REST-konformes Re-Design eines bestehenden Webservices

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften ii
Studiengang Angewandte Informatik

1. Betreuer:	Prof. Dr. Christin Schmidt
2. Betreuer:	Prof. Dr.-Ing. Sebastian Möller

Eingereicht von:	Christopher Zell
Matrikelnummer:	531727
Email:	zelldon91@gmail.com

Datum:	12. August 2013

Danksagung

Hiermit möchte ich mich bei meinen Eltern, die mir das Studium ermöglichten, sowie bei meiner Freundin Jennifer Pogander, die mich immer unterstützte und meine Arbeiten zur Korrektur las, bedanken.
Des Weiteren bedanke ich mich bei Tilo Westermann, meinen Mentor bei den Telekom Innovation Laboratories (T-Labs) für seine Unterstüzung und seinen Rat, sowie bei Frau Prof. Schmidt die bei Fragen immer ansprechbar war.

Als letztes möchte ich mich bei Christopher Kruczek bedanken, der im Laufe des Studiums mir auch immer mit Rat und Tat zur Seite stand.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

1 Einleitung

In dieser Arbeit wird das Thema Representational State Transfer (REST), *RESTful* und *RESTful* Webservices erläutert. Es wird ein bestehender Java Webservice basierend auf *Servlets* neu erstellt bzw. Re-Designed, sodass dieser erstellte Webservice den REST-Kriterien entspricht bzw. einer *RESTful* Webservice-Architektur. Re-Designed wird in dieser Arbeit wie folgt definiert: Neuerstellung eines bestehenden Webservice mit verbesserter Architektur. Für die Umsetztung des Webservices wird die Resource-Oriented Architecture (ROA) und deren, von Richardson und Ruby generischen Entwurfsverfahren, verwendet. Die Vorteile von REST und auch des Erstellten *RESTful* Webservices sowie deren Aufwand wird aufgezeigt. Der Aufwand einer *RESTful* Webservice-Architekturimplementation wird anhand einer *framework* und *Servlet* Lösung verglichen und somit dargestellt, welche Lösung für die Umsetzung eines *RESTful* Webservices geeigneter ist.

1.1 Motivation

Webservices werden immer mehr mit mobilen Clients verwendet, wodurch diese bestimmten Anforderungen, wie z.B. Skalierbarkeit, entsprechen müssen. REST oder *RESTful* Webservices gewinnt daher auch mehr und mehr an Bekanntheitsgrad. Jedoch wissen nicht viele, wobei es sich genau dabei handelt und wie man solch einen Webservice umsetzt. Zumeist sind diese sogenannten *RESTful* Webservices gar nicht *RESTful*. Passend zu diesem Thema ist auch folgendes Zitat von Richardson und Ruby:

> » Both REST and web services have become buzzwords. They are chic and fashionable. These terms are artfully woven into PowerPoint presentations by people who have no real understanding of the subject.«
>
> Richardson und Ruby 2007, S. 314

Um einen Einblick und auch eine Erklärung für dieses Thema zu bieten, wird diese Arbeit erstellt.

1.2 Aufgabenstellung

Die praktische Aufgabe in dieser Bachelorarbeit besteht darin einen bestehenden Webservice, der auf Java-*Servlets* aufbaut, auf eine REST-konforme Architektur umzustrukturieren. Als weitere Idee kam dabei auf dass man aufbauend auf dieser Architektur verschiedene Implementierungsweisen aufzeigen und vergleichen könnte. Dabei ist die Implementierung einer REST-konformen Architektur über *Servlets* und *frameworks* gemeint. Der theoretische Teil dieser Arbeit befasst sich mit der Erläuterung von REST, *RESTful* Webservices und einer *RESTful* Architektur, nach der der Webservice erstellt bzw. der bestehende Re-Designed wird. Zudem wird der Mehraufwand zwischen den bereits erwähnten Implementationsverfahren aufgezeigt und die Vorteile des neu erstellten *RESTful* Webservices gegenüber dem bestehenden. Der praktische Teil dieser Arbeit ist im Rahmen eines Forschungsprojekts am "Quality and Usability Lab" der T-Labs, Technische Universität (TU) Berlin entstanden.

