hier unter anderem vorzeichenlose Integer, allerdings werden diese nicht von allen Programmiersprachen unterstützt, wie beispielsweise Java. Somit kann es zu einem Speicherüberlauf kommen, wenn diese in einem Interface verwendet werden.
Ein weiterer Grund, warum CORBA nicht mehr häufig verwendet wird, ist, dass es der Technologie an Eigenschaften fehlt. So ist zum Beispiel Sicherheit nicht berücksichtigt worden, was ein Nachteil gegenüber den zuvor erwähnten Technologien ist. Der Datenverkehr ist bei CORBA unverschlüsselt, wodurch Attacken wie beispielsweise Man-in-the-Middle [46] erfolgreich angewendet werden können. Eine weiterer Mechanismus, welcher bei CORBA nicht berücksichtigt wurde, ist die Versionierung. Für die Entwicklung kommerzieller Software, ist diese Funktion jedoch notwendig, um Verbesserungen durch verschiedene Versionen herauszubringen. Dadurch, dass CORBA diese Funktionalitäten nicht bieten konnte, wichen viele Softwareentwicklungsfirmen auf andere Technologien aus. [15]
Somit ist die letzte Version von CORBA 2002 erschienen und ist damit nicht mehr auf dem aktuellen Stand der Technik.

#### Anwendung

Zu dem Zeitpunkt, als CORBA veröffentlicht wurde, war die Technologie eine beliebte Middleware. Allerdings ist sie durch die starke Konkurrenz und die zuvor genannten Nachteile nicht mehr sehr verbreitet. Heute wird sie nur noch innerhalb von Firmennetzwerken benutzt, wo sie durch eine Firewall geschützt ist, um Komponenten zu vernetzten. Außerdem wird CORBA noch in eingebetteten Systemen verwendet oder in verteilten Systemen. [15]
Beispielsweise wird CORBA noch für Telekommunikationsnetzwerke verwendet, unteranderem von Firmen wie Ericsson oder Motorola. Des Weiteren wird CORBA auch als Kontrollsystem verwendet, so zum Beispiel von *Co-flight*, ein Kontrollsystem für den Flugverkehr. Eine Liste von weiteren Beispielen in denen die CORBA-Implementation TAO verwendet wird, kann auf der Seite der Washington University[5] eingesehen werden.

## 3.3 Zusammenfassung

Dieses Kapitel hat den Begriff der Komponente, sowie den der Komponententechnologie definiert. Eine Komponente besitzt die Eigenschaften Unabhängigkeit, Kompositionseigenschaft und Zustandslosigkeit. Eine Komponententechnologie legt die Entwicklung, Ausführung, Komposition und Kollaboration fest und stellt eine Infrastruktur bereit. Anschließend wurden in diesem Kapitel die Komponententechnologien Java EE, OSGi, .NET und CORBA vorgestellt. Auf Basis dieser Einführung der Technologien, können sie nun im Bezug auf die Wartbarkeit genauer untersucht und bewertet werden. Diese Bewertung erfolgt im nächsten Kapitel.

[5] http://www.cs.wustl.edu/~schmidt/TAO-users.html

# 4 Bewertung etablierter Komponententechnologien

Inhalt

4.1 Software-Qualitätsmerkmale . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Bewertung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 Fazit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

In diesem Kapitel werden die zuvor vorgestellten Komponententechnologien anhand von Software-Qualitätsmerkmalen bewertet. Dazu werden zunächst die dafür verwendeten Qualitätsmerkmale definiert.

## 4.1 Software-Qualitätsmerkmale

Software-bezogene Qualität bezeichnet die Beschaffenheit von Software und ihrer Entwicklung. Die verschiedenen Software-Qualitäten werden anschaulich, von J. Ludewig und H. Lichter [22] durch den Qualitätenbaum dargestellt (siehe Abbildung 4.1). Die Software-Qualitäten werden in den Entwicklungsprozess der Software und in das fertige Softwareprodukt unterteilt. Bei der Produktqualität wird noch zwischen Wartbarkeit und Brauchbarkeit unterschieden. Der Schwerpunkt der Bewertung liegt, wie in der Einführung (vgl. Kapitel 2) erwähnt, auf Merkmalen, die sich auf die Wartbarkeit bereits Implementierter Software beziehen. Der Grund für diese Wahl liegt darin begründet, dass hier Bezug auf Komponenten der Software genommen wird. Da Komponenten durch ihre Unabhängigkeit die Wartbarkeit erhöhen, gilt es zu überprüfen, ob die Komponententechnologien dies auch erfolgreich durchsetzten. Es wird im Abschnitt 4.2 Software betrachtet, die mit einer bestimmten Komponententechnologie entwickelt wurde und auf Änderbarkeit, Testbarkeit und Portabilität überprüft.

### 4.1.1 Änderbarkeit

Die Änderbarkeit bezieht sich auf den Aufwand, Anforderungen eines Systems zu verändern oder es an eine andere Umgebung anzupassen. Nach Ludewig und Lichter [22] ist die Änderbarkeit gut, wenn Strukturiertheit, Simplizität, Knappheit und Lesbarkeit hoch sind. Dabei definieren sie die Begriffe wie folgt:

**Strukturiertheit** Hoch, wenn die Software in logisch abgeschlossene Einheiten mit hohem Zusammenhalt und geringer Kopplung gegliedert ist.

**Simplizität** Hoch, wenn in der Software nur wenige schwer verständliche Konstruktionen enthalten sind.

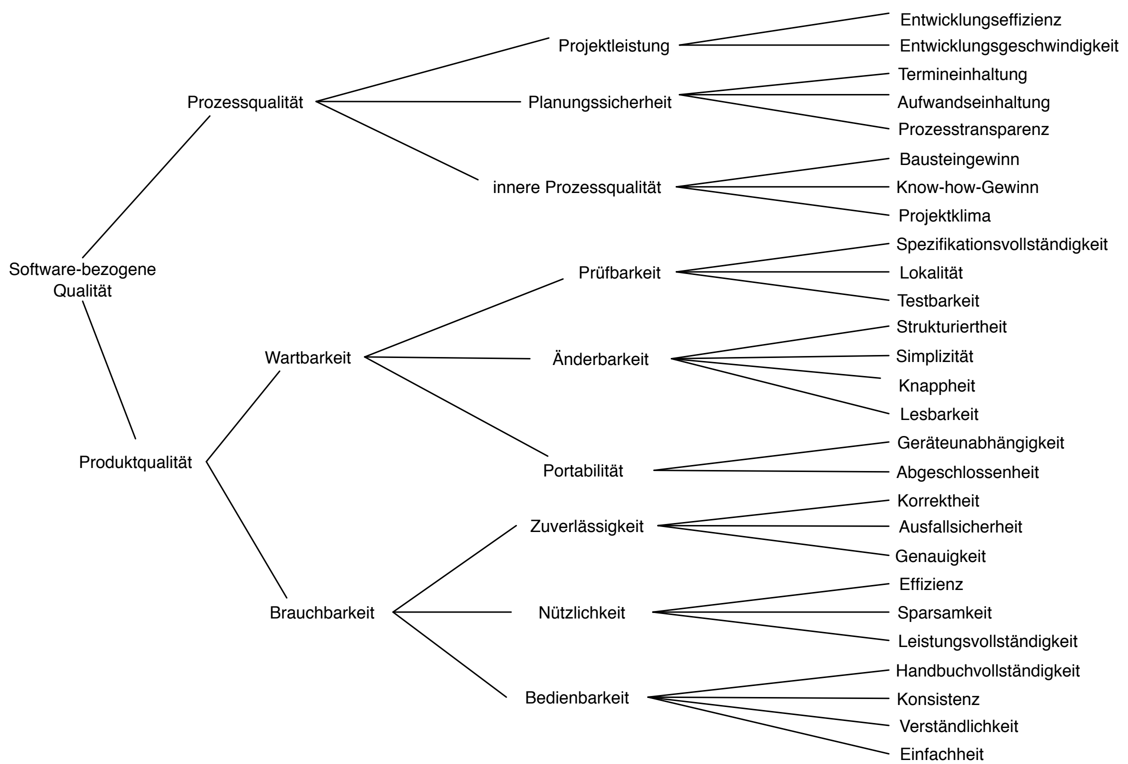


Abbildung 4.1: Qualitätenbaum [22]

**Knappheit** Hoch, wenn ihr Umfang durch Vermeidung von Redundanzen aller Art gering gehalten wurde.

**Lesbarkeit** Hoch, wenn der Leser in der Lage ist, mit minimalem Aufwand den Inhalt korrekt zu erfassen.

Lassen sich die Komponenten einer Software leicht austauschen, da nur die Schnittstellen zu anderen Komponenten angepasst werden müssen, so erhöht dies auch die Änderbarkeit. Komponenten verbessern die Strukturiertheit, da diese abgeschlossen sind und nur durch ihre Interface gekoppelt sind. Somit sollen die Komponententechnologien auf die Strukturiertheit ihrer Komponenten überprüft werden, um zu sehen wie abgeschlossen und gering gekoppelt diese sind.

### 4.1.2 Testbarkeit

Die Testbarkeit bezeichnet den Aufwand zu verifizieren, ob die Software ihre Funktionalitäten korrekt erfüllt. Eine hohe Testbarkeit ist erreicht, wenn die Programme unter definierten Bedingungen ausgeführt und die relevanten Resultate vollständig erfasst werden können [22]. Nach der Norm ISO/IEC 9126 [50] wird Testbarkeit als

der Aufwand bezeichnet, der notwendig ist um geänderte Software zu prüfen. Hier wird betrachtet, wie leicht die einzelnen Komponenten getestet werden. Da Komponenten unabhängig sind, sollten diese einzeln getestet werden können und nicht nur die Anwendung als vollständige Komposition.

### 4.1.3 Portabilität

Portabilität oder auch Übertragbarkeit bezeichnet die Leichtigkeit Software in andere Umgebungen übertragen zu können [50]. Gruhn und Thiel [12] definieren Portabilität mit der Frage, nach einer Plattformunabhängigkeit, also ob die Komponenten portabel auf mehreren Betriebssystemplattformen einsetzbar sind. Und wie hoch ist die Abgeschlossenheit der Software sowie ihrer einzelnen Komponenten? Das heißt erbringt die Software eine gut abgegrenzte Leistung und hat damit kaum Schnittstellen zu anderen Systemen [22]?

## 4.2 Bewertung

Komponententechnologien erhöhen durch ihr Komponentenkonzept die Wartbarkeit von Software. Durch einzelne Komponenten hat Software bessere Änderbarkeit, da Komponenten ausgetauscht werden können. Die Testbarkeit ist höher, denn Komponenten können durch ihre Abgeschlossenheit einzeln getestet werden. Auch die Portabilität ist durch die Abgeschlossenheit der einzelnen Komponenten höher. Dieser Abschnitt beschäftigt sich mit der Fragestellung, ob die Komponententechnologien dies wirklich optimal umsetzten.

### 4.2.1 Java EE

Bei Java EE wird die Softwarestruktur in verschiedene Container unterteilt (Applet, Web, EJB und Application Client Container). Dieses Container-Konzept verbessert die Wartbarkeit der Software. Durch EJBs Komponente wird innerhalb des Java EE Servers standardisiert.

**Änderbarkeit** Die Änderbarkeit wird durch das Container-Konzept verbessert. Bei großen komplexen Softwareprojekten garantieren die Container eine hohe Strukturiertheit und Anpassungen von Funktionalitäten beschränken sich meist nur auf einen Container. Handelt es sich aber um ein kleines Projekt, so werden dennoch diese Container benötigt. Dies beeinträchtigt die Lesbarkeit, da sich viel Code für wenig Funktionalität ergibt. So bringt das Container-Konzept bei kleinen Anwendungen eher Probleme mit sich, als dass es die Änderbarkeit verbessert. Die EJBs bieten zwar eine Standardisierung, allerdings keine hohe Simplizität. EJBs sind umfangreich und komplex spezifiziert und der EJB-Code ist sehr komplex. Mikhalenko [30] schildert dies in seinem Artikel über EJBs als Nachteil, dass ein einfaches *Hello World*-Programm schnell statt nur einer, gleich Zehn Dateien

umfassen kann. Diese Komplexität geht zu Kosten von Simplizität, Knappheit und Lesbarkeit.

**Testbarkeit** Die Testbarkeit von Java EE beziehungsweise ihrer EJB Komponenten ist ein oft diskutiertes Thema (bspw. [42] oder [17]). Taboada [42] erläutert in seinem Artikel, dass die Logik sehr eng mit der Infrastruktur vermengt ist, wodurch es nicht sinnvoll ist einzelne Artefakte zu testen. Außerdem dauert es, bis der Applikation Server gestartet ist, wodurch von den Entwicklern weniger getestet wird, da sie immer erst diesen Applikation Server starten müssen. Zum Testen wird allerdings der EJB-Container bzw. der Application Server immer benötigt [17].

**Portabilität** JavaEE ist Plattformunabhängig, was somit eine höhere Portabilität bedeutet. Außerdem ermöglichen die Container eine hohe Abgeschlossenheit, da sie nur über ihre Schnittstellen mit einander gekoppelt sind. Allerdings wird für EJBs immer der EJB-Container bzw. der Application Server benötigt, wodurch die Portabilität der einzelnen Komponenten beeinträchtigt ist.

### 4.2.2 .NET

Die im vorherigen Kapitel (vgl. 3.2.3) beschriebene CIL ermöglicht es in .NET verschiedene Programmiersprachen zu benutzen. Allerdings sollte man dabei auch berücksichtigen, dass die Wartbarkeit verschlechtert wird, wenn eine Anwendung mit verschiedenen Programmiersprachen realisiert wurde.

**Änderbarkeit** Durch die CIL bietet .NET viele Möglichkeiten und man kann bei Anpassungen an Funktionalitäten auch andere Programmiersprachen verwenden, wenn das einen Vorteil hat. Daher ist man nicht darauf angewiesen neue Funktionen in der Programmiersprache nachzurüsten, wie es am Anfang des Projekts entschieden wurde, falls es sich um eine Komponente handelt die nur gering mit anderen gekoppelt ist. So könnte beispielsweise die Verwendung der funktionalen Programmiersprache F# an einer sinnvollen Stelle, die Lesbarkeit und Knappheit erhöhen. Allerdings sollte diese Möglichkeit auch mit Vorsicht genutzt werden, denn durch viele verschiedene Programmiersprachen in nur einer Anwendung erhöht sich auch schnell die Komplexität und Unübersichtlichkeit der Struktur.

**Testbarkeit** Bei .NET-Anwendungen können die Assemblies als logisch abgeschlossene Komponenten getestet werden, wodurch die Testbarkeit erhöht wird. Allerdings kann auch die Verwendung mehrerer verschiedener Programmiersprachen, die Testbarkeit einschränken. Viele verschiedene Programmiersprachen werden auch unterschiedlich getestet, was das Testen unübersichtlicher macht und unnötig verkompliziert.

**Portabilität** Ursprünglich ist .NET für das Windows Betriebssystem realisiert worden, da die Komponententechnologie von Microsoft entwickelt wurde. Zwar besteht

grundsätzlich die Möglichkeit .NET auch auf anderen Betriebssystemen zu nutzen, allerdings sind diese Umsetzungen nicht so ausgereift. Somit empfiehlt es sich das Windows Betriebssytem zu verwenden, was die Portabilität einschränkt.

### 4.2.3 OSGi

Wie im vorherigen Kapitel erwähnt wurde (vgl. 3.2.2), gibt es in OSGi sogenannte Bundles. Durch diese soll die Wartbarkeit von OSGi Anwendungen verbessert werden, ob dies tatsächlich der Fall ist und wie dies geschieht wird im folgenden betrachtet.

**Änderbarkeit** Bundles realisieren einzelne Anforderungen. Ändern sich Anforderungen und muss die Anwendungen angepasst werden, so bezieht sich dies nur auf einzelne Bundles. Sie garantieren außerdem eine hohe Strukturiertheit, da es sich um abgeschlossene Einheiten handelt. Des Weiteren garantiert das im vorherigen Kapitel beschriebene Hot Deployment eine hohe Änderbarkeit. Es ermöglicht, dass einzelne Funktionalitäten leicht geändert werden können, da man Bundles während der Laufzeit installieren oder bearbeiten kann.

**Testbarkeit** Bundles ermöglichen durch Service-Orientierung und lose Kopplung eine hohe Testbarkeit [16]. Auch das im vorherigen Kapitel erwähnte *Hot Deployment* verbessert die Testbarkeit von OSGi. Die Funktionalität einzelner Bundles kann während der Laufzeit getestet werden, in dem diese installiert und bearbeitet werden können, ohne dass der Server dafür gestoppt oder neu gestartet werden muss.

**Portabilität** OSGi Anwendungen werden in Java implementiert und da Java eine plattformunabhängige Programmiersprache ist, sind auch OSGi Anwendungen plattformunabhängig. Was die Portabilität von einzelnen Komponenten angeht, so sollten Bundles auch diesen Aspekt ermöglichen. Allerdings sind Budles nicht so portabel, wie man zunächst annimmt. So trifft Ross Mason [23] die Aussage, dass mache Bundles nicht auf die gleiche Weise funktionieren, wenn man sie in anderen Containern verwendet. Dies macht die Wiederverwendung von Bundles so problematisch, obwohl mit dem Konzept von Bundles die Wiederverwendung bei OSGi ermöglicht werden sollte.

### 4.2.4 CORBA

Michi Henning, ein Entwickler von CORBA, beschreibt in dem Artikel *The rise and fall of CORBA* [15] die Nachteile dieser Komponententechnologie. Dabei sagt er dass CORBAs größtes Problem dessen Komplexität ist, besonders die der APIs. Viele der APIs von CORBA sind weitaus umfangreicher als es notwendig wäre, dadurch ist es auch aufwändiger mit diesen zu interagieren. Welche Auswirkungen dies auf die einzelnen Qualitätsmerkmale hat, wird im folgenden erläutert.

**Änderbarkeit** Da CORBA eine komplexe Technologie ist, hat dies auch Einfluss auf die Änderbarkeit. Simplizität und Knappheit ist hierbei nicht gegeben, wodurch

es umso aufwändiger ist, Anforderungen eines CORBA Projekts anzupassen. Die aufwändigen Schnittstellen in CORBA erfordern auch dementsprechend viel Code bei der Verwendung dieser, was die Lesbarkeit beeinträchtigt.

**Testbarkeit** Eine gute Testbarkeit in CORBA ermöglichen die Servants da diese für die Objekte zuständig sind und die Funktionalität des Interface eines Objekts implementieren, wodurch die Objekte leichter austauschbar sind. Die CORBA-Komponenten lassen sich gut testen, da sie durch diverse Schnittstellen, ihre Funktionalität gut überprüfen lassen. Beispielsweise durch die event sources, können die Ereignisse der Komponenten überwacht werden.

**Portabilität** CORBA ist eine Komponententechnologie, die plattformunabhängig ist, somit weisen CORBA Anwendungen eine hohe Portabilität auf. Allerdings verringern die vielen Schnittstellen einer CORBA-Komponente die Abgeschlossenheit dieser. Durch viele Schnittstellen kann eine hohe Kopplung zu anderen Komponenten bestehen.

## 4.3 Fazit

In diesem Kapitel wurden die Qualitätsmerkmale Änderbarkeit, Testbarkeit und Portabilität definiert. Anschließend erfolgte auf Basis dieser Qualitätsmerkmale eine Bewertung der Komponententechnologien Java EE, OSGi, .NET und CORBA. Generell verbessern die hier betrachteten Komponententechnologien die Wartbarkeit von Software. Allerdings zeigt die Bewertung, dass es möglich ist die Technologien in dieser Hinsicht noch zu verbessern. Ob Microservices diese Verbesserung ermöglicht, soll im Rahmen dieser Arbeit am Beispiel des Betreiber-Backends des CiBo Projekts ermittelt werden. Dazu muss zunächst der Begriff Microservices definiert werden. Das nächste Kapitel beschäftigt sich daher mit den Eigenschaften von Microservices.