> »Die Telekom Innovation Laboratories (T-Labs) sind der zentrale Forschungs- und Innovationsbereich (F&I) der Deutschen Telekom. Organisatorisch gehören sie zum Verantwortungsbereich des Chief Product and Innovation Officers.
> Der Auftrag der T-Labs ist es, in enger Zusammenarbeit mit den operativen Einheiten der Telekom neue Impulse und Unterstützung bei der Entwicklung und Umsetzung innovativer Produkte, Dienste und Infrastrukturen für die Wachstumsfelder der Telekom zu liefern«
>
> TLabs 2013b

Das Forschungsprojekt dient der Evaluation von mobilen Clients im Einsatz. Dafür wurde eine mobile Applikation für Studenten des Campus Charlottenburg entwickelt. Die Applikation nennt sich Mobiler Campus Charlottenburg (MoCCha). Der Server der die Daten für diese Applikation bereitstellt soll innerhalb dieser Arbeit, nach REST-konformität, neu entwickelt werden.

1.3 Ziele

Das grundlegende Ziel ist das Thema REST-konformes Re-Design eines bestehenden Webservice (*RESTful* Webservice). D.h. es soll der bestehende Webservice anhand einer *RESTful* Architektur Re-Designed, erneuert bzw. verbessert werden. Der zu erstellende Webservice soll unter den bestehenden REST-Bedingungen erstellt werden. Wenn möglich, sollte gezeigt werden welche Vorteile dadurch entstehen. Für einen *RESTful* Webservice, geschrieben in Java, gibt es mehrere Wege diesen zu implementieren (mit *Servlets* oder mit *frameworks*). Durch die Annahme das die Implementierung mit *Servlets* anstatt mit *frameworks* durchaus aufwendiger ist, beanspruchen die verschiedenen Wege auch unterschiedlichen Arbeitsaufwand. Diese Annahme sollte in dieser Arbeit des Weiteren bewiesen werden.

Grundlagen
2

2.1
Webservice

Laut Kemper und Eickler und den W3C Autoren Booth u. a. wird ein Webservice wie folgt definiert: Ein Webservice ist ein Programm, das auf Anfragen (engl. requests) innerhalb des Web's automatisiert reagiert. Die Anfragen werden über Hypertext Transfer Protocol (HTTP) übermittelt. Das Ziel eines Webservices ist das Austauschen von Datensätzen bzw. Informationen. Diese können in den verschiedensten Datenformaten ausgetauscht werden, wie z.B. mit Javascript Object Notation (JSON), Extensible HyperText Markup Language (XHTML), Extensible Markup Language (XML) usw.. Da die Daten wie zuvor beschrieben über HTTP übermittelt werden, spielt es dabei keine Rolle mit welcher Programmiersprache oder auf welchem Betriebssystem der Webservice läuft. Um mit einem Webservice kommunizieren zu können spielt nicht nur HTTP eine Rolle sondern auch der Uniform Resource Identifier (URI) oder auch Uniform Resource Locator (URL) gennant. Mithilfe der URI kann ein Client (engl. client) via HTTP eine Anfrage an den Webservice senden. Ein Webservice ist nicht für menschliche Anwender gedacht sondern für automatisierte Anfragen bzw. Clients wie Smartphones, Webseiten oder andere. Es gibt mehrere verschiedene Möglichkeiten solch einen Webservice umzusetzen. Unter anderem spielen da Simple Object Access Protocol (SOAP), Remote Procedure Call (RPC) und *RESTful* eine große Rolle (vgl. Kemper und Eickler 2009, S. 624 sowie Booth u. a. 2013a).

2.1.1
RPC

Mithilfe von RPC ist es möglich an einem RPC Webservice von der Ferne (engl. remote) eine Methode (engl. procedure) aufzurufen. Meist besitzt ein RPC Webservice nur eine URI. Die URI eines RPC Webservices wird meist auch "endpoint" genannt. Dieser sogennante "endpoint" akzeptiert nur die HTTP-Methode POST. Ein Beispiel für die URI eines RPC Webservice wäre "www.rpc-example.de/rpc".

Um nun eine Methode per remote aufzurufen, muss via HTTP-Methode POST ein bestimmtes Datenformat innerhalb des HTTP-Envolpe an den Webservice gesendet werden. Meist spricht man auch von der Scoping-Information. Bei dem Datenformat handelt es sich zumeist um XML (vgl. Richardson und Ruby 2007, S. 15). Die sogenannte Scoping-Information ist ähnlich wie die Parameter einer Methode/Funktion. Meist sagt die Scoping-Information aus, an welchen Daten/Informationen eine Operation ausgeführt werden soll (vgl. ebd., S. 11 f.). Es wäre ein XML Datenaustauschformat denkbar, welches die Methode und die Werte für die Parameter enthält. (siehe Beispiel: 2.1)

```
<rpc-call>
  <method name="hello_world">
    <param name="date">
      2013-06-04
    </param>
    <param name="user">
      Zell
    </param>
  </method>
</rpc-call>
```

Beispiel 2.1: RPC-Call Scoping-Information

Der Name der aufzurufenden Methode ist ”hello_world”. Die ”hello_world” Methode erhält 2 Parameter. Ein Datum und einen User. Die Parameter könnten z.B. für die Erstellung des Resultats benutzt werden. Ein mögliches Resultat der aufgerufenen RPC-Methode wäre daher ”Hello Zell, Today is 2013-06-04”. Das Resultat wird im HTTP-*Envelope* im XML Datenaustauschformat als Antwort (engl. response) zurückgesendet. Dadurch das von Webservices zu Webservices neue oder andere Methoden erstellt werden und sich somit auch die Namen der Methoden ändern, baut jeder RPC Webservice immer auf neuen Methodennamen auf. Durch die immer neuen Methodennamen ist kein *uniform interface* gewährleistet, wie es bei einem *RESTful* Webservice der Fall ist (vgl. ebd., S. 14). In den Abschnitten 2.1.3 und 2.3 wird das *uniform interface* noch einmal aufgegriffen und erklärt.

2.1.2 SOAP

Ein Webservice der SOAP benutzt, hat ein besonderes Merkmal und zwar den sogenannten SOAP-*Envelope*. Laut Richardson und Ruby ist SOAP ein Umschlagsformat, ähnlich wie HTTP. Bis auf das SOAP ein XML basierendes Umschlagsformat ist (vgl. ebd., S. 19). Dieses Umschlagsformat wird benutzt um die Daten/Information ”einzupacken”. Der SOAP-*Envelope* wird wiederum in den HTTP-*Envelope* (*entity body*) gesteckt. Innerhalb des HTTP wird der HTTP-*Envelope* benutzt um Daten/Informationen zu übertragen. Die HTTP-*Header* gelten als Sticker auf dem HTTP-*Envelope*. SOAP besitzt auch sogenannte Sticker (*Header*), diese werden jedoch innerhalb des SOAP-*Envelope* mit eingebracht (vgl. ebd., S. 302). Der Sinn des SOAP-*Envelope* ist die Unabhängigkeit von dem Protokoll über welches es transportiert wird. Das bedeutet, man kann den SOAP-*Envelope* über jedes andere Protokoll versenden. Nicht nur über HTTP z.B. auch per Email, TCP oder andere (vgl. ebd., S. 303).

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
  <rpc-call>
    <method name="hello_world">
      <param name="date">
        2013-06-04
      </param>
      <param name="user">
        Zell
      </param>
    </method>
  </rpc-call>
</soap:Body>