# 5 Microservices

Inhalt

5.1 Eigenschaften . . . . . . . . . . 29
5.2 Vorteile . . . . . . . . . . 31
5.3 Nachteile . . . . . . . . . . 31
5.4 Zusammenfassung . . . . . . . . . . 32

Nach und nach gibt es mehr Firmen, die ihre Software durch Microservices realisieren. Besonders interessant ist dabei zu beobachten, dass einige der erfolgreichen Verwender, die Microservices ausgehend von einem Monolith realisiert haben. Bekannte Beispiele sind hier Amazon, eBay und Groupon [37]. Genau das ist auch in dieser Arbeit der Fall. Hier existiert bereits ein Java EE Prototyp, der mit Hilfe von Microservices neu entwickelt werden soll. Aber was sind Microservices? Dieses Kapitel erläutert zunächst die Eigenschaft von Microservices. Anschließend werden Vor- und Nachteile von Microservices gegenüber herkömmlichen Komponententechnologien dargestellt (eine Evaluation von Microservices im Kontext dieser Arbeit erfolgt im Kapitel 9).

## 5.1 Eigenschaften

Fowler und Lewis erläutern in ihrem Artikel *Microservices*, die Eigenschaften von Microservices. Dabei merken sie an, dass eine Anwendung nicht alle Eigenschaften erfüllen muss um als Microservices bezeichnet zu werden. Microservices lassen sich durch sechs Eigenschaften beschreiben.

**Unterteilung in Services** Die Anwendung wird in kleine eigenständige Services unterteilt und verwendet diese als Komponenten. Services sind nach Definition von N. Josuttis [19] IT-Repräsentationen von in sich abgeschlossenen fachlichen Funktionalitäten.

**Schicht übergreifend** Microservices enthalten alle Schichten (Datenbank, Middleware und Frontend) einer Funktionalität. Anwendungen werden somit nicht in Schichten unterteilt sondern in Funktionalitäten. Eine visuelle Darstellung dieser Eigenschaft ist in Abbildung 5.1 zusehen.

**Deployment** Microservices werden eigenständig und nicht als gesamte Anwendung deployed. Dadurch entsteht selbst bei der Implementierung keine Abhängkeit zu anderen Services.

**Kommunikation** Die Kommunikation von Microservices erfolgt meist über HTTP basierte Schnittstellen, wie beispielsweise RESTful-APIs (vgl. Unterabschnitt 7.1.1).

**„Smart endpoints and dumb pipes“** [6] Microservices werden so entkoppelt von anderen Komponenten wie möglich entwickelt. Dabei werden ihre Schnittstellen so realisiert, dass sie leicht eingesetzt werden können. Schnittstellen die allgemein entwickelt werden und nicht genau auf eine Anwendung zugeschnitten sind, soll die Microservices wiederverwendar machen.

**Technologisch entkoppelt** Microservices sind auch technologisch von einander entkoppelt. Programmiersprache und Framework sind frei wählbar und können innerhalb einer Anwendung von Microservice zu Microservice variieren. Des Weiteren werden Microservices von Fowler als Polyglot Persistence [5] bezeichnet. Das bedeutet, dass für die Anwendung nicht eine Datenbank festgelegt wird, sondern für jeden Microservice entschieden werden kann, welche Datenbank verwendet wird. So kann für den einen Microservice eine relationale Datenbank, für den anderen aber eine NoSQL Datenbank verwendet werden.

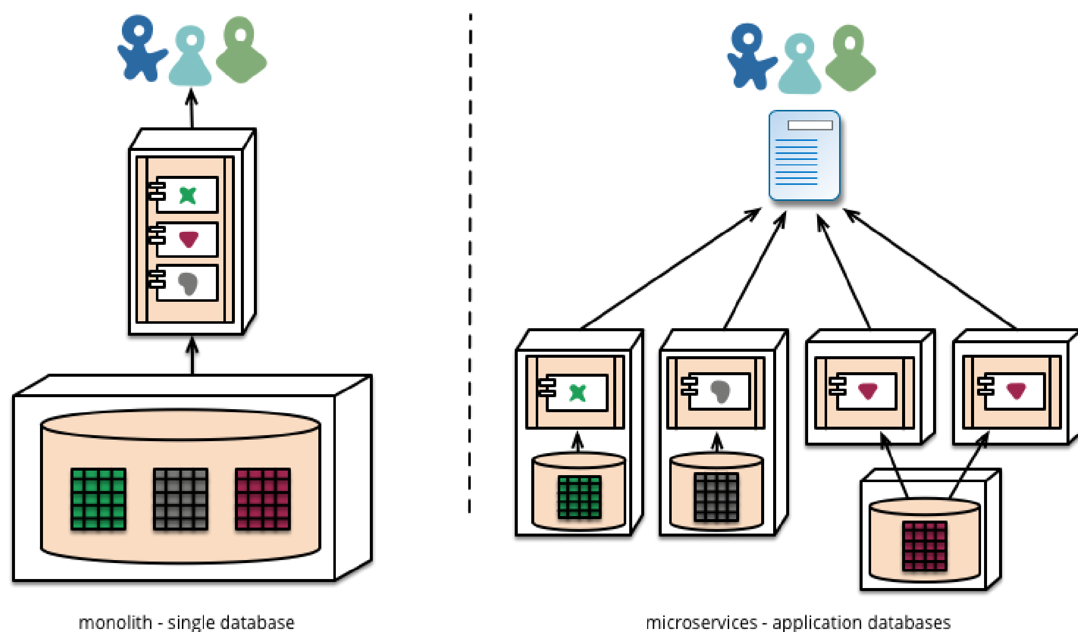


Abbildung 5.1: Vergleich von Monolith und Microservices [6]

## 5.2 Vorteile

Die Entkoppelung der Microservices ist sehr vorteilhaft. Wenn Anforderungen geändert werden und somit der Code angepasst werden muss, sind meist nur einzelne Microservices betroffen, da die Anwendung in Funktionalitäten und nicht in Schichten unterteilt ist. Wie bereits erwähnt werden die Endpoints von Microservices durchdacht entwickelt, so dass sie überall einsetzbar sind. Das von Fowler als „Smart endpoints and dumb pipes“[6] bezeichnete Konzept, ermöglicht eine Verwendung von Microservices ohne Glue Code [44]. Das bedeutet es wird kein Code für die Einbindung von Microservice benötigt, der die Komponente in die Anwendung einbettet. Dies hat außerdem den Vorteil, dass die Microservices auf verschiedenen Servern nur eingesetzt werden, wenn sie tatsächlich nötig sind. Funktionalitäten werden nur dann auf verschiedenen Servern eingesetzt, wenn diese auch benötigt werden, da sie als einzelne Microservices realisiert wurden. Wobei bei einer monolithischen Anwendung immer die gesamte Anwendung auf die verschiedenen Server kopiert wird, auch wenn einzelne Funktionalitäten dort gar nicht verwendet werden.
Ein weiterer Vorteil von Microservices, ist dass eine Anwendung von verschiedenen Entwicklerteams gleichzeitig und unabhängig von einander entwickelt werden kann. Die Teams müssen beispielsweise nicht mit dem deployment auf einander warten und da die Microservices von einander entkoppelt sind, sind die einzelnen Teams auch nicht von einander abhängig.

## 5.3 Nachteile

Microservices ist ein neuer und noch kein vollständig definierter Ansatz. Daher hat man zwar eine große Entscheidungsfreiheit, bei der Entwicklung von Microservices, muss allerdings auch viel mehr über Vorgehensweisen bei der Entwicklung nachdenken, da es keine klaren Vorgaben gibt. Harlem-Law beschreibt dies in seinem Artikel *Microservices - A reality check(point)* [13] damit , dass sich der Entwickler nicht hinter genauen vorgaben einer Komponententechnologie verstecken kann.
Ein weiterer Nachteil entsteht, wenn zu viele verschiedene Technologien für die Entwicklung von Microservices verwendet werden. Zum Beispiels die Verwendung von vielen verschiedenen Programmiersprachen verkompliziert die Anwendung, da beispielsweise Tests oder Struktur der Services immer völlig unterschiedlich sind. Die Verwendung der gleichen Programmiersprache für alle Services fördert die Vereinheitlichung.
Des Weiteren ist es schwierig (besonders im ersten Projekt) die richtige Größe eines Services zu finden. Das dies gerade für Entwickler, die neu mit Microservices arbeiten, eine zentrale Frage ist, zeigt Beispielsweise der Blogeintrag *A microservices implementation retrospective* von Skurrie [40]. Sie Schildert, dass sich für sie bei der Entwicklung die Frage gestellt hat, wie groß ein Microservice sein sollte.

## 5.4 Zusammenfassung

In diesem Kapitel wurden Microservices und ihre Eigenschaften genauer definiert. Dabei wurden Microservices als Ansatz beschrieben, bei dem die Anwendung in Services unterteilt werden und nicht in Schichten. Die Microservices werden dabei so entkoppelt wie möglich realisiert, damit diese leichter austauschbar sind. Auch technologisch sind Microservices unabhängig von einander, da sie mit verschiedenen Programmiersprachen, Frameworks oder Datenbanken realisiert werden können. Anschließend erfolgte in diesem Kapitel eine Betrachtung der Vor- und Nachteile von Microservices.
Bevor die Software auf Basis der vorgestellten Eigenschaften von Microservices entwickelt werden kann, müssen zunächst die Anforderungen an die Software definiert werden. Diese Anforderungsanalyse erfolgt im nächsten Kapitel.

# 6 Anforderungen

Inhalt


6.1 Zentrale Begriffe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.2 Stakeholder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.3 Funktionale Anforderungen . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4 Nicht funktionale Anforderungen . . . . . . . . . . . . . . . . . . . . . . . 37
6.5 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37


In diesem Kapitel werden die Anforderungen an das Betreiber-Backend erläutert, welches im Rahmen dieser Bachelorarbeit entwickelt wurde. Es werden hierbei die funktionalen und die nicht funktionalen Anforderungen der Software betrachtet. Zunächst erfolgen Definitionen zentraler Begriffe und eine Auflistung der Stakeholder des Betreiber-Backends.

## 6.1 Zentrale Begriffe

Im Folgenden werden zentrale Begriffe definiert, die in diesem und in den nächsten Kapiteln immer wieder auftauchen.

**Bus** Alle in dieser Arbeit bezeichneten Busse, sind Omnibusse des öffentlichen Nahverkehrs.

**Buslinie** Eine Buslinie ist ein Bus mit einer eindeutigen Bezeichnung, der eine bestimmte Route abfährt.

**Haltestelle** Eine Haltestelle ist ein Ort, an dem Busse anhalten um Passagiere mitzunehmen oder aussteigen zu lassen.

**Fahrt** Eine Fahrt wird von einem Passagier durchgeführt und beschreibt eine Beförderung mit Hilfe eines Busses von einer Haltestelle zu einer anderen, auf einer bestimmten Buslinie. Eine Fahrt gilt als begonnen, wenn ein Passagier in einen Bus einsteigt und als abgeschlossen, wenn dieser wieder an einer anderen Haltestelle aussteigt.

**System** Als System wird in den Anforderungen das gesamte Betreiber-Backend bezeichnet.

**Web-Interface** Das Web-Interface ist das Frontend des Betreiber-Backends, welches die Schnittstelle zwischen Betreiber und Betreiber-Backend darstellt.

## 6.2 Stakeholder

Stakeholder sind Personengruppen oder Systeme, die mit der Software in Berührung kommen. Das sind Personen, die das Software Produkt später verwenden, Kunden, die das Projekt in Auftrag gegeben haben und Systeme die mit der Software des Projekts interagieren. Für das Betreiber-Backend des CiBo Projekts gibt es folgende fünf verschiedene Stakeholder:

**App** Die App des CiBo Projekts (vgl. Kapitel 2) ist die Schnittstelle zwischen Nutzer und CiBo Projekt. Sie ist zunächst für die Registrierung von neuen Nutzern zuständig. Außerdem sendet sie den Standort an das Tracking & BeOut System und ermöglicht es dem Nutzer Buslinien auszuwählen, um ein E-Ticket zu erhalten.

**Tracking & BeOut System** Das Tracking & BeOut System des CiBo Projekts. Das ist das Teilsystem, welches das Backend der App darstellt und den BeOut Algorithmus enthält.

**Betreiber** Die Personen der ASEAG, für die das Web-Interface entwickelt werden soll. Die jenigen, die Einblick in alle getätigten Fahrten haben dürfen und Haltestellendaten oder das Tarifsystem verwalten.

**Nutzer** Die Nutzer sind die Personengruppe, auf die das CiBo Projekt ausgelegt ist. Sie registrieren sich mit Hilfe der CiBo App Fahrten im System und können anschließend Buslinien auswählen, um ein E-Ticket der ASEAG zu erhalten.

**Payment-Anbieter** Der Payment-Anbieter ist für die Abwicklung der Bezahlung einer getätigten Fahrt zuständig. Über diesen Dienst können Nutzer die Kosten ihrer getätigten Fahrten begleichen. Der Payment-Anbieter interagiert mit der App und dem Betreiber-Backend um einen Nutzer registrieren zu können.

## 6.3 Funktionale Anforderungen

Funktionale Anforderungen die von einander abhängen oder sich innerhalb des gleichen Prozesses befinden, werden in einer Kategorie zusammengefasst. Daher ergeben sich die folgenden Kategorien:

- F1: Registrierung
- F2: Login
- F3: Fahrten speichern
- F4: Fahrtensuche
- F5: Ticketpreis
- F6: Haltestellen

### F1 Registrierung

**F1.1 Das System muss der App die Möglichkeit bieten einen neuen Nutzer zu registrieren.** Ein Nutzer muss in der Datenbank gespeichert werden bevor dieser die volle Funktionalität der App nutzen und somit ein E-ticket für eine Busfahrt erhalten kann.

**F1.2 Bei der Registrierung eines Nutzers muss das System eine eindeutige CiBoId generieren.** Es muss vom System eine CiBoId generiert werden und zusammen mit den anderen Nutzerinformationen gespeichert werden, um den Nutzer systemweit eindeutig zu identifizieren.

**F1.3 Die Registrierung muss unter Angabe des Payment-Accounts des Nutzers erfolgen.** Die Registrierung muss in Verbindung mit einem Payment-Anbieter erfolgen, damit die Bezahlung einer Fahrt gewährleistet werden kann. Dazu muss der Nutzer bei der Registrierung seinen Nutzernamen des Payment-Anbieters angeben.

**F1.4 Wenn der Payment-Anbieter die Korrektheit der Angabe des Payment-Accounts des Nutzers verifiziert hat, muss das System dem Payment-Anbieter die Möglichkeit bieten, den CiBo-Account des Nutzers zu aktivieren.** Ein Nutzer kann nur dann E-Tickets verwenden und somit Fahrten mit Hilfe der App durchführen, wenn er den Status *aktiviert* hat.

**F1.5 Wenn der Payment-Anbieter eine Bezahlung des Nutzers nicht mehr garantieren kann, muss das System dem Payment-Anbieter die Möglichkeit bieten, den CiBo-Account des Nutzers zu deaktivieren.** Der Payment-Anbieter kann den Status eines Nutzer jederzeit zu *deaktiviert* ändern, damit dieser keine E-Tickets mehr erhalten kann. Beispielsweise wenn der Nutzer seinen Payment-Accounts löscht, dann kann die Zahlung über diesen Account nicht mehr erfolgen und er muss vom Payment-Anbieter im CiBo-System gesperrt werden.

### F2 Login

**F2.1 Jederzeit muss das System dem Betreiber die Möglichkeit bieten, sich über das Web-Interface in das System einloggen können.** Unter Angabe eines Benutzernamens und eines Passworts muss sich der Betreiber in das System einloggen können, um die Funktionalitäten des Systems nutzen zu können.

**F2.2 Ist ein Betreiber im Web-Interface eingeloggt muss das System ihm die Möglichkeit bieten, sich jederzeit aus dem Web-Interface wieder ausloggen zu können.**

**F2.3 Ist ein Betreiber im Web-Interface eingeloggt soll das System ihm die Möglichkeit bieten, sein Passwort ändern zu können.**

#### F3 Fahrten speichern

**F3.1** **Nach Abschluss einer Fahrt muss das System dem Tracking & BeOut System die Möglichkeit bieten, die Fahrtdaten in einer Datenbank zu persitieren.** Eine Fahrt ist abgeschlossen sobald das Tracking & BeOut System mit Hilfe des BeOut Algorithmus erkennt, dass der Nutzer den Bus verlassen hat. Die Fahrtdaten, die gespeichert werden sollen beinhalten Informationen über die Buslinie, den Nutzer, die Starthaltestelle mit Zeitstempel sowie die Endhaltestelle mit Zeitstempel.

#### F4 Fahrtensuche

**F4.1** **Ist der Betreiber im Web-Interface eingeloggt muss das System dem Betreiber die Möglichkeit bieten, alle Fahrten aus der Datenbank einsehen zu können.** Das Web-Interface stellt dem Betreiber alle Information zu allen Fahrten, die in der Datenbank vorhanden sind, zur Verfügung. Bei diesen Informationen handelt es sich um Buslinie, Nutzer, Starthaltestelle mit Zeitstempel sowie der Endhaltestelle mit Zeitstempel von jeder Fahrt.

**F4.2** **Ist der Betreiber im Web-Interface eingeloggt muss das System ihm die Möglichkeit bieten, die Fahrten aus der Datenbank filtern zu können.** Der Betreiber kann Fahrten nach Datum, Uhrzeit, Haltestellen, Benutzern oder Buslinien filtern.

#### F5 Ticketpreis

**F5.1** **Wird eine Fahrt aus der Datenbank über das Web-Interface angezeigt, so muss das System den Ticketpreis der Fahrt berechnen.** Zu jeder Fahrt muss ein Ticketpreis auf Basis der Distanz der Fahrt berechnet werden.

**F5.2** **Wird eine Fahrt aus der Datenbank über das Web-Interface angezeigt, so muss auch der Ticketpreis der Fahrt angezeigt werden.**

#### F6 Haltestellen

**F6.1** **Ist der Betreiber im Web-Interface eingeloggt soll das System ihm die Möglichkeit bieten, alle Haltestellen aus der Datenbank einsehen zu können.** Das Web-Interface stellt dem Betreiber Informationen zu allen Haltestellen, die in der Datenbank vorhanden sind, zur Verfügung. Bei diesen Informationen handelt es sich um den Namen der Haltestelle und um die Tarifzone in der sich die Haltestelle befindet.

**F6.2** **Ist der Betreiber im Web-Interface eingeloggt soll das System ihm die Möglichkeit bieten, neue Haltestellen in einer Datenbank persistiert werden können.** Für die Speicherung neuer Haltestellen, muss der Betreiber den Namen der Haltestelle und die Tarifzone der Haltestelle angeben.

## 6.4 Nicht funktionale Anforderungen

**NF1** **Funktionen realisieren mittles Microservices.** Das Backendsystem der Web-Anwendung, welches die oben genannten funktionalen Anforderungen erfüllt, muss mit Microservices realisiert werden.

**NF2** **Microservices in Java.** Die Microservices sollen in der Programmiersprache Java implementiert werden.

**NF3** **AngularJS als Frontendtechnologie.** Für die Realisierung des Frontends muss AngularJS also Technologie zum Einsatz kommen.

## 6.5 Zusammenfassung

In diesem Kapitel wurden die funktionalen und nicht funktionalen Anforderungen des Betreiber-Backends definiert. Die Stakeholder dieser Software sind die CiBo App, das Tracking & BeOut System, der Betreiber, der Nutzer und der Payment Anbieter. Auf Basis der vorgestellten Anforderungen wird im folgenden Kapitel der Entwurf der Software des Betreiber-Backends dargestellt.

# 7 Entwurf

Inhalt


7.1 Schichten . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
7.2 Komponenten . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
7.3 Entitäten . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
7.4 Dynamische Sicht . . . . . . . . . . . . . . . . . . . . . . . . . . 51
7.5 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . 54


Der Softwareentwurf des Betreiber-Backends wird nun im folgenden beschrieben. Dabei wird nach dem Top Down Ansatz [22] vorgegangen. Es werden also zunächst die groben Schichten der Architektur beschrieben und anschließend genauer auf die einzelnen Komponenten eingegangen. Danach werden die Entitäten dargestellt und im letzten Abschnitt wird die dynamische Sicht der Funktionen betrachtet.