</soap:Envelope>
```

Beispiel 2.2: SOAP-Call Scoping-Information

In dem Beispiel 2.2 wurde das Beispiel 2.1 aufgegriffen und an einen Webservice, der SOAP als Austauschdatenformat benutzt, angepasst. Das selbe passiert hier wieder wie im Beispiel 2.1. Es wird die Methode ”hello_world” mit zwei Parametern, User und Datum, aufgerufen.

2.1.3 RESTful

RESTful ist die Erfüllung aller REST-Kriterien. Hat ein Webservice alle REST-Kriterien erfüllt spricht man von einem *RESTful* Webservice. Die größten Merkmale eines *RESTful* Webservices sind das *uniform interface*, die Zustandslosigkeit, die Skalierbarkeit und die Einfachheit. REST bzw. die REST-Kriterien werden in dem Abschnitt 2.3 näher erläutert. Um den Unterschied zu einem RPC Webservice zu verdeutlichen, wird noch einmal das Beispiel 2.1 aufgegriffen. Die Domain in diesem Beispiel war ”www.rpc-example.de”. Diese wird für das *RESTful* Beispiel wiederverwendet. Bei den Beispielen 2.1 und 2.2 wurde eine Methode namens ”hello_world” mit 2 Parametern aufgerufen. Bei einem *RESTful* Webservice wird keine Methode aufgerufen sondern eine Ressource (engl. resource) angefragt. Die Anfrage wird mithilfe der HTTP GET-Methode gesendet und nicht wie im RPC Beispiel via POST. Die anzufragende Ressource wird ”greeting” heißen. Zur Erklärung bzw. Benutzung von den verschieden HTTP-Methoden siehe Abschnitt 2.4. Die Ressource wurde ”greeting” genannt, da der Ressourcenname gleich der URI ist. Dieser sollte selbsterklärend sein, siehe dazu auch Abschnitt 2.4.1 (vgl. Richardson und Ruby 2007, S. 84 f.). Somit lautet die gesamte URI ”www.rpc-example.de/greeting”. Die beiden Parameter User und Datum werden auch in die URI als Queryparamter eingefügt. Sodass die GET-Anfrage an die folgende URI ”www.rpc-example.de/greeting?user=zell&datum=2013-06-04” gesendet werden kann. Dadurch das nun die Scoping-Informationen in der URI und die Methodeninformation in der HTTP-Methode enthalten sind, beschreibt das Beispiel eine *RESTful* Architektur (vgl. ebd., S. 13).

Die HTTP-Methode GET wird an der ”greeting” Ressource deshalb benutzt, weil eine Begrüßung mit einem Namen und einem Datum ausgeliefert werden soll (”get a greeting”). Siehe dazu die Benutzung und Erklärung von GET in dem Abschnitt 2.4.6. Der Unterschied zu RPC und *RESTful* Webservices ist somit schon ersichtlich. Denn *RESTful* Webservices benutzen das ganze Spektrum der verfügbaren HTTP-Methoden und RPC nur POST (bzw. ”overloaded POST”). Dadurch wird auch klar, dass RPC HTTP nicht in seinem vollen Umfang bzw. dessen Stärken nutzt (vgl. Richardson und Ruby 2007, S. 15 f.).

2.1.4

Zusammenfassung

> »HTTP is not designed to be a transport protocol. It is a transfer protocol in which the messages reflect the semantics of the Web architecture by performing actions on resources through the transfer and manipulation of representations of those resources. [...]
> It therefore makes no sense to do those extensions on top of HTTP, since the only thing HTTP accomplishes in that situation is to add overhead from a legacy syntax.«
>
> Fielding 2000, S. 142

Wie aus dem Zitat 2.1.4 von Roy Fielding schon hervorgeht, macht es keinen Sinn HTTP mit SOAP zu erweitern und zu nutzen, da ein zu großer *overhead* dabei entsteht. Doch würden die Webservices die SOAP benutzen, auf HTTP verzichten und ihre Dokumente/Daten über andere Protokolle senden, wären sie nicht mehr Teil des Web's (vgl. Richardson und Ruby 2007, S. 4). Laut Richardson und Ruby macht in einem gut entworfenen *RESTful* Webservice alles genau das, was es auch besagt. Das heißt wenn eine GET-Anfrage an die Ressource aus dem Abschnitt 2.1.3 gesendet wird, (”www.rpc-example.de/greeting?user=zell&date=2013-06-04”) dann enthält die Antwort auch eine erwartete Begrüßung. Wenn jedoch bei dem RPC Webservice, wie aus dem Beispiel 2.1, eine POST-Anfrage gesendet (”www.rpc-example.de”) wird, ist nicht klar was als Antwort zurückkommt oder was überhaupt ausgelöst wird. Würde man der HTTP Definition von POST (siehe Abschnitt 2.4.6) folgen, könnte man denken das die Repräsentation (engl. representation) aus dem Beispiel 2.1 gespeichert wird und man als Antwort den Ort (URI) der erstellten Ressource zurückbekommt.