## 7.1 Schichten

Eine Übersicht über die Architektur der Lösung des Betreiber-Backendsystems ist in Abbildung 7.1 dargestellt. Hierbei wird zunächst in drei Schichten unterteilt. Diese werden im folgenden genauer beschrieben. Anschließend erfolgt eine Begründung der Architekturentscheidung.

**Extern** Diese Schicht stellt die Verbindung zu anderen Systemen dar. Es handelt sich um externe Systeme, die nicht im Rahmen dieser Arbeit entwickelt wurden sondern schon bereits bestanden. So gibt es hier die Schnittstelle zu einem Paymentanbieter, wodurch ermöglicht wird, dass ein Nutzer seine getätigten Fahrten auch bezahlen kann. Außerdem wird somit das im Rahmen der Arbeit entwickelte Programm in den Kontext von CiBo eingeordnet, da es auch Schnittstellen enthält für die CiBo App und das Tracking & BeOut System. Im folgenden wird nicht genauer auf die Komponenten dieser Schicht eingegangen, da sie nicht innerhalb dieser Arbeit entwickelt wurden.

**Backend** Das Backend enthält die Microservices, welche die funktionalen Anforderungen aus Kapitel 6.3 beinhalten. In dieser Schicht werden Daten empfangen, verarbeitet und weitergeleitet. Des Weiteren ist diese Schicht für die Persitierung und Verwaltung von Daten zuständig. Diese Schicht beinhaltet Komponenten, die von einander entkoppelt sind. Das bedeutet innerhalb diese Schicht gibt es keinen Datenaustausch zu anderen Komponenten. Einzig gibt es Verbindungen zu anderen Schichten.

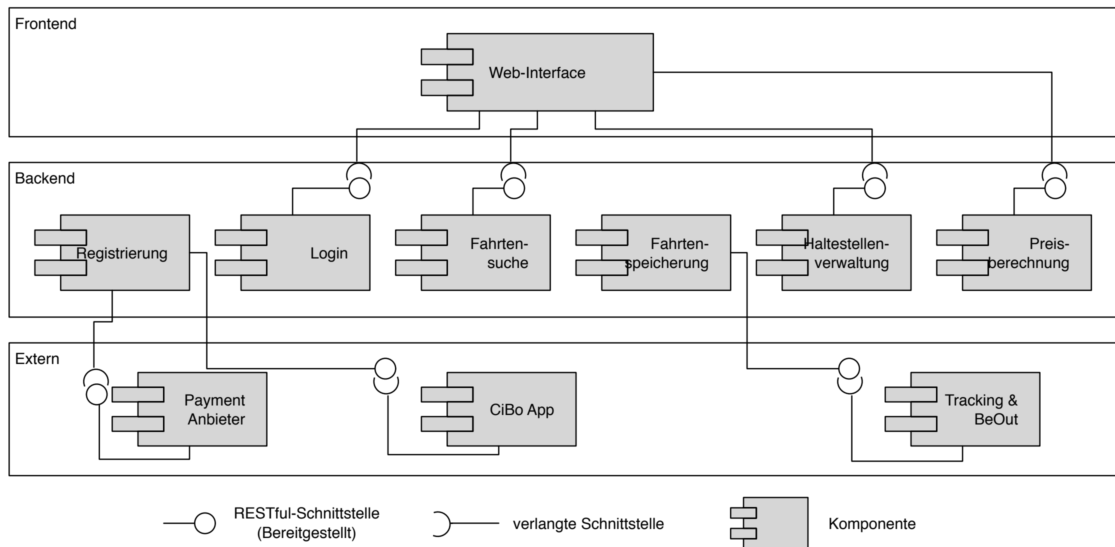


Abbildung 7.1: Schichtendarstellung des Betreiber-Backends

**Frontend** Die oberste Schicht, stellt das Frontend dar. Hier werden zum einen alle Funktionalitäten visuell dargestellt und zum anderen die Backend Services gesteuert. Die Frontend Schicht ist dafür verantwortlich, den Datenfluss zwischen den Microservices zu kontrollieren, da diese von einander entkoppelt und somit nicht miteinander verbunden sein sollen.

Die Einteilung in diese drei Schichten fasst Komponenten mit ähnlichem Verhalten zusammen. Zwar sind die einzelnen Microservices nicht direkt miteinander verbunden, gehören aus fachlicher Sicht jedoch zusammen.
Es gibt verschiedene architektonische Möglichkeiten um so ein Backend umzusetzen. Eine der Varianten ist in Abbildung 7.2 als Option 1 dargestellt.

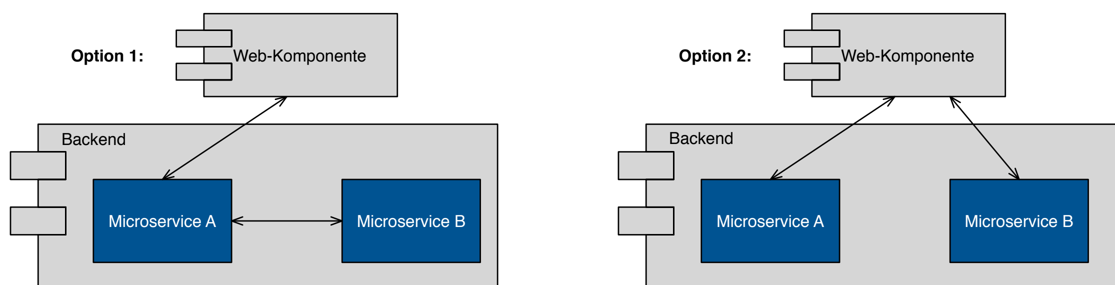


Abbildung 7.2: Architekturoptionen

Es gibt mehrere Services, die innerhalb der Schicht miteinander kommunizieren und einen Service (hier Microservice A), der für die Kommunikation nach außen (bspw.

zu einem Web-Interface) zuständig ist. Je mehr Microservices existieren, desto mehr Kommunikation findet zwischen den einzelnen Services statt, wodurch eine höhere Kopplung entsteht. So kann schnell eine monolithische Struktur entstehen, was jedoch bei Microservices vermieden werden soll. Außerdem wäre dies keine Veränderung im Vergleich mit der bestehenden JavaEE Lösung, die in der Einführung (Kapitel 2) beschrieben wurde. Eine Lösung der Problematik bietet Option 2 des Schaubilds, welche dem Vermittler-Entwurfsmuster [7] entspricht. Wobei die Web-Komponente der Vermittler ist, welcher die Kommunikation steuert. Jeder Microservice kommuniziert eigenständig mit der Web-Komponente und ist somit von anderen Microservices unabhängig. Dadurch wird die Komplexität aus den einzelnen Services des Backends verringert, erhöht sich allerdings in der Web-Komponente. Der Vorteil ist, dass durch diese Lösung die Microservices entkoppelt voneinander sind und leicht ersetzt werden können. Hierbei muss nur das Frontend angepasst werden, jedoch nichts in der Backend-Schicht. Somit ist gewährleistet, dass diese Architektur wirklich als eine Microservice Architektur bezeichnet werden kann. Daher wurde Option 2 gewählt.

### 7.1.1 Kommunikation mit REST

Die Kommunikation zwischen den Komponenten der in Abbildung 7.1 dargestellten Architektur, wird durch sogenannte RESTful-Schnittstellen realisiert.
REST ist ein Architekturprinzip zum Datenaustausch im Internet, welches im Jahr 2000 von Roy Fielding [3] eingeführt wurde. REST ermöglicht die Kommunikation zwischen Webservices mit Hilfe des HTTP. Es werden Schnittstellen erstellt (sog. RESTful-APIs), die mit HTTP-Anfragemethoden definieren, ob Daten gelesen (GET), geschrieben (PUT oder POST), gelöscht (DELETE) oder aktualisiert (PUT oder PATCH) werden. In dieser Arbeit wird für den Datentransfer, der RESTful-APIs die sogenannte JSON verwendet.

## 7.2 Komponenten

Im Folgenden wird genauer auf die Komponenten eingegangen, die in Abbildung 7.1 dargestellt sind. Es werden dabei nur die Komponenten betrachtet, die auch im Rahmen dieser Arbeit entwickelt wurden. Zunächst erfolgt eine allgemeine Beschreibung der Architektur eines Microservices, so wie sie hier in der Arbeit realisiert ist. Die Architektur jedes Microservices einzeln zu beschrieben ist nicht notwendig, da diese alle sehr ähnlich designed sind. Bei den einzelnen Microservices werden dann lediglich die Schnittstellen nach außen kurz erläutert und zur Übersicht jeweils in einer Tabelle dargestellt.

### 7.2.1 Microservice Architektur

In der Abbildung 7.3 ist die Architektur eines Microservices dargestellt. Sie ist in drei Schichten unterteilt, die wie folgt zu beschreiben sind:

**REST-Controller** Die REST-Controller Schicht beinhaltet die Klasse mit den RESTful-APIs. Sie ist somit die Schicht nach außen, die bei den meisten Microservices mit der Web-Komponente verbunden ist. Allerderdings kann je nach Microservice, die externe Komponente auch das Tracking & BeOut System, oder die CiBo-App sein.

**Service** Die Service-Schicht stellt Funktionalitäten für die Klassen der REST-Controller Schicht bereit, um diese zu entlasten und somit Übersichtlichkeit zu gewährleisten.

**Persitierung** Die Persitierungsschicht ist dafür zuständig, Daten aus einer Datenbank zu lesen und neue dort zu speichern. Außerdem stellt diese Schicht die Entitäten zur Verfügung, auf die in Abschnitt 7.3 noch genauer eingegangen wird.

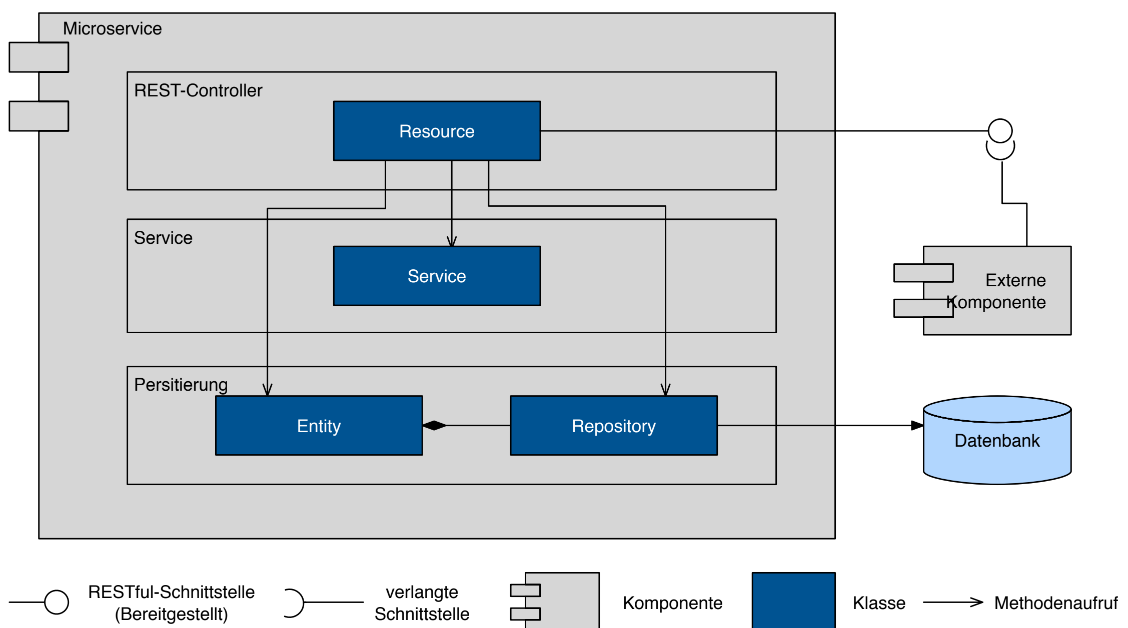


Abbildung 7.3: Komponentendiagramm des Registrierungs Microservice

### 7.2.2 Registrierung

Der Microservice Registrierung bildet die Schnittstelle zur Android App und zum Payment Anbieter. Er erfüllt die funktionalen Anforderungen *F1 Registrierung*. Die Registrierungskomponente ermöglicht es einem Nutzer, sich über die App mit Hilfe eines Payment Anbieters zu registrieren. Nur registrierte Nutzer können Fahrten tätigen, denn nur so ist die Zahlung gewährleistet. Es wird eine eindeutige Id (CiBoId) generiert und ein neuer Nutzer angelegt, welcher anschließend in der Datenbank persistiert wird. Die Services Schicht ist bei diesem Microservice dafür zuständig, eine

Verbindung zu der Schnittstelle des Paymentanbieters herzustellen, um somit auf dessen RESTful-Schnittstelle zugreifen zukönnen. Dadurch kann die Korrektheit des von der App erhaltenden Usernamens verifiziert werden und ein Token für diesen Benutzer angefordert werden.

#### Schnittstellen

Der REST-Controller beinhaltet die RESTful-Schnittstellen für die App. Eine Schnittstelle mit Schreibzugriff ermöglicht es, einen neuen Nutzer in der Datenbank anzulegen. Es wird der Username des Nutzers für den verwendeten Paymentanbieter und die GeräteId des Geräts, auf dem die App läuft, benötigt. Zurückgegeben wird die von der Registrierung generierte CiBoId. Eine weitere Schnittstelle mit Schreibzugriff ermöglicht es dem Payment Anbieter einen Nutzer zu aktivieren oder zu deaktivieren, je nach dem ob er die Möglichkeit hat Zahlungen zu tätigen.

| **Schnittstelle** | **HTTP-Methode** | **benötigte Parameter** | **Rückgabe** |
|---|---|---|---|
| /app/rest/register | POST | *String* paymentUsername<br>*String* deviceId | *int* ciboId |
| /app/rest/activate | POST | *String* paymentUsername<br>*boolean* activated | *int* ciboId |

Tabelle 7.1: Schnittstellen des Registrierungs-Microservice

### 7.2.3 Login

Der Login Microservice erfüllt die funktionalen Anforderungen *F2 Login*. Dieser ist dafür verantwortlich, dass der Betreiber sich in das Backendsystem einloggen kann um Fahrdaten einzusehen. Dafür wird der Benutzername und das Passwort abgefragt. Der Login verifiziert, ob die übergebenen Daten Nutzername und Passwort in der Datenbank vorhanden sind und ob somit Einsicht in die Fahrdaten gewährleistet werden kann. Des Weiteren stellt dieser Service noch Funktionen zur Verwaltung des Benutzeraccounts zur Verfügung, wie beispielsweise die Passwortänderung.

#### Schnittstellen

Der Login Microservice besitzt einen REST-Controller, wodurch er dem Web-Interface ermöglicht den Login durchzuführen. Dafür gibt es eine Schnittstelle zur Authentifizierung und eine um Nutzerdaten zu erhalten. Um auch die Möglichkeit zu bieten das Passwort zu ändern, existiert noch eine weitere Schnittstelle, die ein neues Passwort entgegen nimmt und das alte in der Datenbank ersetzt.

| **Schnittstelle** | **HTTP-Methode** | **benötigte Parameter** | **Rückgabe** |
|---|---|---|---|
| /app/rest/authenticate | GET | - | *String* userLogin |
| /app/rest/account | GET | - | *UserDTO* user |
| /app/rest/account/ change_password | POST | *String* password | void |

Tabelle 7.2: Schnittstellen des Login-Microservice

### 7.2.4 Fahrtenspeicherung

Die Fahrtenspeicherung ist der Microservice, welcher das Betreiber-Backend mit dem Tracking & BeOut System verbindet. Sie wird somit von dem Tracking & BeOut System angesteuert, welches Informationen einer abgeschlossenen Fahrt übergibt, um diese anschließend in der Datenbank zu persistieren. Es wird hierfür der Start- und Endzeitpunkt der Fahrt, die CiBoId des Nutzers, die Id der Start-/Endhaltestelle sowie die Id des Busses benötigt. Dieser Microservice erfüllt die funktionalen Anforderungen aus *F3 Fahrten speichern*.

#### Schnittstellen

Die Fahrtenspeicherung stellt eine RESTful-Schnittstelle zur Verfügung, die ein Fahrten-Data Transfer Object (DTO)[4] mit den zuvor genannten Informationen über die Fahrt benötigt, um diese in der Datenbank zu speichern. Diese Schnittstelle liefert das erstellte Objekt zurück.

| **Schnittstelle** | **HTTP-Methode** | **benötigte Parameter** | **Rückgabe** |
|---|---|---|---|
| /app/rest/savetrip | POST | *TripDTO* tripDTO (beinhaltet *String* startTimestamp *String* stopTimestamp *long* busId *String* userCiboId *long* startStopId *long* endStopId) | *Trip* trip |

Tabelle 7.3: Schnittstellen des Fahrtenspeicherung-Microservice

### 7.2.5 Fahrtensuche

Die Fahrtensuche ist aus den funktionalen Anforderungen *F4 Fahrtensuche* abgeleitet und ermöglicht es dem Betreiber alle getätigten Fahrten einzusehen. Hierbei bietet dieser Microservice die Funktion, Fahrten nach bestimmten Kriterien zu filtern. Fahrten

lassen sich nach Datum, Uhrzeit, Buslinie, Haltestelle oder CiBoId des Nutzers filtern. Die Haltestelle bezieht sich sowohl auf Fahrten, deren Starthaltestelle der angegebenen entspricht sowie auf Fahrten, deren Endhaltestelle mit der gegebenen übereinstimmt. Die Uhrzeit bezieht sich auf alle Fahrten, die während der angegebenen Zeit getätigt wurden. Das bedeutet also, dass die Fahrten angezeigt werden, bei denen der Startzeitpunkt kleiner oder gleich der angegeben Zeit ist und deren Endzeitpunkt größer oder gleich der angegeben Zeit ist. Die Fahrtensuche kommuniziert nur mit der Komponente des Web-Interfaces.

#### Schnittstellen

Bei der Fahrtensuche gibt es eine RESTful-Schnittstelle, die Datum, Uhrzeit, Name der Buslinie, CiBoID des Nutzers und HaltestellenId entgegen nimmt um aufgrund dieser Kriterien die Fahrten zu filtern. Allerdings sind hierbei alle Variablen optional.

| **Schnittstelle** | **HTTP-Methode** | **optionale Parameter** | **Rückgabe** |
|---|---|---|---|
| /app/rest/trips/filter | GET | *String* date<br>*String* time<br>*String* busName<br>*String* userCiboId<br>*List<Long>* stopIds | *List<Trip>* trips |

Tabelle 7.4: Schnittstellen des Fahrtensuche-Microservice

### 7.2.6 Preisberechnung

Der Microservice Preisberechnung (abgeleitet aus funktionalen Anforderungen *F5 Ticketpreis*) ist für die Ermittlung der Kosten einer getätigten Fahrt zuständig. Er enthält Informationen über das Tarifmodell und ermittelt somit den Preis einer Fahrt. Dafür ist wichtig, in welcher Tarifzone sich die Starthaltestelle und in welcher Tarifzone sich die Endhaltestelle befindet. Dadurch wird die Entfernung der beiden Haltestellen ermittelt und somit die Preisstufe erhalten. Die Kommunikation erfolgt mit dem Web-Interface, da hier der Preis dargestellt wird. Ansonsten ist dieser Microservice mit keiner weiteren Komponente verbunden.

#### Schnittstellen

Als RESTful-Schnitsttelle wird hier eine Schnittstelle mit Lesezugriff zur Verfügung gestellt, die die Tarifzone der Start- und Endhaltestelle benötigt, um den Preis der getätigten Fahrt zurück liefern zu können.

| Schnittstelle | HTTP-Methode | benötigte Parameter | Rückgabe |
|---|---|---|---|
| /app/rest/price | GET | *int* startZone<br>*int* endZone | *Double* price |

Tabelle 7.5: Schnittstellen des Preisberechnung-Microservice

### 7.2.7 Haltestellenverwaltung

Die Haltestellenverwaltung ist abgeleitet aus den funktionalen Anforderungen *F6 Haltestellen*. Außerdem ist sie notwendig um Abhängigkeit zwischen verschiedenen Microservices zu vermeiden. Er ist dafür zuständig Haltestelleninformationen aus der Datenbank zu übergeben, wenn diese mit einer Id angefordert wurden. Für neue Haltestellen kann mithilfe dieses Microservice ein neuer Datenbankeintrag angelegt werden. Die Haltestellenverwaltung ist notwendig, da somit die Komplexität des Projekts reduziert wird. Haltestellen enthalten in der Realität sehr umfangreiche Informationen, jedoch werden die wenigsten dieser Informationen innerhalb der Microservices benötigt, die mit Haltestellen arbeiten. Die Fahrtensuche beispielsweise benötigt nur den Namen der Start- und Endhaltestelle, um diese Informationen im Web-Interface anzeigen zu können. Die Preisberechnung benötigt nur Information über die Tarifzonen, in denen sich die beiden Haltestellen befinden. In beiden Microservices ein Haltestellenobjekt zu haben, in dem ein Eintrag sowohl für den Namen der Haltestelle, als auch für die Tarifzone in der sich die Haltestelle befindet, existiert, wäre unnötig. Beispielsweise benötigt die Fahrtensuche keine Tarifzonen und soll diese auch gar nicht enthalten, da sie zu der Preisberechnung entkoppelt sein soll. Somit gibt es einen Microservice, der alle Informationen über die Haltestellen verwaltet und diese bei Bedarf zurück gibt. Dafür bietet er Schnittstellen für das Web-Interface und ist nicht direkt mit den Microservices verbunden, die Haltestelleninformationen benötigen. Dies ist notwendig um eine Entkoppelung alle Microservices zu erhalten. In Unterabschnitt 7.4.5 wird noch aus der dynamischen Sicht verdeutlicht, wie die Haltestellenverwaltung funktioniert.

#### Schnittstellen

Die Haltestellenverwaltung stellt mehrere RESTful-Schnittstellen zur Verfügung um Haltestellen anzufordern. Es existiert eine Schnittstelle um alle Haltestellen aus der Datenbank zu erhalten, eine um mit Hilfe einer id ein Haltestellenobjekt zu erhalten und eine, die alle ids der Haltestellen zurückgibt, deren Haltestellenname mit dem angefragten übereinstimmt. Des Weiteren existiert eine RESTful-API, die es ermöglicht neue Haltestellen in der Datenbank zu speichern.

### 7.2.8 Web-Interface

Das Web-Interface ist eine sehr wichtige Komponente, da diese den kompletten Datenfluss der Anwendung kontrolliert. Somit ist diese Komponente nicht nur für die

| Schnittstelle | HTTP-Methode | benötigte Parameter | Rückgabe |
|---|---|---|---|
| /app/rest/stops | GET | - | *List<Stop>* stops |
| /app/rest/stop/id | GET | *long* stopId | *Stop* stop |
| /app/rest/stop/ stationName | GET | *String* stationName | *List<long>* stopIds |
| /app/rest/stops | POST | *StopDTO* stopDTO (beinhaltet *String* stationName *int* fareZone) | void |

Tabelle 7.6: Schnittstellen des Haltestellen-Microservice

Visualisierung der Daten zuständig sondern auch für die Handhabung von Daten. Das Web-Interface ist die Kontrolleinheit, die die Microservices steuert und somit miteinander verknüpft. Wie in Kapitel 5 deutlich wurde, ist das Kriterium, welches Microservices ausmacht, die Unabhängigkeit von Microservices zu einander. Dies wird allein durch das Web-Interface ermöglicht. Die Architektur (vgl. Abbildung 7.4) der Komponente des Web-Interface ist von dem Entwurfsmuster Model View Controller [7] hergeleitet. Der Grund für die Wahl dieses Entwurfsmusters ist, dass die für das Web-Interface verwendete Technologie AngularJS (siehe Unterabschnitt 8.1.2), auf Model View Controller basiert. Dadurch enthält die Architektur der Komponente folgende drei Schichten:

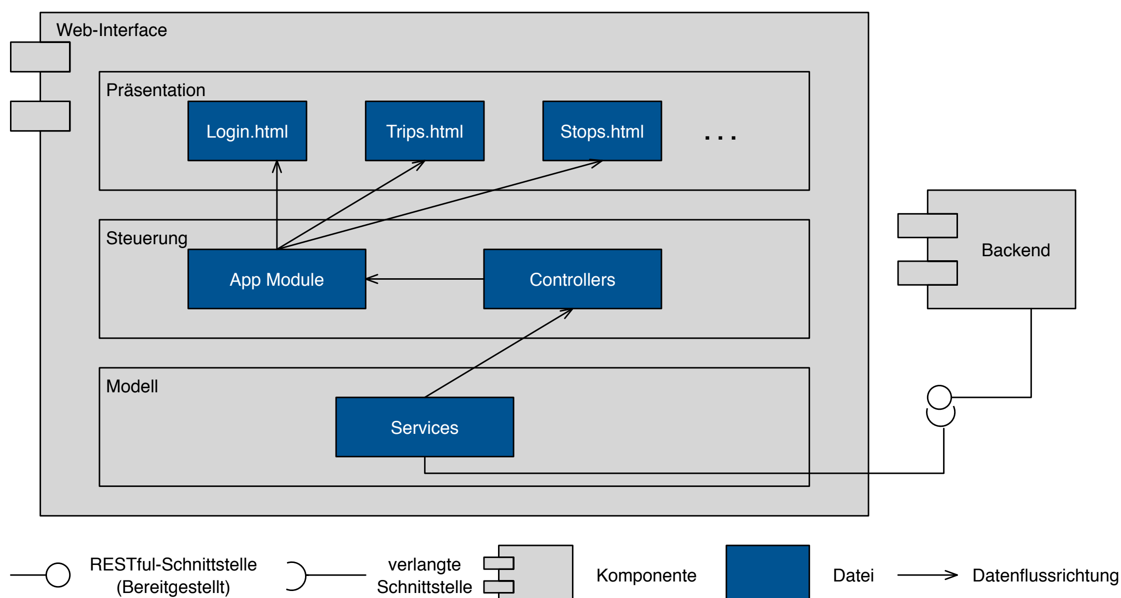


Abbildung 7.4: Komponentendiagramm der Web-Interface Komponente

**Modell** Die Modell-Schicht ist für den Datentransfer und die Datenverarbeitung zuständig. Die JavaScript Services kommunizieren mit den RESTful-Schnittstellen der Microservices und sind somit für die externe Kommunikation zuständig. Dadurch spiegeln sich in der Modell-Schicht die einzelnen Microservices in den JavaScript Services wieder.

**Steuerung** Die Steuerungs-Schicht ist für die Datenverarbeitung zuständig. Die Controller, greifen auf die Services zu, um die externen Daten zu erhalten. Außerdem reagieren sie auf das Verhalten auf den Hypertext Markup Language (HTML)-Seiten und geben je nach Anfrage die gewünschten Daten zurück. Das App Module ist der Zusammenhalt von Controllern und HTML-Seiten. Hier wird definiert, welcher Controller für welche Seite zuständig ist.

**Präsentation** Die Präsentations-Schicht ist für die Darstellung zuständig und beinhaltet somit die HTML-Seiten. Von hier aus werden Funktionen des Controllers aufgerufen um auf die Eingaben des Nutzers einzugehen. An dieser Stelle befinden sich Schnittstellen zwischen Nutzer und der gesamten Anwendung.

## 7.3 Entitäten

### 7.3.1 Fahrten Datenmodell

Das fachliche Datenmodell, welches aus den funktionalen Anforderungen abgleitet wurde, ist in der Abbildung 7.5 zusehen. Im folgenden erfolgt eine genauere Betrachtung des ER-Diagramms, allerdings werden dabei nicht die Bedeutungen der einzelnen Attribute erläutert, hierzu soll die Tabelle 7.7 dienen.
Im Zentrum des ER-Modells stehen die Fahrten (Trips), denn in der Anwendung geht es darum Fahrten zu speichern und anzuzeigen (vgl. Kapitel 6 *Anforderungen*). Des Weiteren gibt es die Entitäten Benutzer (User), Buslinie (BusLine), Haltestelle (Stops) und Preisstufe (PriceLevel). Ein Benutzer kann beliebig viele Fahrten durchführen, eine Fahrt ist aber nur mit einem Benutzer in Verbindung zu bringen. Auf einer Fahrt wird genau eine Buslinie benutzt, Buslinien können aber natürlich Teil beliebig vieler Fahrten seien. Jede Fahrt besitzt genau zwei Haltestellen und zwar eine Starthaltestelle an der der Nutzer einsteigt und eine Endhaltestelle, an der der Nutzer aussteigt. Haltestellen können Teil beliebig vieler Fahrten sein. Abschließend hat jede Fahrt noch eine Preisstufe, die die Kosten der Fahrt darstellt (abhängig von der Entfernung die zurück gelegt wurde). Eine Preisstufe kann in beliebig vielen Fahrten verwendet werden.

### 7.3.2 Login Datenmodell

Um den Login in das Web-Interface für den Betreiber zu ermöglichen, wird allerdings noch ein weiteres ER-Modell benötigt (vgl. Abbildung 7.6). Bei diesem Datenmodell handelt es sich um ein nicht fachliches Modell, da es unabhängig von dem CiBo

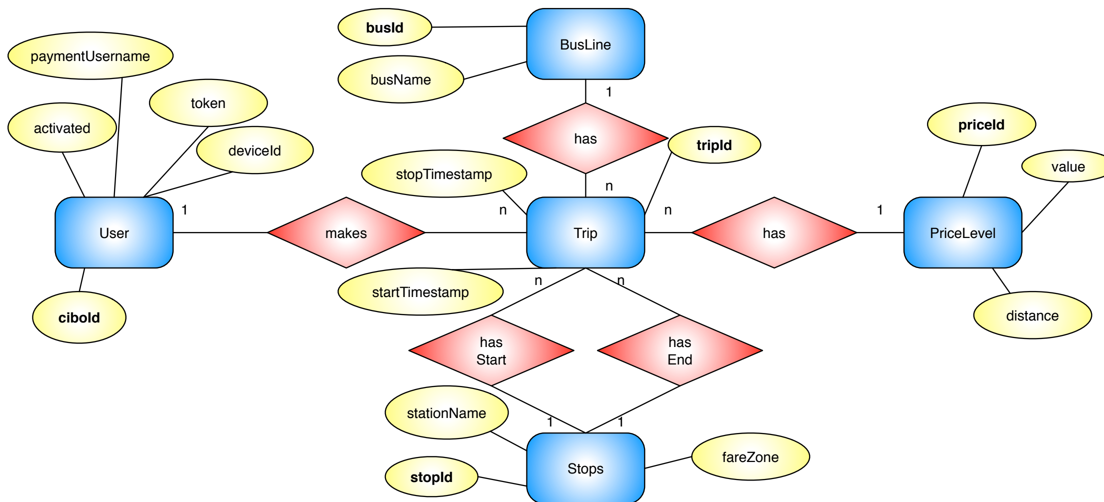


Abbildung 7.5: Übersich über das Fahrten ER-Modell

Projekt ist. Es beinhaltet die Benutzer, die sich in das Web-Interface des CiBo Informationssystems einloggen können. Zwar ist dies zunächst nur auf den Betreiber ausgelegt, allerdings besteht die Möglichkeit, dass verschiedene Nutzer sich einloggen können. Für den Login besitzt ein Nutzer einen Nutzernamen (login), der ihn eindeutig identifiziert und ein Passwort. Des Weiteren können noch Vor- und Nachname des Nutzers gespeichert werden. Ein Nutzer kann beliebig viele PersitentToken haben, die es ermöglichen, dass ein Nutzer automatisch eingeloggt wird. Außerdem hat ein Nutzer noch mindestens eine Autorität. Er kann entweder normaler Nutzer sein, also eingeschränkten Zugriff erhalten oder Admin sein mit vollem Zugriff auf das System erhalten. Es wird davon ausgegangen, dass der Betreiber auch der Admin ist. Die Funktionalität des eingeschränkten Nutzer bietet nur die Möglichkeit, das System so zu erweitern, dass auch Nutzer ihre getätigten Fahrten einsehen können.

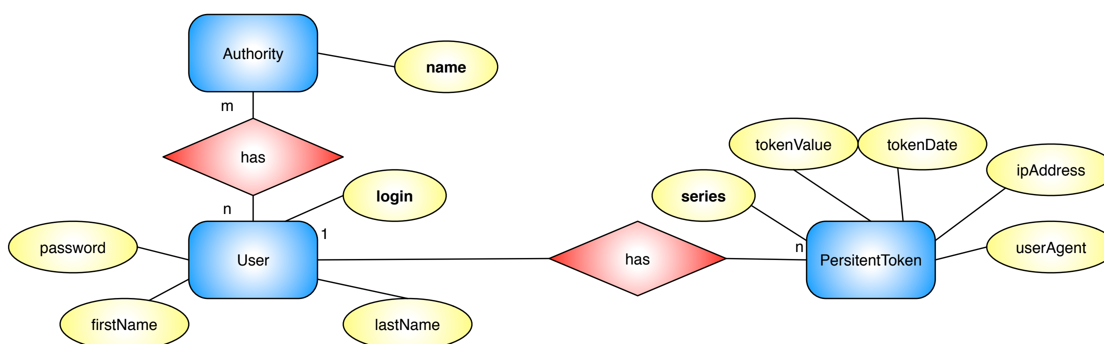


Abbildung 7.6: Übersich über das Login ER-Modell

| Variable | Typ | Beschreibung |
|---|---|---|
| ***User Object*** | | |
| *ciboId* | int | Eindeutige id, die dem User zugeordnet wird |
| *deviceId* | String | Id des verwendeten Geräts des Nutzers |
| *token* | int | Vom Payment-Anbieter generiertes Token |
| *paymentUsername* | String | Nutzername des verwendeten Payment-Accounts |
| *activated* | boolean | Gibt an ob der Account aktiviert ist oder nicht |
| ***Trip Object*** | | |
| *tripId* | long | Id um die Fahrt eindeutig zu identifizieren |
| *startTimestamp* | Timestamp | Datum und Uhrzeit des Beginns der Fahrt |
| *stopTimestamp* | Timestamp | Datum und Uhrzeit des Endes der Fahrt |
| *busLine* | BusLine | Buslinie, die für Fahrt verwendet wurde |
| *userCiboId* | String | ciboId des Nutzers, der die Fahrt getätigt hat |
| *startStopId* | long | id der Start Haltestelle |
| *endStopId* | long | id der End Haltestelle |
| ***BusLine Object*** | | |
| *busId* | long | Id um die Buslinie eindeutig zu identifizieren |
| *busName* | String | Name der Buslinie |
| ***Stop Object*** | | |
| *stopId* | long | Id um die Haltestelle eindeutig zu identifizieren |
| *stationName* | double | Name der Haltestelle |
| *fareZone* | int | Tarifzone, in der sich die Haltestelle befindet |
| ***Price Object*** | | |
| *priceId* | String | Id um den Preis eindeutig zu identifizieren |
| *value* | double | Betrag des Preises in € |
| *distance* | int | Entfernung der Tarifzonen für diese Preislevel |

Tabelle 7.7: Übersicht über die Objekte des Datenmodells

## 7.4 Dynamische Sicht

Zuvor wurden die statischen Aspekte der Anwendung betrachtet, im Folgenden wird nun auf die dynamische Sicht eingegangen. Dazu werden die Vorgänge der Registrierung, des Logins, der Passwortänderung, der Fahrtenspeicherung, der Darstellung der Fahrten im Web-Interface und der Ersterstellung neuer Haltestellen, genauer betrachtet. Eine Veranschaulichung dieser Vorgänge erfolgt durch UML-Sequenzdiagramme. B. Rumpe [38] beschreibt Sequenzdiagramme als Darstellung von Interaktionen zwischen Objekten. In diesem Fall handelt es sich allerdings um Interaktionen zwischen Komponenten. Die Kommunikation zwischen den Komponenten stellt hier, anders als von Rumpe erläutert, nicht immer konkrete Methodenaufrufe dar.

### 7.4.1 Registrierung

Wie in Abbildung 7.7 zu sehen, geht die Registrierung von der CiBo App aus, wenn der Nutzer die Zahlungsmethode des Payment Anbieters wählt. Die App übergibt dann dem Registrierungs-Microservice den Payment Usernamen und die GeräteId des verwendeten Android Geräts, woraufhin das Backend des Payment Anbieters interagiert, um zu dem gegebenen Nutzer ein Token generieren zu lassen. Dieses Token wird dann an die CiBo App weitergeleitet. Im späteren Verlauf logged der Nutzer sich in das Payment System ein, um seinen Account zu verifizieren. Sind die Angaben (Username, Token, Passwort) des Nutzers korrekt, so aktiviert der Payment Anbieter den Account des Nutzers. Falls irgendetwas mit dem Account des Nutzers nicht stimmen sollte, kann der Payment Anbieter jederzeit den Account deaktivieren, um zu verhindern, dass der Nutzer weiterhin Fahrten durchführen kann, ohne das die Zahlung gewährleistet ist.

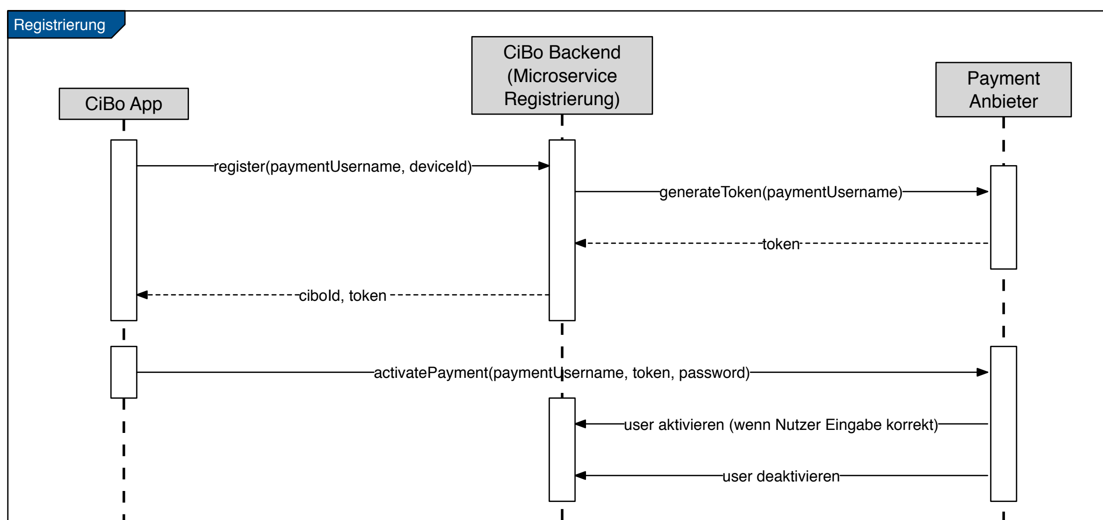


Abbildung 7.7: Sequenzdiagramm Registrierung eines Nutzers

### 7.4.2 Login

Um Zugriff auf die Webseite zu bekommen und somit Fahrten einzusehen, muss man sich zunächst einloggen. Eine Übersicht über diesen Ablauf bietet das Sequenzdiagramm in Abbildung 7.8. Zunächst wird überprüft, ob die aktuelle Session bereits authentifiziert ist, also ob der Nutzer bereits eingeloggt ist. Ist dies der Fall, so erhält er direkt Zugriff auf die Seite. Ist dies nicht der Fall, muss die Person den Benutzernamen und das Passwort über das Web-Interface eingeben. Anschließend wird eine Anfrage an den Login Microservice gestartet, um die Daten zu dem Account zu erhalten. Der login Microservice gibt den Nutzer Account aus der Datenbank zurück. Daraufhin verifiziert die Web-Applikation, ob Username und Passwort mit den Daten aus der Datenbank übereinstimmen. Ist dies der Fall, so gilt der Nutzer als eingeloggt.