2.2 Media Type

Laut Fielding hat HTTP die Syntax zur Beschreibung des Nachrichtendatenaustauschformats (engl. media type) von Multipurpose Internet Mail Extensions (MIME) geerbt, sodass definierte media types wie auch für andere Internet-Protokolle benutzt werden können. MIME wird in Emails oder anderen Internet-Protokollen verwendet (vgl. Fielding 2000, S. 133). MIME definiert einen sogenannten *Header* namens Content-Type, der anzeigt welches Datenaustauschformat der body des MIME Dokuments beinhaltet. Das Format wird in media type und subtype identifier unterteilt. So entsteht folgende Syntax "media type/subtype". Als Beispiel wäre "text/xml", das für das XML Datenaustauschformat steht. Die Internet Assigned Numbers Authority (IANA) gilt als zentrale Zulassung der verschiedenen MIME media types und bietet auch eine Liste aller existierenden (vgl. Freed, Innosoft und Borenstein 1996). Der bereits zuvor erwähnte Content-Type-*Header* wurde wie bereits erwähnt von HTTP geerbt und beschreibt den media type des *entity body*. Sodass z.B. eine Antwort eines Servers ein Content-Type-*Header* mit "image/jpeg" beinhaltet. Der Client weiß mithilfe dieser Angabe das es sich bei dem *entity body* der Antwort um ein Bild im JPEG-Format handelt (vgl. Richardson und Ruby 2007, S. 7). Es existieren viele verschiedene media types. In diesem Abschnitt werden nur die erklärt, welche auch in dieser Arbeit benutzt werden.

2.2.1 XML

XML beschreibt eine Klasse von Daten, auch XML-Dokumente genannt und zum Teil das Vorgehen wie Computerprogramme diese verarbeiten. XML-Dokumente sind zusammengesetzt aus Einheiten die entities genannt werden. Diese beinhalten geparste oder ungeparste Daten. Die geparsten Daten sind aus Zeichen zusammengesetzt die entweder Informationen über das XML-Dokument oder Charakter Daten, auch als CDATA bezeichnet, enthalten (vgl. Bray u. a. 2008). XML-Dokumente sind strukturiert und wohlgeformt. Zudem ist XML eine Teilmenge von der Standard Generalized Markup Language (SGML). Es gibt verschiedene MIME media types für XML. In dieser Arbeit wird folgender benutzt: `text/xml` (vgl. Murata u. a. 2001).

2.2.2 XHTML

XHTML ist unter einigen Auflagen, laut Richardson und Ruby, HyperText Markup Language (HTML). Die Auflagen ermöglichen das jedes XHTML-Dokument ein valides XML-Dokument ist. Mithilfe dieser Eigenschaft ist es möglich ein XHTML-Dokument mit einem XML-Parser zu verarbeiten. Der MIME media type für XHTML lautet: `application/xhtml+xml` (vgl. Richardson und Ruby 2007, S. 259 f.).

2.2.3 JSON

JSON ist ein leichtgewichtiges, text basierendes und sprachunabhängiges Datenaustauschformat. Es ist ein Textformat um strukturierte Informationen zu serialisieren. Der MIME media type für JSON lautet: `application/json` (vgl. Crockford 2006). Es ist leichtgewichtiger und besser lesbar als ein identisches XML-Dokument (vgl. Richardson und Ruby 2007, S. 266).

2.2.4 Form-Encoded Key-Value Paare

Dieses Datenaustauschformat wird laut Richardson und Ruby zu meist für Repräsentationen, die vom Client zum Server gesendet werden, benutzt. Es ist auch der default media type einer HTML form, mit der die Parameter via POST an den Server gesendet werden. Der MIME media type lautet: `application/x-www-form-urlencoded` und er besteht nur aus einfachen *key-value* Paaren in dieser Form: key=value (vgl. ebd., S. 266).

2.3 REST

> »REST consists of a set of architectural constraints chosen for the properties they induce on candidate architectures«
>
> Fielding 2000, S. 85

Wie zuvor schon in dem Abschnitt 2.1 und auch aus dem Zitat 2.3 herauszulesen ist, handelt es sich bei REST um eine Sammlung von Architektur-Kriterien.

Laut Richardson und Ruby ist der Begriff *RESTful* ähnlich dem Begriff objektorientiert. Wird ein Programm in einer objektorientierten Sprache und unter deren Bedingungen umgesetzt, spricht man von einem objektorientierten Programm, ähnlich wie bei dem Begriff *RESTful*. Wird ein Webservice nach den REST-Bedingungen implementiert so spricht man von einem *RESTful* Webservice. Kurz gesagt werden die REST-Kriterien erfüllt spricht man von *RESTful* (vgl. Richardson und Ruby 2007, Preface: S. XX f.).

2.3.1 REST-Bedingungen

Bedingung	**Erklärung**
Client-Server	Es werden Komponenten/Funktionalitäten von einem Server bereitgestellt, die ein Client nutzen kann via Anfragen.
Stateless	Jede Anfrage muss alle nötigen Information beinhalten.
Cache	Jede Antwort muss markiert sein ob sie *cachable* ist oder nicht.
Uniform interface	Einheitliche Schnittstelle (wie z.B. HTTP)
Identification of resources	Jede Ressource muss eindeutig identifizierbar gemacht werden (z.B. mithilfe einer URI).
Manipulation of resources through representations	Die Ressourcen können via Repräsentationen von sich selbst verändert werden (z.B. mit XML, JSON etc.).
Self-descriptive messages	Beschreibung der Nachricht (als Beispiel HTTP-*Header* die den Content oder anderes beschreiben)
Hypermedia as the engine of application state	Hypermedia-Auslieferung bei z.B. GET-Anfragen (Link's auf andere Ressourcen oder z.B. HTML Seiten)
layered system	Die Implementierung der Ressourcen sind für den Client nicht sichtbar.
code-on-demand [optional]	Code downloaden, erweitern und ausführen

Tabelle 2.1.: REST-Bedingungen

Die folgende Beschreibung basiert auf der Definition von Fielding und dient zur Erklärung der Bedingungen von REST aus der Tabelle 2.1 (vgl. Fielding 2000, S. 77 ff.). Die erste Bedingung für REST ist das Client-Server Model. Dieses Model gehört zu den meist benutzten Architekturstilen für netzwerkbasierte Applikationen. Ein Server stellt bestimmte Funktionalitäten bereit und wartet auf Anfragen. Ein Client der eine bestimmte Funktionalität benutzen möchte, sendet dem Server eine Anfrage. Der Server kann dann diese Anfrage ablehnen oder beantworten.

Normalerweise ist ein Server ein nicht endender Prozess und stellt oft Dienstleistungen für mehrere Clients zur Verfügung (vgl. Fielding 2000, S. 45 f.). Durch das Aufteilen des user interface auf dem Client und der Datenspeicherung auf dem Server, wird die Portierbarkeit des Clients und die Skalierbarkeit des Servers erhöht (vgl. ebd., S. 78). Eine weitere Bedingung ist die Zustandslosigkeit. Das bedeutet das jede Anfrage an den Server alle nötigen Informationen beinhalten muss. Ansonsten kann diese nicht korrekt beantwortet werden. Somit verbleibt das "session-handling" beim Client und die Skalierbarkeit wird damit erhöht (vgl. ebd., S. 79). Die *cache* Bedingung erfordert das jede Antwort als *cacheable* oder *non-cachable* gekennzeichnet wird. Das bedeutet wenn eine Antwort *cacheable* ist, dann kann der Client die Daten speichern und später noch einmal wiederverwenden (wenn z.B. gleiche Anfragen anstehen.). Der Vorteil dabei ist, dass manche Interaktionen zwischen dem Client und dem Server eingespart werden können. Somit wird die Effizienz erhöht (vgl. ebd., S. 79 f.). Eine Hauptbedingung von REST ist die einheitliche Schnittstelle (*uniform interface*). Dies