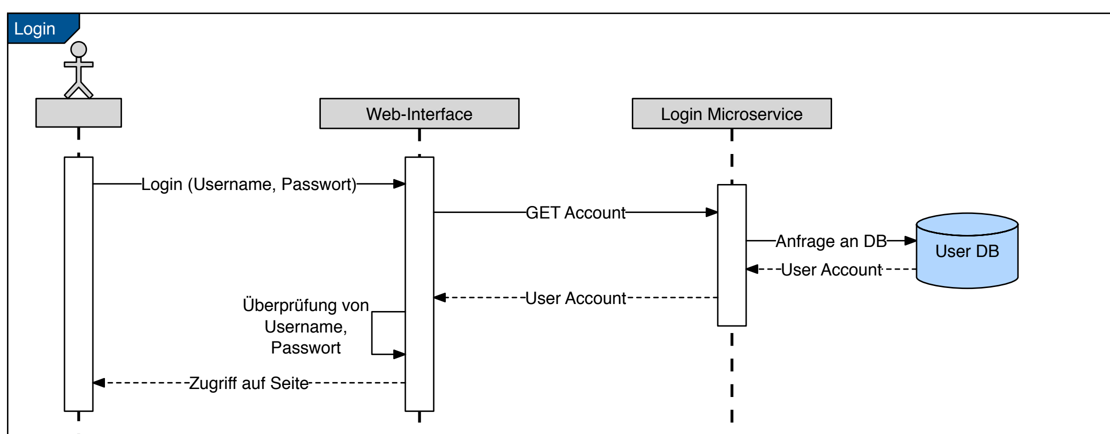


Abbildung 7.8: Sequenzdiagramm einloggen in das Web-Interface

### 7.4.3 Passwort ändern

Ist ein Nutzer erstmal in dem Web-Interface eingeloggt, kann dieser dort auch sein Passwort ändern (vgl. Abbildung 7.9). Dazu gibt er über das Web-Interface das gewünschte neue Passwort ein und bestätigt dieses um versehentlich falsche Eingaben vorzubeugen. Darauf hin überprüft die Web-Komponente, ob die beiden eigegebenen Passwörter identisch sind und überträgt das neue Passwort an den Login Microservices, mit der Aufforderung dieses zu speichern. Der Login Microservice ersetzt dann in der Datenbank das alte Passwort mit dem neuen.

### 7.4.4 Fahrtenspeicherung

Abgeschlossene Fahrten werden vom Tracking & BeOut System erkannt. Daher ist es auch diese Komponente, die die Fahrtenspeicherung anstößt. In Abbildung 7.10 ist dieser Vorgang grafisch mittels eines Sequenzdiagramms dargestellt. Das Tracking

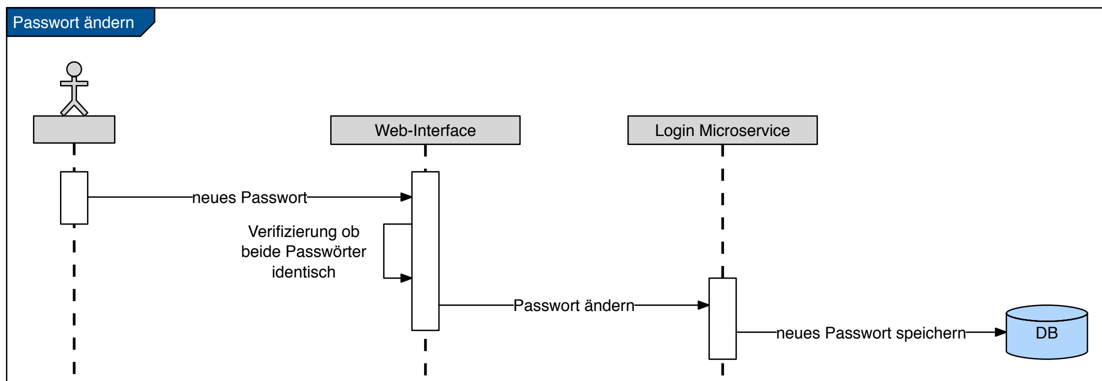


Abbildung 7.9: Sequenzdiagramm Änderung des Passworts

& BeOut System übermittelt die Fahrdaten an den Microservice Fahrtenspeicherung. Hierzu werden CiBoId des Nutzers, Start- & Endzeitpunkt, Id der Buslinie und Ids der Start- & Endhaltestellen benötigt. Mit Hilfe dieser Informationen erstellt der Fahrtenspeicherungs-Microservice ein neues Fahrten Objekt und persistiert dieses anschließend in der Datenbank. Das Fahrten Objekt wird anschließend zurück gegeben.

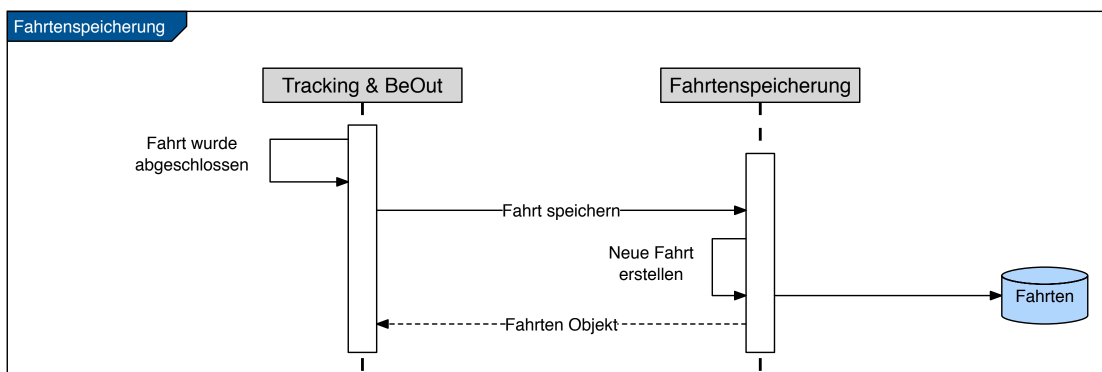


Abbildung 7.10: Sequenzdiagramm Speicherung von Fahrten

### 7.4.5 Darstellung der Fahrten inklusive Preis

Um eine Fahrt im Web-Interface dazustellen, werden die drei Microservices Fahrtensuche, Haltestellenverwaltung und Preisberechnung benötigt. Das Sequenzdiagramm in Abbildung 7.11 bietet eine Übersicht über den dynamischen Ablauf. Zunächst fragt das Web-Interface die Fahrten (ggf. gefiltert nach Haltestelle, CiBoId, Haltestellen etc.) vom Fahrtensuche Microservice an. Um anschließend alle Fahrt mit vollständigen Informationen visuell darstellen zu können, werden folgende Schritte mit jeder Fahrt einzeln durchgeführt (siehe loop im Schaubild):

1. Mit den Haltestellen Ids der Start- und Endhaltestelle (die von der Fahrtensuche übergeben wurden), werden über die Haltestellenverwaltung die Namen und Tarifzonen der Start- und Endhaltestellen ermittelt.
2. Die beiden Tarifzonen werden vom Web-Interface an die Preisberechnung übergeben, welche anschließend den Preis ermittelt und zurückgibt.
3. Schließlich kann die Fahrt über das Web-Interface inklusive der Namen der Start- und Endhaltestelle sowie die Kosten der Fahrt dargestellt werden.

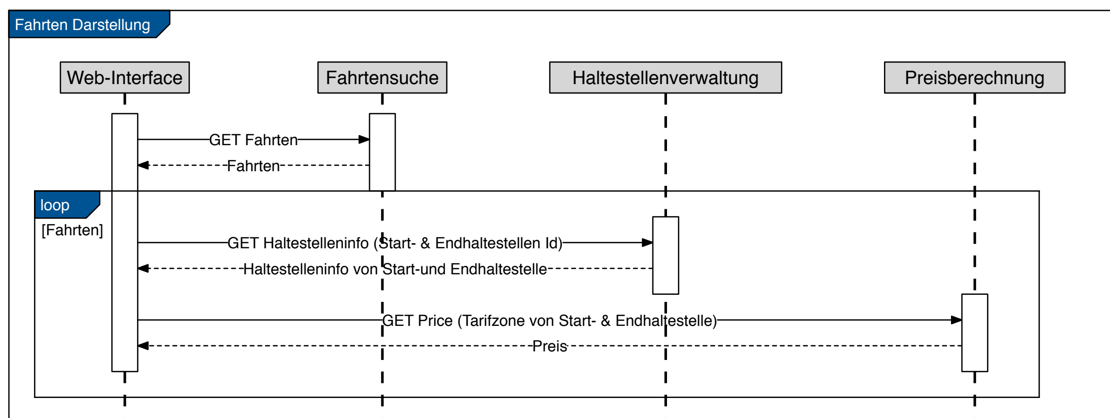


Abbildung 7.11: Sequenzdiagramm Darstellung der Fahrten im Web-Interface

### 7.4.6 Erstellung neuer Haltestellen

Der Ablauf der Erstellung neuer Haltestellen, ähnelt sehr der Fahrtenspeicherung. Daher wurde hier auf das Sequenzdiagramm verzichtet, welches den Ablauf grafisch darstellt.
Über das Web-Interface können alle Haltestellen eingesehen werden. Hierzu wird eine Anfrage an die Haltestellenverwaltung gesendet, woraufhin alle Haltestellen aus der Datenbank zurück gegeben werden. Des Weiteren können auch neue Haltestellen hinzugefügt werden. Dazu wird der Name der neuen Haltestelle und deren Tarifzone über das Web-Interface eingegeben, welches daraufhin die beiden Informationen an die Haltestellenverwaltung übermittelt. Die Haltestellenverwaltung erstellt ein neues Haltestellenobjekt und speichert dieses in der Datenbank. Werden daraufhin erneut alle Haltestellen angefordert, so lässt sich die neue Haltestelle in der Haltestellenübersicht des Web-Interfaces finden.

## 7.5 Zusammenfassung

In diesem Kapitel wurde der Entwurf des Betreiber-Backendsystems beschrieben. Dazu erfolgte eine Einteilung der funktionalen Anforderungen in Microservices, welche sich

architektonisch ähneln. Diese Microservices kommunizieren über RESTful-Schnittstellen mit der Web-Komponente, die als Vermittler agiert und die Daten mittels eines Web-Interfaces zur Verfügung stellt. Außerdem erfolgte eine Beschreibung der beiden Datenmodelle, die für das Betreiber-Backend nötig sind. Es wurde das fachliche Datenmodell erläutert, welches aus den funktionalen Anforderungen entstand sowie das nicht fachliche Datenmodell, welches für die Authentifizierung im Web-Interface notwendig ist. Am Ende dieses Kapitels erfolgte eine dynamische Sicht auf die Funktionalitäten des Betreiber-Backends.
Auf diesem Architekturentwurf basiert die Implementierung der Microservices und der Web-Komponente, die das nächste Kapitel detaillierter beschreibt.

# 8 Implementierung

Inhalt


8.1 Verwendete Technologien . . . 57
8.2 Microservices . . . 59
8.3 Web-Frontend . . . 64
8.4 Gesamtübersicht . . . 64
8.5 Testen . . . 65
8.6 Zusammenfassung . . . 67


In diesem Kapitel wird die Implementierung des Softwareentwurfs aus dem vorangegangenen Kapitel dargestellt. Es erfolgt eine Erläuterung der, im Rahmen dieser Arbeit, verwendeten Technologien. Außerdem wird auf die Implementierung der einzelnen Komponenten eingegangen, sowie das Betreiber-Backend als gesamte Anwendung betrachtet. Das Kapitel schließt mit der Vorgehensweise beim Testen ab.

## 8.1 Verwendete Technologien

Für die Realisierung des Betreiber-Backends wurden verschiedene Technologien verwendet.

### 8.1.1 Spring & Spring Boot

Spring ist ein Framework für Java, welches eine Alternative zu Java EE für enterprise Projekte bietet. Dazu stellt dieses Framework ein sehr breites Spektrum an Komponenten zur Verfügung. Java EE Konzepte wie EJB, und die API werden nicht über einen Container bereitgestellt sondern über Komponenten innerhalb des Frameworks. Dabei unterstützt Spring alle wesentlichen Java EE Standard APIs wie JAX-WS (Webservices), JPA (Persistence). Zur Laufzeit benötigt Spring daher keinen kompletten Application Server sondern es reicht ein einfacherer Servlet Web-Container wie Tomcat. Mit der Erweiterung Spring Boot wurde diese Abhängigkeit aufgelöst, indem durch Spring Boot ein solcher Container direkt in Spring eingebettet wurde. Diese kompakte Architektur unterstützt die Entwicklung von Mikroservices. Daher wurde diese Komponententechnologie für die Untersuchung genutzt.
Spring mit Spring Boot ist demnach das erste Framework, welches Software auf Basis von Microservices unterstützt.

### 8.1.2 AngularJS

AngularJS [11] ist ein Framework von Google, welches für die Erstellung von Webanwendungen entwickelt wurde. Durch dieses Framework können mit JavaScript und HTML single-page-Webanwendungen erstellt werden. AngularJS basiert auf dem Model-View-Controller-Entwurfsmuster [38], wobei das Model bidirektional mit der View verbunden ist. Das bedeutet, dass Eingabefelder so mit dem Model verbunden sind, dass eine Eingabe sofort als Ausgabe an einer anderen Stelle sichtbar ist. Innerhalb des HTML-Codes müssen keine JavaScript Tags verwendet werden, sondern Funktionen des Controllers können direkt im HTML-Code aufgerufen werden. Dies wird durch Direktive ermöglicht, die an dem Präfix `ng` zu erkennen sind. Beispielsweise ist `ng-repeat` als Schleife vordefiniert oder `ng-click="onClick()"` führt die Funktion `onClick()` des Controllers aus.

### 8.1.3 Swagger

Swagger [41] ist ein Web-Anwendungs-Framework, welches RESTful APIs visuell darstellt. Somit können diese Schnittstellen leicht getestet werden. Dazu analysiert Swagger die RESTful-APIs der Anwendung und stellt diese über ein Web-Interface bereit, in dem alle GET, POST, DELETE, PUT und PATCH Methoden aufgelistet werden. Für jede Schnittstelle kann man eine passende Anfrage eingeben und somit testen, wie sich die Schnittstelle verhält, also welche Antwort zurück gesendet wird.
Ein Beispiel eines Swagger Tests zeigt Abbildung 8.1. Hier wird als Parameter eine Id erwartet, die man in ein Textfeld eingeben kann. Daraus wird dann ein Anfrage Uniform Resource Locator (URL) (*Request URL*) erstellt auf dessen Basis die Antwort (*Response Body*) erzeugt wird. Man kann also mit Swagger überprüfen ob bei einer bestimmten Anfrage, der Response Body wie gewünscht aussieht.

### 8.1.4 JHipster

JHipster [18] ist ein quelloffenes Projekt auf Basis von Yeoman [53]. Es fungiert als Scaffolder, da es Grundgerüste für Web-Anwendungen generiert. Auf Basis eines solchen Grundgerüst, kann dieses abgeändert werden, um somit eine eigene Web-Anwendung zu erstellen. Dabei kann für die Generierung der Anwendung zwischen verschiedenen Technologien gewählt werden. Beispielsweise welche Java Version (7 oder 8) oder welche Datenbank verwendet werden soll (SQL oder MongoDB). JHipster besteht aus einer Client und einer Server Seite [18].

**Client** Die Client Seite basiert auf AngularJS. Sie ist für die Schnittstelle zwischen Programm und Nutzer zuständig. Dabei kann sie Dienste vom Server anfordern.

**Server** Die Serverseite basiert auf Spring Boot. Hier werden die Dienste bereitgestellt, die der Client nutzen kann.

**Parameters**

| Parameter | Value | Description | Parameter Type | Data Type |
|---|---|---|---|---|
| stopId | 1 | stopId | query | integer |

**Response Messages**

| HTTP Status Code | Reason | Response Model |
|---|---|---|
| 200 | OK | Model Model Schema |
| 403 | Forbidden | |
| 401 | Unauthorized | |
| 404 | Not Found | |

```
{
  "stationName": "",
  "fareZone": 0,
  "stopId": 0
}
```

Try it out! Hide Response

**Request URL**

```
http://localhost:8086/app/rest/stop/id?stopId=1
```

**Response Body**

```
{
  "stopId": 1,
  "stationName": "westbhf",
  "fareZone": 1
}
```

Abbildung 8.1: Screenshot eines Swagger Tests

## 8.2 Microservices

Im Folgenden wird nun die Implementierung der Microservices erläutert. Sie beruhen alle auf der gleichen Architektur und ähneln sich demnach sehr in der Implementierung. Bei allen Microservices handelt es sich um Java Projekte, die mit Hilfe von RESTful-Schnittstellen nach außen kommunizieren. Am Beispiel des Registrierungs-Microservice wird ein Klassendiagramm eines Microservices dargestellt. JHipster wurde verwendet um eine prototypische Webanwendung zu generieren, auf deren Basis dann diese zu einem vollständigen Microservice erweitert wurde. Alle weiteren Microservices orientierten sich an dem Design des Login Service und erhielten die gleiche Infrastruktur wie beispielsweise Liquibase [21] für die Datenbankverwaltung oder Swagger zum Testen der REST-Controller. Im Folgenden wird auf die Umsetzung der Entitäten und auf die Implementierung der einzelnen Microservices eingegangen.

### 8.2.1 Umsetzung Entitäten

Um die in Abschnitt 7.3 beschriebenen Entitäten als Datenmodell umzusetzen, wurde ein relationales Datenmodell auf SQL Basis gewählt. Allerdings hätte man hierbei auch ein

nicht realtionales Datenmodell wie noSQL wählen können. Bei den Datenbanktabellen wurde auf eine referentielle Integrität verzichtet, die Datenbanktabellen sind somit nicht direkt mit einander verbunden. Der Grund dafür ist, dass jeder Microservice nur mit den Entitäten verbunden sein sollen, die auch für seine Funktionalitäten wichtig sind. Eine Fahrt beispielsweise beinhaltet nur die ciboId des Nutzer, ist aber nicht mit dem Nutzer Objekt des Registrierungs-Microservice verbunden, damit keine Kopplung erzeugt wird. Diese indirekte Verbundenheit zwischen den Datenbanktabellen ermöglicht eine Entkoppelung der Microservices.

### 8.2.2 Registrierung

Wie in Kapitel 7 bereits erläutert wurde, greift der Registrierung-Microservice auf den Payment Anbieter zu. Dazu wurde eine Java Projekt erstellt, welches den Payment-Anbieter simuliert. Dort existiert eine RESTful-Schnittstelle, die einen Benutzernamen entgegen nimmt, einen Token generiert und diesen zurück gibt.
Die Registrierung verwendet aus dem in Abschnitt 7.3 beschriebenen Datenmodell nur die User-Entität, da hier neue Nutzer in der Datenbank gespeichert werden müssen. Ein Klassendiagramm des Registrierungs-Microservice ist in Abbildung 8.2 zu sehen, es basiert auf der Architektur die in Unterabschnitt 7.2.1 vorgestellt wurde. Die User Klasse, erzeugt Nutzer Objekte, die alle wichtigen Informationen eines Nutzers enthalten und stellt somit die User Entität dar. HttpURLConnection greift auf die RESTful-API des Payment Anbieters zu und UserResource stellt die Verbindung zur Datenbank zur Verfügung. Die RESTful-APIs, die zur Registrierung des Nutzer benötigt werden, sind in der UserResource Klasse enthalten.
Da wie bereits erwähnt, sich die Microservices ähneln wird bei den anderen Microservices auf Klassendiagramme verzichtet.

### 8.2.3 Login

Der Scaffolder generiert eine Standardimplementierung für eine Loginkomponente. Daher bestand die Implementierung des Login-Microservices im Wesentlichen darin das vom Scaffolder generierte Projekt um alle nicht benötigten Klassen und Methoden anzupassen, bis schließlich nur noch der Login und die Webanwendung übrig blieb.