verringert jedoch die Effizienz etwas, da der Datenaustausch dann über standardisierte Wege stattfindet und nicht auf applikationsspezifische, die darauf zugeschnitten sind. Damit dieses *uniform interface* gewährleistet werden kann, wurden vier weitere Bedingungen definiert (vgl. ebd., S. 81 f.). Diese vier Bedinungen lauten: Identifizierung von Ressourcen, Manipulation von Ressourcen via Repräsentationen, selbsterklärende Nachrichten und Hypermedia als Motor (engl. engine) für den Applikationszustand. Die Erklärung für die ersten drei Schnittstellenbedingungen folgt im Abschnitt 2.3.4. Die Erklärung bzw. Umsetzung der Bedingung "Hypermedia as the engine of application state" ist dem Abschnitt 2.4.5 zu entnehmen. Die Bedingung *layered system* erhöht die Evolvier- und Wiederverwendbarkeit eines Servers, dadurch das einzelne Kompenenten in hierarchischen Schichten unterteilt werden. Ein *layered system* reduziert den Verbund von mehreren *layers*/Schichten dadurch, dass die inneren *layers* verborgen werden. Mit Ausnahme der angrenzenden, äußeren *layers*. Die inneren *layers* stellen für die äußeren *layeres* die Funktionalitäten bereit. Der Hauptnachteil solcher *layered systems* ist der damit verbundene *overhead* und die Latenz zur Verarbeitung von Daten (vgl. ebd., S. 46). Diese *layers* können benutzt werden um veraltete Dienste einzukapseln und neue Dienste vor veralteten Clients zu schützen. Um das zu gewährleisten werden Funktionalitäten die nur selten im Gebrauch sind an einen verteilten Vermittler verschoben. Dieser sogenannte Vermittler/Proxy kann auch die Skalierbarkeit erhöhen in dem man das load balancing ermöglicht (vgl. ebd., S. 83). Die einzige, optionale Bedingung von REST ist die code-on-demand Bedingung. Diese besagt, dass REST dem Client erlaubt Funktionalitäten zu erweitern, in dem man den Code in der Form von Applets oder Skripten downloaded und ausführt. Dadurch wird die Systemerweiterbarkeit erhöht. Diese Bedingung ist optional, da das sogenannte code-on-demand nicht immer gewährleistet werden kann (vgl. ebd., S. 84).

2.3.2 REST-Konnektor

REST benutzt verschiedene Konnektortypen um das Zugreifen auf Ressourcen und übertragen von Ressource Repräsentationen (engl. representations) ein zukapseln. Diese Konnektoren bilden eine abstrakte Schnittstelle für die Komponentenkommunikation. Sie erhöhen die Einfachheit durch die klare Trennung von Aufgaben und verstecken die Implementierung von Ressourcen und die Kommunikationsmechanismen (vgl. Fielding 2000, S. 92 f.). Für einen Konnektor ist es erforderlich das jede Anfrage alle nötigen Informationen beinhaltet, damit dieser die Anfrage versteht. Egal ob vor dieser Anfrage eine gesendet wurde. Die primären Konnektortypen sind Client und Server. Wobei der Client die Kommunikation beginnt und der Server auf Verbindungen horcht und auf Anfragen antwortet. Eine dritter Konnektortyp ist der *cache*, er kann an der Schnittstelle der Clients oder Servers platziert werden, sodass Antworten die *cachable* sind gespeichert werden können. Diese gespeicherten Antworten können später wiederverwendet werden. Diese Eigenschaft ermöglicht das Vermeiden von wiederholten Netzwerkkommunikationen für den Client oder wiederholtes Generieren von Antworten, was in beiden Fällen die Interaktionslatenz vermindert (vgl. ebd., S. 93 f.). Ein Konnektor kann die *cacheability* einer Antwort mithilfe von *control data* bestimmen. Mit dieser control data kann eine Antwort als *cachable*, *non-cachable*, oder *cachable* für eine bestimmte Zeit, markiert werden. Der *resolver* ist ein weiterer Konnektortyp. Dieser übersetzt teilweise oder komplett den *resource identifier* in eine Netzwerkadresse um eine Verbindungen zwischen den Komponenten aufzubauen. Der letzte Konnektortyp ist der *tunnel* der die Kommunikation zwischen z.B. firewalls oder lower-level Netzwerk gateways ermöglicht. Dieser Konnektortyp ermöglicht den Komponenten das dynamische switchen zwischen aktiven Komponentenverhalten zu einem *tunnel* Verhalten (vgl. ebd., S. 95 f.).