### 8.2.4 Fahrten speichern

Der Fahrtenspeicherung-Microservice verwendet die folgenden Entitäten des *Fahrten Datenmodell*. Es existiert eine Trip- und ein BusLine-Entity-Klasse mit den entsprechenden Attributen. Damit das Tracking & BeOut System Fahrten speichern kann wurde eine REST-Schnittstelle implementiert. Der Code dazu ist in Quelltext 8.1 zu finden. Es wird ein DTO [4] entgegen genommen (vgl. Codezeile 9), um mit den enthaltenen Informationen eine Trip-Entität zu erzeugen (Codezeile 18-31). Außerdem wird noch die Abhängigkeit zur BusLine-Entität hergestellt (Codezeile 13). Die erstellte Trip-Entität wird in der Datenbank gespeichert (Codezeile 31) und anschließend zurück

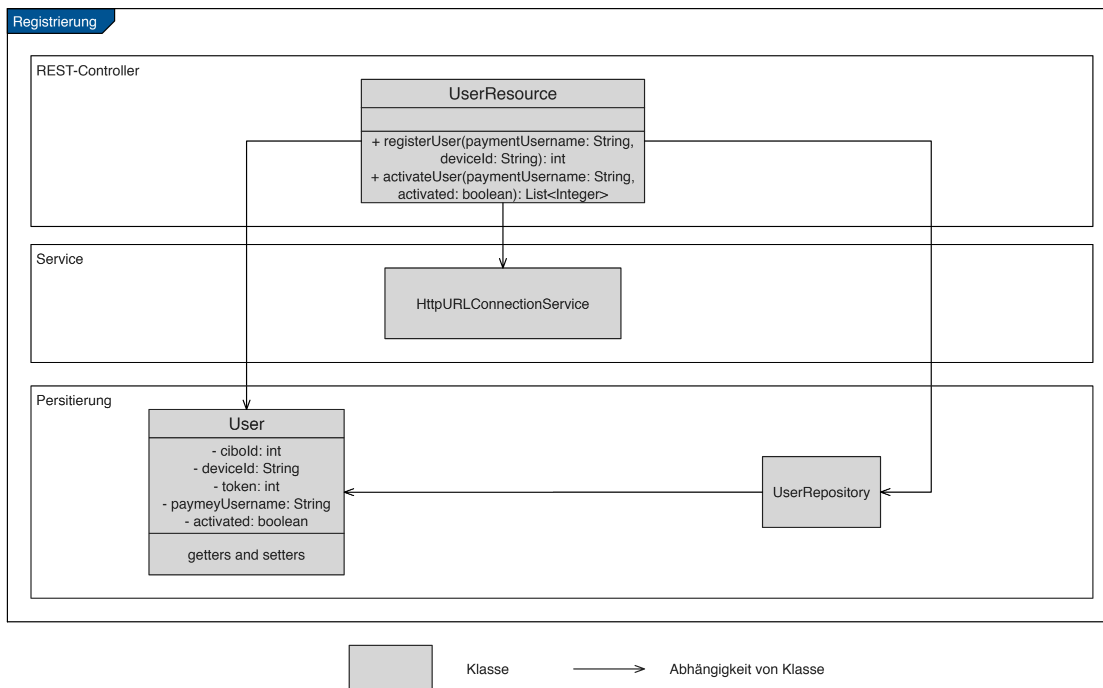


Abbildung 8.2: Klassendiagramm Registrierung

gegeben (Codezeile 36). Wie ein solches TripDTO aussieht ist in Quelltext 8.2 dargestellt. Dies ist ein Beispiel eines JSON-Objekts, welches mittels erfolgreichen POSTs an die RESTful-Schnittstelle übergeben wird. Für einen erfolgreichen POST müssen alle Variablen belegt sein.

```
1  /**
2  * POST /rest/savetrip -> create a new trip
3  */
4  @RequestMapping(value = "/rest/savetrip",
5    method = RequestMethod.POST,
6        produces = MediaType.APPLICATION_JSON_VALUE)
7  @Timed
8  public ResponseEntity<Trip> createNewTrip(
9        @RequestBody TripDTO tripDTO, HttpServletRequest request,
10       HttpServletResponse response) {
11
12   //get BusLine Object from the database
13   BusLine busLine = busLineRepository.findOne(tripDTO.getBusId());
14   if(busLine == null){
15     return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
16   }
17
18   Trip newTrip = new Trip();
```

```
19
20   Timestamp startTimestamp =
21       Timestamp.valueOf(tripDTO.getStartTimestamp());
22   Timestamp stopTimestamp =
23       Timestamp.valueOf(tripDTO.getStopTimestamp());
24
25   //set trip information
26   newTrip.setStartTimestamp(startTimestamp);
27   newTrip.setStopTimestamp(stopTimestamp);
28   newTrip.setBusLine(busLine);
29   newTrip.setUserCiboId(tripDTO.getUserCiboId());
30   newTrip.setStartStopId(tripDTO.getStartStopId());
31   newTrip.setEndStopId(tripDTO.getEndStopId());
32
33   //save new trip in database
34   tripRepository.save(newTrip);
35
36   return new ResponseEntity<Trip>(newTrip, HttpStatus.CREATED);
37 }
```

Quelltext 8.1: Methode des REST-Controllers TripResource.java zum Speichern von Fahrten

```
1 {
2   "startTimestamp": "2014-12-21 14:33:12",
3   "stopTimestamp": "2014-12-21 14:37:31",
4   "userCiboId": "user",
5   "startStop": "audimax",
6   "endStop": "ponttor",
7   "busId": "1"
8 }
```

Quelltext 8.2: JSON Bespiel eines POST

### 8.2.5 Fahrtensuche

Der Microservice Fahrtensuche benutzt die gleiche Datenbank wie die Fahrtenspeicherung, um Zugriff auf die gespeicherten Fahrten zu erhalten. Dadurch existiert auch hier ein Trip und ein BusLine Entity. Die RESTful-Schnittstelle liefert Fahrten zurück, die nach Bedarf gefiltert werden können. Es kann nach Datum, Uhrzeit, Buslinie, CiBoId des Nutzers oder Haltestelle gefiltert werden, jedoch sind diese Variablen alle optional. Ein Beispiel einer Rückgabe nachdem Fahrten mit der Buslinie 3A gesucht wurden, ist in Quelltext 8.3 zu sehen. Die Anfrage-URL befindet sich in Codezeile 1.

```
1 http://localhost:8083/app/rest/trips/filter?busName=3A
```

```
2
3 [
4   {
5     "tripId": 1,
6     "startTimestamp": 1417433400000,
7     "stopTimestamp": 1417433635000,
8     "busLine": {
9       "busId": 1,
10      "busName": "3A"
11    },
12    "userCiboId": "user",
13    "startStopId": 2,
14    "endStopId": 1
15  }
16 ]
```

Quelltext 8.3: JSON Beispiel eines POST inklusive Anfrage-URL

### 8.2.6 Ticketpreis

Die Ticketpreisberechnung besitzt nur das Entity PriceLevel. Dazu wurde ein Tarifmodell entwickelt um einen Preis ermitteln zu können. Jede Haltestelle hat einen integer Wert, der die Tarifzone angibt. Zur Preisberechnung wird der RESTful-Schnittstelle die Tarifzone der Start- und Endhaltestelle übergeben, um anschließend die Differenz dieser beiden Werte zu berechnen. Es wurde angenommen, dass alle Tarifzonen nebeneinander liegen und somit die Differenz ein realistischer Wert für die Entfernung ist. Eine Übersicht der verschiedenen Preisstufen inklusive Entfernung und Kosten ist in Tabelle 8.1 zu finden (Angelehnt an das Tarifmodell der ASEAG[1]).

| Preisstufe | Entfernung | Preis | Beschreibung |
|---|---|---|---|
| *1* | 0 | 1,50 € | In der selben Tarifzone |
| *2* | 1 | 2,65 € | In der Benachbarten Tarifzone |
| *3* | 2 | 3,55 € | Zwei Tarifzonen entfernt |
| *4* | 3 | 5,30 € | Drei Tarifzonen entfernt |
| *5* | 4 | 8,30 € | Vier Tarifzonen entfernt |

Tabelle 8.1: Übersicht über das Tarifmodell

[1] http://www.aseag.de/Tickets_und_Tarife/Tarife/Tarif-Tabelle_ab_Januar_2015.html

### 8.2.7 Haltestellenverwaltung

Der Microservice Haltestellenverwaltung beinhaltet nur die Haltestellen (Stop) Entität, da dieser nur für das Bereitstellen von Halteselleninformationen und das Speichern neuer Haltestellen in der Datenbank zuständig ist. So gibt es RESTful-Schnittstellen um alle Haltestellen, Haltestelle mit einer bestimmten Id oder aller Haltestellen-Ids , welche einen bestimmten Haltestellennamen haben zu übergeben. Ein Beispiel einer Rückgabe mit der Anfrage nach der Haltestelle mit der id=2 ist in Quelltext 8.4 zusehen. Die dazu gehörige Anfrage-URL ist in Codezeile 1 zu finden.

```
http://localhost:8086/app/rest/stop/id?stopId=2

{
  "stopId": 2,
  "stationName": "audimax",
  "fareZone": 1
}
```

Quelltext 8.4: JSON Beispiel eines Response Body inklusive Anfrage-URL

## 8.3 Web-Frontend

Für das Web-Frontend wurde auch JHipster als Basis verwendet. So bestand bereits die Infrastruktur und die ersten HTML-Seiten wie Login und Passwortänderung. Die Entwicklung der JavaScript Services und Controller orientiert sich an den durch JHipster bereits bestehenden. Die JavaScript Services greifen dann auf die verschiedenen RESTful-APIs der Microservices zu. Die somit erhaltenen Daten werden nach Bedarf in den JavaScript Controllern weiter verarbeitet, um sie anschließend über die HTML Seiten visuell dargestellt zu werden. Screenshots des Web-Interface sind in Abschnitt A.1 des Anhangs zu finden.

## 8.4 Gesamtübersicht

Alle Microservices wurden so konfiguriert, dass sie auf verschiedenen Ports laufen und sind somit eigenständige kleine Anwendungen. Das Web-Frontend greift über die JavaScript Services auf die verschiedenen RESTful-APIs der Ports zu. Ein Beispiel eines solchen Services ist in Quelltext 8.5 zu sehen.

```
reghipsterApp.factory('BusStop', function ($resource) {
      return $resource('http://localhost:8086/app/rest/stops', {}, {
         'query': { method: 'GET', isArray: true},
         'get': { method: 'GET'}
      });
});
```

Quelltext 8.5: JavaScript Service, der auf den Haltestellen-Microservice zugreift

Hierbei greift die Web-Anwendung, die auf Port 8080 läuft, auf Port 8086 zu (Codezeile 2), auf welchem der Microservice Haltestellenverwaltung gestartet wurde. Auf diese Weise werden die Komponenten der Anwendung mit einander verbunden. Eine Übersicht über das gesamte Betreiber-Backend inklusive aller Microservices, Datenbank und sonstigen Komponenten bietet Abbildung 8.3.

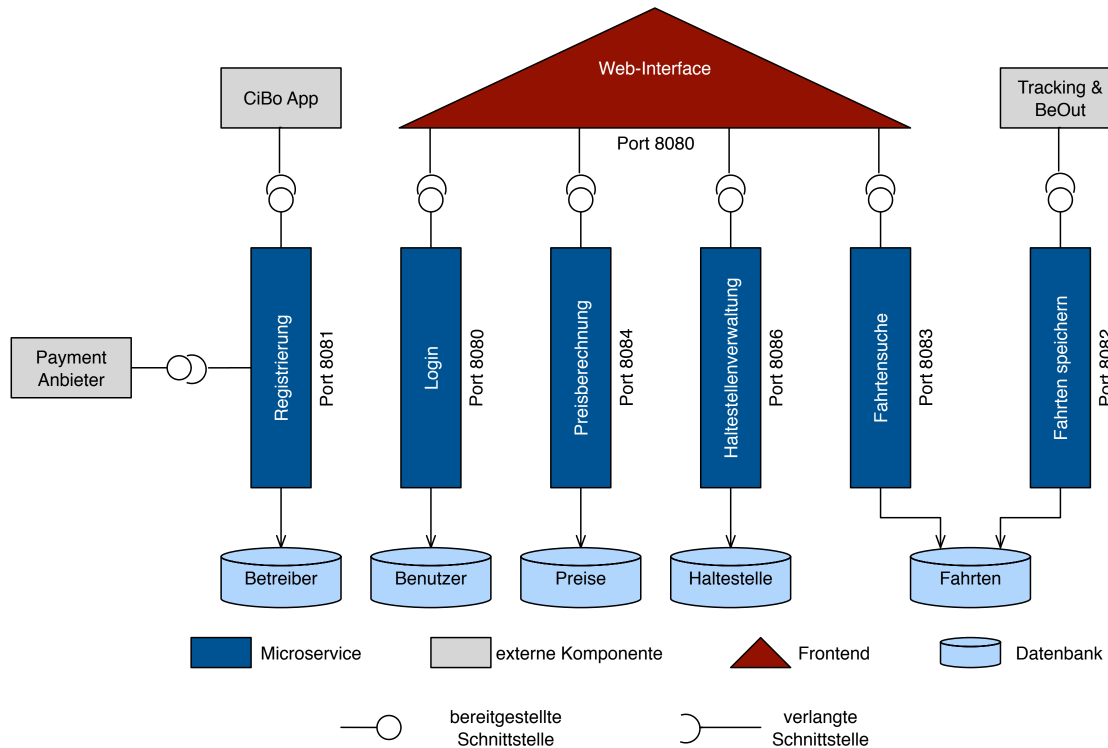


Abbildung 8.3: Übersicht über das Betreiber Backend

## 8.5 Testen

Um die Microservices zu testen wurde auf zwei verschiedene Arten vorgegangen. Zum einen wurden automatische Unittests mit Hilfe von JUnit [20] durchgeführt. JUnit ist ein Framework zum Testen von Klassen und Methoden von Java Programmen. Hiermit konnte das korrekte Verhalten der Komponenten getestet werden, wie beispielsweise, ob die richtigen Daten in die Datenbank geschrieben werden. Ein Beispiel eines solchen JUnit Tests ist in Quelltext 8.6 zu sehen. Die Methode `testBus()` (Zeile 27) testet, ob Businformationen korrekt in die Datenbank geschrieben und wieder ausgelesen werden.

Die Methode `testTrip()` (Zeile 39) tut dies für die Fahrten.

```
1  @RunWith(SpringJUnit4ClassRunner.class)
2  @SpringApplicationConfiguration(classes = Application.class)
3  @WebAppConfiguration
4  @Transactional
5  public class DatabaseTest {
6    private static BusLine busLine1;
7    private static BusLine busLine2;
8    private static Trip trip;
9
10   @Inject
11   private TripRepository tripRepository;
12
13   @Inject
14   private BusLineRepository busLineRepository;
15
16   @BeforeClass
17   public static void setUpBeforeClass() throws Exception {
18
19     //Initialize busLine
20     busLine1 = new BusLine();
21     busLine1.setBusName("12");
22     busLine2 = new BusLine();
23     busLine2.setBusName("22");
24   }
25
26   @Test
27   public void testBus() throws Exception {
28     busLineRepository.save(busLine1);
29
30     BusLine testBus = busLineRepository.findOne(busLine1.getBusId());
31     assertThat(testBus).isEqualTo(busLine1);
32
33     busLineRepository.delete(busLine1);
34     testBus = busLineRepository.findOne(busLine1.getBusId());
35     assertThat(testBus).isNull();
36   }
37
38   @Test
39   public void testTrip() throws Exception {
40     busLineRepository.save(busLine2);
41
42     BusLine test = busLineRepository.findOne(busLine2.getBusId());
43     assertThat(test).isNotNull();
44
45     //Initialize trip
46     trip = new Trip();
47
```

```
48      //initialize timestamps
49      Timestamp startTimestamp =
50             Timestamp.valueOf("2014-12-04 10:28:11");
51      Timestamp stopTimestamp =
52             Timestamp.valueOf("2014-12-04 10:31:19");
53
54      //set trip information
55      trip.setStartTimestamp(startTimestamp);
56      trip.setStopTimestamp(stopTimestamp);
57      trip.setStartStopId(1);
58      trip.setEndStopId(2);
59      trip.setBusLine(busLine2);
60      trip.setUserCiboId("user");
61
62      tripRepository.save(trip);
63      Trip testTrip = tripRepository.findOne(trip.getTripId());
64      assertThat(testTrip).isEqualTo(trip);
65
66      tripRepository.delete(trip.getTripId());
67      testTrip = tripRepository.findOne(trip.getTripId());
68      assertThat(testTrip).isNull();
69
70      busLineRepository.delete(busLine2);
71    }
72
73  }
```

Quelltext 8.6: JUnit Test der Fahrtenspeicherung

Um die RESTful-interfaces der Microservices zu testen wurde Swagger verwendet. Dadurch konnte verifiziert werden, dass die RESTful APIs die Daten korrekt entgegen nehmen und darauf hin die korrekten Informationen bereit stellt.
Die Entwicklung der Microservices erfolgte nacheinander und auch erst dann, wenn die Tests des vorherig implementierten Service erfolgreich absolviert wurden.

## 8.6 Zusammenfassung

Dieses Kapitel hat die Implementierung der Komponenten des Betreiber-Backends betrachtet. Auf Basis des Datenmodells aus dem vorherigen Kapitel, wurde eine relationale Datenbank realisiert. Die einzelnen Entitäten wurden durch Entity-Klassen in den Microservices erzeugt. Das die Web-Komponente basiert auf AngularJS und greift über Services auf die einzelnen Microservices zu, um Daten zu erhalten und über ein Interface darstellen zu können. Dabei laufen alle Microservices als eigenständige Anwendungen auf verschiedenen Ports. Tests wurden als automatische Unittests und manuellen Integrationstests durchgeführt.
Auf Basis dieser Implementierung wird nun im nächsten Kapitel die Entwicklung von Microservices evaluiert. Es wird überprüft, ob sich eine solche Architektur als

Erweiterung etablierter Komponententechnologien eignet.

# 9 Evaluation

Inhalt

9.1 Microserivces und der Komponentenbegriff . . . . . . . . . . . . . . . . . 69
9.2 Beurteilung von Microservices . . . . . . . . . . . . . . . . . . . . . . . . . 70
9.3 Erkannte Herausforderungen und Fragestellungen . . . . . . . . . . . . . 72
9.4 Fazit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

In diesem Kapitel wird evaluiert ob Microservices zusammen mit Spring Boot als Komponententechnologie betrachtet werden können. Anschließend erfolgt eine Bewertung anhand von Software-Qualitätsmerkmalen. Außerdem wird die zentrale Frage der Arbeit, ob Microservices als Erweiterung etablierter Komponententechnologien gesehen werden kann, geklärt. Zum Ende dieses Kapitels werden noch Herausforderungen und Fragestellungen erläutert, die bei der Entwicklung von Microservices im Rahmen dieser Arbeit erkannt wurden.

## 9.1 Microserivces und der Komponentenbegriff

In Abschnitt 3.1 wurden Eigenschaften einer Komponententechnologie dargestellt. Anhand dieser Definition wird nun erläutert, ob der Microservices-Architekturansatz in Verbindung mit Spring Boot als Komponententechnologie betrachtet werden kann. Dafür wird zunächst überprüft, ob ein Microservice die Eigenschaften einer Komponente erfüllt.

### 9.1.1 Komponenten

In der Definition des Komponentenbegriffs (Abschnitt 3.1) wurden einer Komponente drei Eigenschaften (Unabhängigkeit, Kompositionseigenschaft und Zustandslosigkeit) zugeschrieben. Diese werden nun auf Microservices bezogen

**Unaghängigkeit** Ein Microservice ist unabhängig. Er bildet eine abgeschlossene Einheit, die eigenständig deployed werden kann. Interne Operationen und Daten sind von außen unzugänglich. Es besteht lediglich durch APIs eine Kommunikation nach außen.

**Kompositionseigenschaft** Es lassen sich mehrere Microservices zusammenschließen. Durch Interfaces definieren sie, welche Dienste sie bereitstellen und welche sie in Anspruch nehmen. Das im Rahmen dieser Arbeit entwickelte Programm zeigt, dass Microservices Teil einer Komposition seien können. Es ist also ein Zusammenschluss

von Microservices über ihre Interface möglich, welcher wieder als Komponente betrachtet werden kann. Somit ist die Kompositionseigenschaft für Microservices erfüllt.

**Zustandslosigkeit** Auch Microservices besitzen keinen dauerhaften Zustand. Wenn nach Abarbeitung eines Microservice die Kontrolle an eine andere Komponente übergeht, geht auch der Zustand des Microservices verloren. Beispielsweise der Fahrtensuche Microservices liefert bei gleichen Anfragen immer die gleiche Rückgabe, unabhängig von den vorherigen Transaktionen. Somit ist auch die Eigenschaft der Zustandslosigkeit erfüllt.

Alle Microservices erfüllen die gleichen Eigenschaften wie eine Komponente, somit sind Microservices Komponenten.

### 9.1.2 Komponententechnologie

Microservices können als Komponenten betrachtet werden. Erfüllen diese in Verbindung mit Spring Boot die Eigenschaften einer Komponententechnologie? (Für Definition einer Komponententechnologie vgl. Abschnitt 3.1)

- Eine Komponententechnologie legt den Rahmen für Entwicklung und Ausführung von Komponenten fest. Auch Microservices tut dies: Microservices sollen so entkoppelt wie möglich Implementiert werden und auch technologisch unabhängig sein können (vgl. Abschnitt 5.1).
- Eine Komponententechnologie stellt Anforderungen hinsichtlich Kompositions- und Kollaborationsmöglichkeiten. Microservices sollen über APIs kommunizieren und somit definieren, welche Funktionen sie anbieten und welche sie benötigen. Generell lassen sie sich zusammenschließen, sollen aber so wenig wie möglich untereinander kommunizieren um so entkoppelt wie möglich zu sein. Somit gibt Microservices auch in diesem Punkt klare Vorgaben, wie es für eine Komponententechnologie typisch ist.
- Eine Komponententechnologie stellt eine Infrastruktur bereit. Diesen Punkt erfüllen Microservices zwar nicht, da es sich nur um eine Architektur handelt, jedoch wird diese Eigenschaft durch Spring Boot erfüllt. Spring Boot unterstütz alle Java EE Standard APIs und stellt einen eingebetteten Web-Container bereit.

Da alle Eigenschaften erfüllt sind, können Microservices in Verbindung mit Spring Boot als eine Komponententechnologie betrachtet werden.

## 9.2 Beurteilung von Microservices

Auf Basis der im vorherigen Abschnitt gewonnenen Kenntnisse, dass Microservices gemeinsam mit Spring Boot als Komponententechnologie betrachtet werden kann, erfolgt nun eine Beurteilung dieser.

### 9.2.1 Bewertung anhand von Software-Qualitätsmerkmalen

Die Bewertung erfolgt anhand der gleichen Software-Qualitätsmerkmalen, die auch in Kapitel 4 verwendet wurden.

**Änderbarkeit** Microservices weisen eine hohe Änderbarkeit auf. Die einzelnen Microservices sind voneinander entkoppelt und können somit leicht entnommen und ersetzt werden. Da es sich meist um sehr kleine Services handelt, können diese bei Anpassung der Anforderungen neu entwickelt werden. In dem in der Arbeit vorgestellten Beispiel musste bei Änderungen immer nur ein Microservice angepasst werden und eventuell noch die Web-Komponente. Mehr jedoch nicht, da die einzelnen Services nicht miteinander interagieren. Die Microservices wurden hier so „schlank“ wie möglich implementiert, was eine hohe Simplizität und Knappheit bedeutet. Dadurch, dass beispielsweise in dieser Arbeit Microservices von einander abgegrenzt entwickelt wurden und pro Anforderungsbereich ein Microservice entstand, ist die Strukturiertheit und damit auch die Lesbarkeit hoch.

**Testbarkeit** Microservices lassen sich gut testen. Man kann zunächst einen Microservice entwickeln und diesen anschließend auf volle Funktionsfähigkeit testen. Es wird immer dann erst der nächste Microservice entwickelt, wenn der vorherige funktioniert. Somit lässt sich der Code in kleinen Einheiten testen und es entsteht nicht am Ende ein hoher Bedarf an Tests. Man kann während der Implementierung schon immer sicher seien, dass die bisher geleistete Arbeit auch funktioniert. In manchen Fällen kann man auch auf Grund der Testerfahrung des vorherigen Microservice schon Fehler beim Entwickeln nachfolgender Microservices vorbeugen.

**Portabilität** Die Portabilität ist bei Microservices hoch. Zum einen weil Microservices plattformunabhängig sind. Zum anderen weisen die einzelnen Microservices eine hohe Abgeschlossenheit auf, da sie von anderen Komponenten so entkoppelt wie möglich entwickelt wurden.

### 9.2.2 Sinnvolle Erweiterung etablierter Komponententechnologien?

Microservices sind eine Architektur und können daher nicht als Komponententechnologie betrachtet werden. Die zentrale Frage dieser Arbeit war daher, ob sich Microservices als Erweiterung etablierter Komponententechnologien eignen. In dieser Arbeit wurden die Microservices mit Hilfe von Spring Boot realisiert um eine Infrastruktur zu erhalten und es wurde zu beginn dieses Kapitels evaluiert, ob Microservices in Verbindung mit Spring Boot eine Komponententechnologie darstellt. Da die Antwort positiv war, können Microservices als Erweiterung etablierter Komponententechnologien gesehen werden. In Unterabschnitt 8.1.1 wurde dargelegt, dass Spring Boot eine Komponententechnologie wie Java EE ist. Man könnte somit also auch Microservices in Verbindung mit Java EE realisieren. Microservices stellen dann zusätzliche Forderungen an die Implementierung, Komposition und ähnliches. Die Architektur wird auf Basis von Microservices entworfen, also so dass die Komponenten so simpel und so entkoppelt wie möglich realisiert werden.

Diese Erweiterung etablierter Komponententechnologien ist auch dahingehend sinnvoll, dass in Kapitel 4 festgestellt wurde, dass herkömmlich Komponententechnologien im Bezug auf Wartbarkeit noch ausbaufähig sind. Daher dienen Microservices als eine Verbesserung von Komponententechnologien, da sie die Wartbarkeit dieser erhöhen, wie in Unterabschnitt 9.2.1 gezeigt wurde.

## 9.3 Erkannte Herausforderungen und Fragestellungen

Bei der Implementierung der Software für diese Bachelorarbeit ergaben sich Herausforderungen und Fragestellungen bezüglich der Entwicklung von Microservices.

**Unterstützung** Zu Beginn der Implementierung stellte sich die Frage, wie Microservices realisiert werden. Zwar sind im Laufe des letzten Jahres diverse Artikel zu diesem Thema erschienen, allerdings erklären diese meist nur grob was Microservices sind und welche Vor- bzw. Nachteile diese haben. Es gibt jedoch keine Tutorials oder Beispiele wie man Microservices entwickelt. Der Grund dafür ist, dass Microservices noch neu sind. Wer Microservices entwickeln möchte, muss selbst viele Entscheidungen treffen und eine Vorgehensweise finden. Bei etablierten Komponententechnologien ist dies anders. Beispielsweise Java EE hat eine sehr große Gemeinschaft, die aus sehr erfahrenen Java EE entwicklen besteht. Man bekommt also sehr viel Unterstützung von erfahrenen Entwicklern. Ziel dieser Arbeit war es diesem Problem entgegenzuwirken, indem Microservices und deren Entwicklung genauer betrachtet werden. Allerdings stellte sich die Entwicklung dieser ohne gute Unterstützung durch beispielsweise Tutorials oder Literatur zunächst als Herausforderung dar.

**Micro** Bei der Entwicklung von Microservices muss darauf geachtet werden, diese vom Umfang tatsächlich klein zu halten und nur genau die Funktionalitäten zu implementieren, die für diesen Microservice notwendig sind. Beim Fahrtenspeicherungs-Microservice beispielsweise wurde zunächst eine Methode entwickelt um nach Nutzern in der Datenbank zu suchen, mit dem Gedanken, dass diese Funktionalität später sicherlich benötigt wird. Beim Überdenken der Methode, wurde jedoch klar, dass diese zur Fahrtensuche gehört. Bei der Entwicklung von Microservices war es daher nötig sich immer bewusst zu machen, dass die Microservices so klein wie möglich entwickelt werden.

**Entkoppelung** Während der Entwicklung der Software kam die Frage auf, wie vermieden werden kann, dass zwei Microservices die gleichen Entitäts-Klassen benötigen. Die Microservices sollen nicht von einander abhängen, daher wäre eine Einbindung von einem Microservices in den anderen keine zufriedenstellende Lösung. Beispielsweise bei der Entwicklung des Microservices Fahrtenspeicherung, stellte sich die Frage, ob auch hier User Entitäten benötigt werden. Im Datenmodell stehen Fahrten und Nutzer in einer Relation (n zu 1), allerdings würde man dann auf die gleiche Datenbank zugreifen müssen. Außerdem müsste die

Fahrtenspeicherung die User-Klasse des Registrierungs-Microservice importieren (ein ähnliches Problem wurde im Zusammenhang mit der Haltestellenverwaltung in Unterabschnitt 7.2.7 beschrieben). Dadurch würde eine Kopplung zwischen Fahrtenspeicherung und Registrierung entstehen. Die Lösung hierbei war, dass nur die CiBoId des Nutzers als reduzierte Entität in der Fahrtenspeicherung verwendet wird und alles weiteren Nutzerinformationen im Registrierungs-Microservice verwaltet werden. Dies ist zunächst im Hinblick auf Microservices eine gute Lösung, da die Microservices voneinander vollkommen abgeschirmt sind und keine Code-Redundanzen beinhalten. Ein Problem ergibt sich dabei dennoch. Dadurch dass die Datenbanken nicht mit einander verbunden sind, kann es vorkommen, dass in der einen Datenbanktabelle noch Werte enthalten sind die in der anderen bereits gelöscht wurden. Löscht beispielsweise ein Nutzer seinen Account und hat aber einige Fahrten vorher durchgeführt, ist seine CiBoId noch in der Fahrtendatenbanktabelle enthalten. Somit ist die referentielle Integrität nicht mehr gewährleistet, wodurch die Datenbank nicht konsistent ist.

## 9.4 Fazit

Zusammengefasst lässt sich sagen, dass Microservices die Komponenteneigenschaften erfüllen. Sie stellen beispielsweise in Verbindung mit Spring Boot eine Komponententechnologie dar und können somit als Erweiterung etablierter Komponententechnologien betrachtet werden. Dabei legen Microservices den Rahmen für die Entwicklung von Komponenten fest und stellen Anforderungen hinsichtlich Kompositions- und Kollaborationsmöglichkeiten. Die Infrastruktur wird von der Komponententechnologie wie beispielsweise Spring Boot oder Java EE bereitgestellt. Auf diese Weise wird die Wartbarkeit von Software verbessert, wodurch Verwendung von Microservices durchaus sinnvoll ist. Allerdings sollte bei der Entwicklung darauf geachtet werden die Microservices so klein und so entkoppelt wie möglich zu realisieren.

# 10 Zusammenfassung und Ausblick

Inhalt

10.1 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
10.2 Ausblick . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

Da im vorherigen Kapitel Microservices evaluiert wurden und damit auch die zentrale Fragestellung dieser Bachelorarbeit geklärt ist, wird die Arbeit nun zusammengefasst. Anschließend erfolgt ein Ausblick auf das, was im Rahmen dieser Arbeit nicht realisiert wurde und in welche Richtung das Thema Microservices weiterentwickelt werden könnte.

## 10.1 Zusammenfassung

Komponententechnologien unterstützen Softwareprojekte in der Entwicklungsphase und bieten eine bessere Übersicht. Dabei wird unter anderem die Wartbarkeit durch das Komponentenkonzept verbessert. Die Bewertung aus Kapitel 4 zeigte jedoch, dass Komponententechnologien in Bezug auf Wartbarkeit noch ausbaufähig sind. Gerade bei kleineren Softwareprojekten können Komponententechnologien mit ihren Spezifikationen und ihrer Infrastruktur diese unnötig verkomplizieren. Es wurde überprüft, ob Microservices dies verbessern und somit eine sinnvolle Erweiterung von Komponententechnologien darstellen.
Die Umsetzung des Betreiber-Backends des CiBo Projekts zeigte (vgl. Kapitel 7), dass sich Microservices gut entwerfen lassen, da ihre Architektur ähnlich realisiert werden kann. Es wurden sechs Microservices entwickelt (Registrierung, Login, Fahrtenspeicherung, Fahrtensuche, Preisberechnung und Haltestellenverwaltung), die alle aus den Schichten Persistierung, Service und REST-Controller bestehen. In der REST-Controller Schicht werden RESTful-APIs bereitgestellt, die als Schnittstellen nach außen dienen. Mit Hilfe des Vermittler-Entwurfsmusters lassen sich Microservices gut über das Web-Frontend steuern, wodurch die Entkoppelung zwischen den Microservices ermöglicht wird. Die Microservices kommunizieren somit nur mit dem Web-Frontend oder aber mit externen Komponenten wie beispielsweise der CiBo App nicht aber untereinander.
Mit Unterstützung der Komponententechnologie Spring Boot, wurde das Betreiber-Backend implementiert (vgl. Kapitel 8). Die Microservices beinhalten jeweils eine eigenständige Datenbank, da sie unterschiedliche Entitäten des Datenmodells benötigen. Somit wurde bei der Datenbank auf referentielle Integrität verzichtet, wodurch eine Entkopplung der Microservices gewährleistet werden kann.

Die Evaluation (vgl. Kapitel 9) zeigt, dass Microservices die Komponenteneigenschaften erfüllen und in Verbindung mit Spring Boot auch die Eigenschaften einer Komponententechnologie. Eine anschließende Bewertung verdeutlicht, dass das Konzept von Microservices eine sinnvolle Erweiterung etablierter Komponententechnologien darstellt, da somit die Wartbarkeit von Software verbessert wird.

## 10.2 Ausblick

Es gibt in Bezug auf das Betreiber-Backend noch Funktionalitäten, die in dieser Bachelorarbeit nicht realisiert wurden, da sie über den Rahmen der Arbeit hinausgehen. Des Weiteren lässt sich das Thema Microservices noch intensiver untersuchen.

**Payment Anbindung** Das Betreiber-Backend ist immer noch ein Prototyp. Der Payment-Anbieter ist beispielsweise simuliert worden. Der nächste logische Schritt wäre, eine Anbindung an einen realen Payment-Anbieter zu realisieren, über den tatsächlich die Zahlung einer getätigten Fahrt durchgeführt werden kann. Dafür müsste der Registrierungs-Microservice angepasst werden, der auf die vom Payment-Anbieter bereitgestellten APIs zugreift.

**Preisberechnung** Die in dieser Arbeit implementierte Preisberechnung liest Preise, die auf der zurückgelegten Strecke einer Fahrt basieren, aus einer Datenbank. Hilfreich und vor allem benutzerfreundlich wäre jedoch eine „intelligente" Preisberechnung. Diese könnte einem Benutzer ein Monatsticket berechnen, sobald dieses günstiger wäre als die Summe der Preise der Fahrten, die der Nutzer in dem Monat durchgeführt hat. Dafür müsste sowohl ein neues Tarifmodell erstellt als auch der Microservice Preisberechnung angepasst werden.

**Referentielle Integrität** In der Evaluation (vgl. Kapitel 9) wurde erwähnt, dass referentielle Integrität ein Problem darstellt, da die Microservices nicht auf eine gemeinsame Datenbank zugreifen. Es wäre sinnvoll, das Betreiber-Backend dahingehend zu erweitern, dass dieses Problem gelöst wird. Beispielsweise in dem automatisch über eine RESTful-API des Fahrtenspeicherungs-Microservice alle Fahrten eines Nutzers gelöscht werden, wenn dieser aus der User-Datenbank entfernt wird. Damit wird sichergestellt, dass die CiBoId des Nutzers nach seiner Löschung nicht mehr im System vorhanden ist.

**Fahrtenübersicht für Nutzer** Im Web-Interface des Betreiber-Backends werden mit Hilfe des Fahrsuche-Microservice alle Fahrten dargestellt. Eine hilfreiche Erweiterung wäre es, eine solche Übersicht auch dem Nutzer für seine getätigten Fahrten zu bieten. Beispielsweise könnte über die App eine solche Fahrtenübersicht dargestellt werden. Hierfür würde die App sogar die gleiche API nutzen, die auch das Betreiber-Backend verwendet. Die App müsste dazu eine Anfrage, gefiltert nach dem Nutzer, senden um somit alle Fahrten dieses Nutzers zu erhalten.

**Wiederverwendbarkeit** Hinsichtlich Microservices wäre es interessant eine ähnliche Web-Anwendung zu entwickeln. Auf Basis einer solchen Implementierung könnte man Microservices genauer auf Wiederverwendbarkeit überprüfen. Beispielsweise, ob es mit wenig Aufwand möglich ist einen Login-Microservice in einem anderen Softwareprojekt wiederzuverwenden. Ist die Wiederverwendbarkeit bei Microservices hoch, so würde Open Source eine interessante Möglichkeit bieten. Es könnten Microservices einfach für alle zugänglich bereitgestellt werden, um anderen Programmierern beispielsweise einen Login oder eine Registrierung vorgefertigt zur Verfügung zu stellen. Softwareprojekte könnten somit aus Bausteinen in Form von Microservices zusammen gestellt und nach eigenen Anforderungen erweitert werden.

**Integrierte Microservices** In dieser Arbeit wurden Microservices mit Hilfe von Spring Boot realisiert. Dieses Framework ist allerdings nicht auf die Entwicklung von Microservices ausgelegt. Dadurch erstellt man bei einer Anwendung für jeden Microservice eine Spring Boot Applikation. Hilfreich wäre ein Framework, über das sich eine Menge an Microservices verwalten lässt. Vielleicht gibt es ein Hybridmodell, welches Microservices als feste Komponentenarchitektur in eine Komponententechnologie integriert, die die wesentlichen Eigenschaften von Microservices enthält.

# A Anhang

## A.1 Web-Interface Screenshots

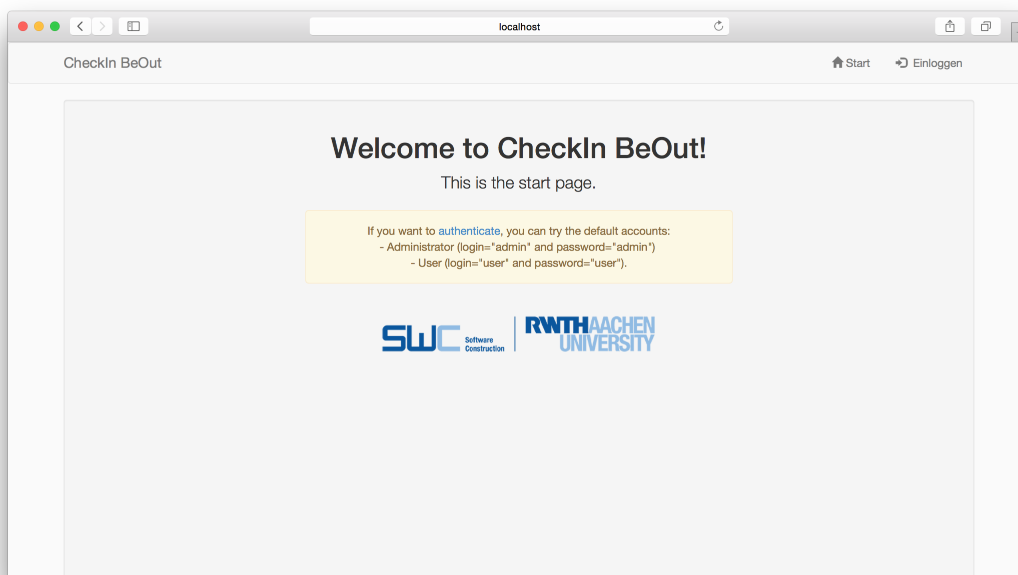


Abbildung A.1: Screenshot der Startseite

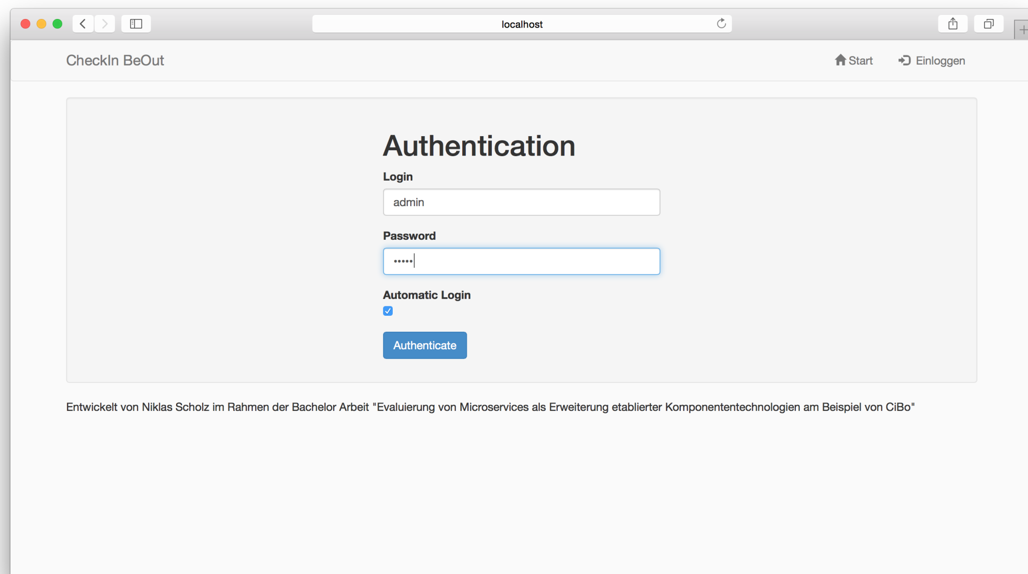


Abbildung A.2: Screenshot des Logins

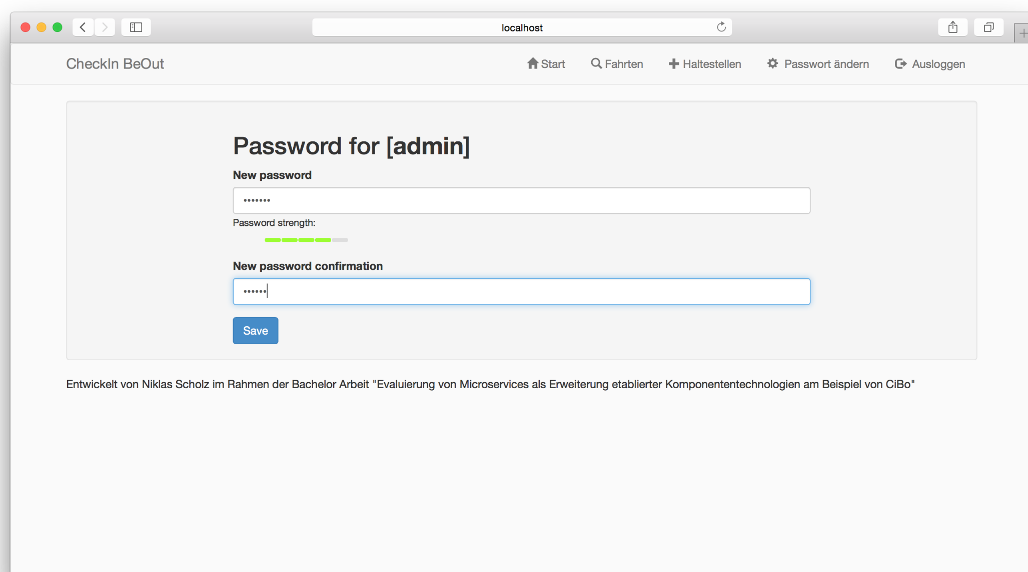


Abbildung A.3: Screenshot der Passwortänderung

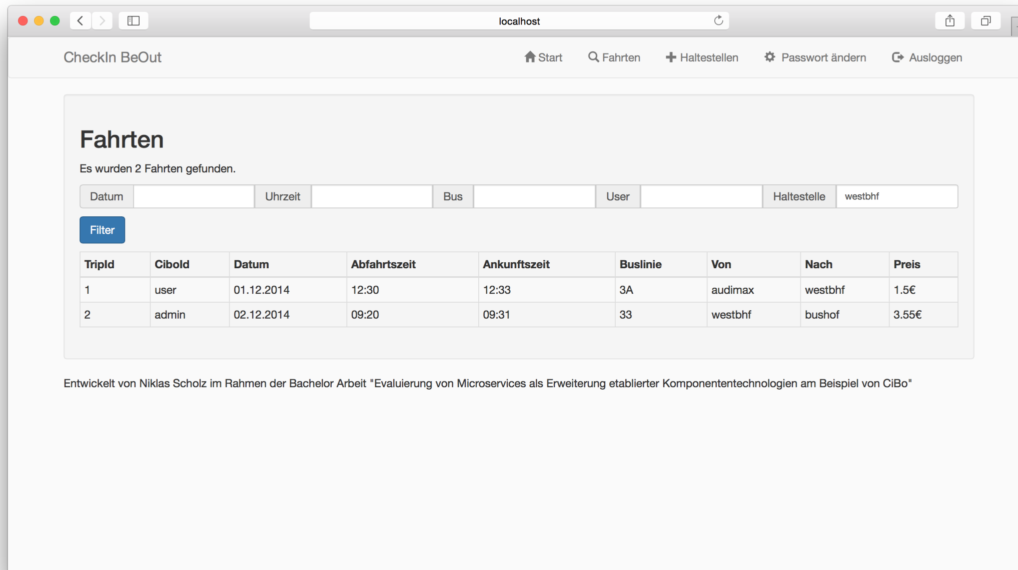


Abbildung A.4: Screenshot der Fahrtensuche

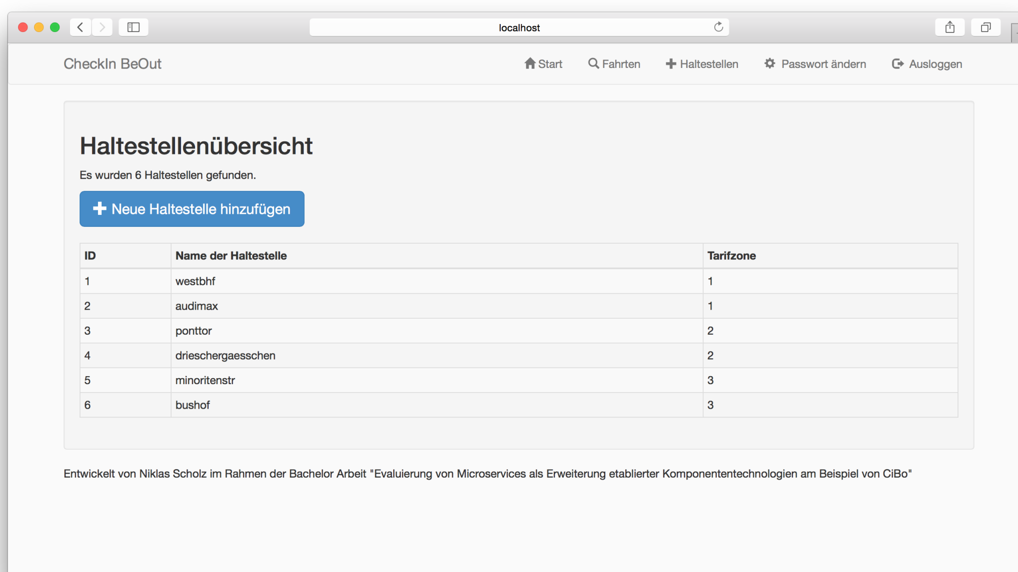


Abbildung A.5: Screenshot der Haltestellenübersicht

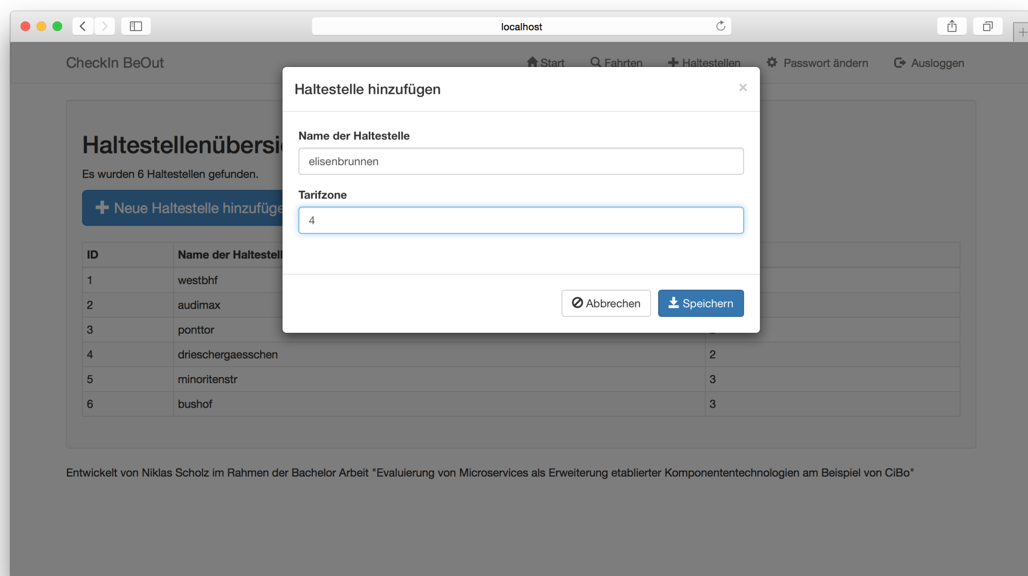


Abbildung A.6: Screenshot neue Haltestelle hinzufügen

# Literaturverzeichnis

[1] Atlassian: *Lifecycle of a Bundle*. 2013. – URL https://developer.atlassian.com/display/DOCS/Lifecycle+of+a+Bundle. – Zugriffsdatum: 20.10.2014

[2] Beer, Wolfgang ; Birngruber, Dietrich ; Mössenböck, Hanspeter ; Prähofer, Herbert ; Wöss, Albrecht: *Die .NET-Technologie*. 2. Auflage. dpunkt.verlag GmbH, 2006

[3] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000

[4] Fowler, Martin: *Data Transfer Object*. – URL http://martinfowler.com/eaaCatalog/dataTransferObject.html. – Zugriffsdatum: 22.01.2015

[5] Fowler, Martin: *PolyglotPersistence*. 2011. – URL http://martinfowler.com/bliki/PolyglotPersistence.html. – Zugriffsdatum: 09.10.2014

[6] Fowler, Martin ; Lewis, James: *Microservices*. 2014. – URL http://martinfowler.com/articles/microservices.html. – Zugriffsdatum: 09.10.2014

[7] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster*. Addison-Wesley Verlag, 2004

[8] Gemert, Jan van: *Practical tutorial for using CORBA*. 2000. – URL https://staff.fnwi.uva.nl/j.c.vangemert//pub/learncorba.pdf. – Zugriffsdatum: 28.04.2014

[9] Goncalves, Antonio: *Beginning Java EE 7*. Apress Media LLC, 2013

[10] Google: *Android*. – URL http://www.android.com. – Zugriffsdatum: 23.01.2015

[11] Google: *AngularJS - Superheroic JavaScript MVW Framework*. – URL https://angularjs.org. – Zugriffsdatum: 13.01.2015

[12] Gruhn, Volker ; Thiel, Andreas: *Komponentenmodelle: DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Addison-Wesley Verlag, 2000

[13] Harlem-Law, Andrew: *Microservices - A reality check(point)*. 2014. – URL http://capgemini.github.io/architecture/microservices-reality-check/. – Zugriffsdatum: 31.10.2014

[14] Heineman, George T. ; Councill, William T.: *Component-Based Software Engeneering*. Addison-Wesley Verlag, 2001

[15] Henning, Michi: The Rise and Fall of CORBA. In: *Communications of the ACM* 51 (2008), Nr. 8, S. 52–57. – URL `http://cacm.acm.org/magazines/2008/8/5336-the-rise-and-fall-of-corba/fulltext`

[16] Horn, Torsten: *OSGi: Dynamisches Komponentensystem für Java*. 2009. – URL `http://www.torsten-horn.de/techdocs/java-osgi.htm`. – Zugriffsdatum: 27.01.2015

[17] Ihns, Oliver: *Umkehr der Beweislast*. 2005. – URL `http://jaxenter.de/artikel/Umkehr-Beweislast`. – Zugriffsdatum: 27.01.2015

[18] JHipster: *JHipster home page*. – URL `https://jhipster.github.io`. – Zugriffsdatum: 13.01.2015

[19] Josuttis, Nicolai: *SOA in der Praxis*. dpunkt.verlag GmbH, 2008

[20] JUnit: *JUnit*. – URL `http://junit.org`. – Zugriffsdatum: 29.01.2015

[21] Liquibase: *Documentation Home*. – URL `http://www.liquibase.org/documentation/`. – Zugriffsdatum: 13.01.2015

[22] Ludewig, Jochen ; Lichter, Horst: *Software Engineering*. 2. dpunkt.verlag GmbH, 2010

[23] Mason, Ross: *OSGi? No Thanks*. 2010. – URL `http://blogs.mulesoft.org/osgi-no-thanks/`. – Zugriffsdatum: 23.10.2014

[24] Micro Focus: *Orbacus*. – URL `http://microfocus.com/products/corba/orbacus/index.aspx`. – Zugriffsdatum: 26.01.2015

[25] Microsoft: *Major Car Insurer Sees 60 Percent Faster Time-to-Market on Windows Phone 7*. – URL `http://www.microsoft.com/casestudies/Microsoft-.NET-Framework-4/GEICO/Major-Car-Insurer-Sees-60-Percent-Faster-Time-to-Market-on-Windows-Phone-7/4000008644`. – Zugriffsdatum: 26.01.2015

[26] Microsoft: *Chapter 1: Introduction to .NET*. 2006. – URL `http://technet.microsoft.com/en-us/library/bb496996.aspx`. – Zugriffsdatum: 20.10.2014

[27] Microsoft: *Telefónica Drives Opportunities for Developers with Cloud and Mobile Services*. 2011. – URL `http://www.microsoft.com/casestudies/Microsoft-.NET-Framework/Telefonica/Telefonica-Drives-Opportunities-for-Developers-with-Cloud-and-Mobile-Services/4000009463`. – Zugriffsdatum: 23.10.2014

[28] Microsoft: *Get Started with ASP.NET.* 2014. – URL http://www.asp.net/get-started. – Zugriffsdatum: 22.10.2014

[29] Microsoft: *Globaler Assemblycache.* 2014. – URL http://msdn.microsoft.com/de-de/library/yf1d93sz(v=vs.110).aspx. – Zugriffsdatum: 21.10.2014

[30] Mikhalenko, Peter V.: *Enterprise Java Beans: Wann lohnt der Einsatz?* 2007. – URL http://www.zdnet.de/39156487/enterprise-java-beans-wann-lohnt-der-einsatz/. – Zugriffsdatum: 27.01.2015

[31] Mono Project: *Mono.* – URL http://www.mono-project.com. – Zugriffsdatum: 26.01.2015

[32] Neubauer, Bertram ; Ritter, Tom ; Stoinski, Frank: *CORBA Komponenten.* Springer Verlag, 2004

[33] Oracle: *GlassFish Server.* – URL https://glassfish.java.net. – Zugriffsdatum: 24.01.2015

[34] Oracle: JavaTM Platform Enterprise Edition (Java EE) Specification, v6. 2009. – Forschungsbericht

[35] OSGi Alliance: *Bundle (OSGi Service Platform Release 4 Core Specification Version 4.3).* 2012. – URL http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/Bundle.html. – Zugriffsdatum: 19.10.2014

[36] PHP Space Community: *Die Php Programmiersprache.* – URL http://www.php-space.info/php/. – Zugriffsdatum: 26.01.2015

[37] Richardson, Chris: *Microservices: Decomposing Applications for Deployability and Scalability.* 2014. – URL http://www.infoq.com/articles/microservices-intro. – Zugriffsdatum: 09.10.2014

[38] Rumpe, Bernhard: *Modellierung mit UML.* 2. Springer-Verlag, 2011

[39] Schmidt, Douglas C.: *Real-time CORBA with TAO (the ACE ORB).* – URL http://www.cs.wustl.edu/~schmidt/TAO.html. – Zugriffsdatum: 26.01.2015

[40] Skurrie, Beth: *A microservices implementation retrospective.* 2014. – URL http://techblog.realestate.com.au/a-microservices-implementation-retrospective/. – Zugriffsdatum: 09.10.2014

[41] Swagger: *Swagger 2.0.* – URL http://swagger.io. – Zugriffsdatum: 21.01.2015

[42] Taboada, P. G.: *Testbarkeit - Umkehr der Sinnhaftigkeit?* 2007. – URL http://pgt.de/2007/09/27/testbarkeit-umkehr-der-sinnhaftigkeit/. – Zugriffsdatum: 27.01.2015

[43] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Computing, 2011. – URL http://openbook.galileocomputing.de/javainsel9/javainsel_23_001.htm#mjd7254da57686a2ef9e5fcb69a2a97220

[44] University of Ottawa: *Object Oriented Software Engineering Knowledge Base: Glue Code*. – URL http://www.site.uottawa.ca:4321/oose/index.html#gluecode. – Zugriffsdatum: 28.01.2015

[45] Wikipedia: *Java Platform, Enterprise Edition*. – URL http://de.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition. – Zugriffsdatum: 24.01.2015

[46] Wikipedia: *Man-in-the-Middle-Angriff*. – URL http://de.wikipedia.org/wiki/Man-in-the-Middle-Angriff. – Zugriffsdatum: 26.01.2015

[47] Wikipedia: *.NET*. – URL http://de.wikipedia.org/wiki/.NET. – Zugriffsdatum: 26.01.2015

[48] Wikipedia: *ASP.NET*. 2014. – URL http://de.wikipedia.org/wiki/ASP.NET. – Zugriffsdatum: 23.10.2014

[49] Wikipedia: *Common Object Request Broker Architecture*. 2014. – URL http://de.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture. – Zugriffsdatum: 27.10.2014

[50] Wikipedia: *ISO/IEC 9126*. 2014. – URL http://de.wikipedia.org/wiki/ISO/IEC_9126. – Zugriffsdatum: 14.01.2015

[51] Wikipedia: *OSGi*. 2014. – URL http://de.wikipedia.org/wiki/OSGi#Enterprise_.2F_Application_Server. – Zugriffsdatum: 24.10.2014

[52] Wütherich, Gerd ; Hartmann, Nils ; Kolb, Bernd ; Lübken, Matthias: *Die OSGi Service Platform*. 1. dpunkt.verlag GmbH, 2008

[53] Yeoman: *Yeoman: The web's scaffolding tool for modern webapps*. – URL http://yeoman.io. – Zugriffsdatum: 28.01.2015

# Abkürzungsverzeichnis

**A**

**API** Application Programming Interface
**App** Applikation
**ASEAG** Aachener Straßenbahn und Energieversorgungs-AG

**B**

**BCL** Base Class Library

**C**

**CCM** CORBA Component Model
**CiBo** CheckIn BeOut
**CIL** Common Intermediate Language
**CLR** Common Language Runtime
**CLS** Common Language Specification
**CORBA** Common Object Request Broker Architecture
**CTS** Common Type System

**D**

**DII** Dynamic Invocation Interface
**DTO** Data Transfer Object

**E**

**EJB** Enterprise JavaBeans
**ER** Entity-Relationship

**G**

**GPS** Global Positioning System
**GUI** Graphical User Interface

**H**

**HTML** Hypertext Markup Language
**HTTP** Hypertext Transfer Protocol

**I**

**IDL** Interface Definition Language

**J**

**Java EE** Java Platform, Enterprise Edition
**JDK** Java Development Kit
**JIT** Just-in-time
**JSF** JavaServer Faces
**JSON** JavaScript Object Notation

**M**

**MOM** Message Oriented Middleware

**O**

**ORB** Object Request Broker

**R**

**REST** Representational State Transfer

**S**

**SDK** Software Development Kit
**SSL** Secure Sockets Layer

**U**

**URA** Unified Realtime API
**URL** Uniform Resource Locator