2.3.3 REST-Komponenten

In REST gibt es vier verschiedene Komponenten: *origin server*, *gateway*, *proxy* und *user agent*. Der *user agent* ist ein Clientkonnektor der eine Anfrage ausführt und dafür vom *origin server* eine Antwort erhält. Am weit verbreitetsten ist als Client der Webbrowser, der bestimmte Services nach Informationen fragt und deren Antworten rendert bzw. darstellt (vgl. ebd., S. 96). Der sogenannte *origin server* ist nichts anderes als der Serverkonnektor, der auf Anfragen von einem Client (*user agent*) wartet, die zum Beispiel den Zustand der Ressourcen via Repräsentationen ändern (siehe Abschnitt 2.3.4). Der *origin server* beantwortet diese Anfragen. Jeder *origin server* besitzt eine generische Schnittstelle für seine Ressourcen Hierarchie. Die Implementierungen der Ressourcen sind somit hinter der Schnittstelle verborgen.

Der *proxy* wird spezifisch vom Client eingesetzt um andere Services einzukapseln, Daten zu konvertieren, Performance zu erhöhen oder zur Sicherheit. Der *gateway* dagegen wird vom Netzwek oder vom *origin server* eingesetzt um die zuvor genannten Eigenschaften zu gewährleisten (vgl. Fielding 2000, S. 97).

2.3.4 REST spezifische Elemente

Es gibt laut Fielding folgende REST spezifische Elemente (oder auch data elements genannt): *resource*, *resource identifier*, *representation*, *representation metadata*, *resource metadata* und *control data*. Diese Elemente sind wichtige Bestandteile von REST. Die Ressource (engl. *resource*) ist die Abstraktion einer Information. Jede Information die benannt werden kann ist eine Ressource: ein Dokument, ein Bild, eine temporäre Dienstleistung, eine Liste von anderen Ressourcen usw. (vgl. ebd., S. 88). Kurz gesagt, eine Ressource ist alles was wichtig genug ist um darauf zu referenzieren (vgl. Richardson und Ruby 2007, S. 83). Der *resource identifier* wird gebraucht um jede Ressource eindeutig zu identifizieren (vgl. Fielding 2000, S. 90). Mithilfe des *resource identifier* wird die Bedingung "Identification of resources" abgehandelt. Es ist aber durchaus möglich das mehrere *resource identifier* auf die selbe Ressource verweisen. Als Beispiel hat Fielding ein Versions-Kontroll Programm aus der Softwareentwicklung genannt. Da dort z.B. die "last revision" und die "revision number 1.2.7" auf die selbe Änderung verweisen (vgl. ebd., S. 89). Eine Repräsentation (engl. *representation*) ist eine Sequenz von Bytes plus die *representation metadata*, welche diese Bytes beschreibt. Sogenannte REST-Komponenten (siehe Abschnitt 2.3.3) können Handlungen an Ressourcen mithilfe von Repräsentationen ausführen und somit den Zustand der Ressourcen verändern oder an andere Komponenten den derzeitigen Zustand mithilfe von Repräsentationen liefern (vgl. ebd., S. 90 und vgl. Richardson und Ruby 2007, S. 93). Dies entspricht der REST-Bedingung "Manipulation of resources through representations". Fielding definiert Metadaten (engl. *metadata*) als Form von name-value Paaren (ähnlich wie *key-value* Paare), wobei die Namen standardisiert sind und somit die Wertestruktur und -semantik definieren. Eine Antwort kann als Nachricht die Repräsentationsmetadaten sowie die Ressourcemetadaten enthalten. ("Self-descriptive messages") Wobei die Ressourcemetadaten Informationen über die Ressource enthält die unabängig von der Repräsentation ist. Die sogenannte Kontrollinformation (engl. *control data*) beschreibt das Vorhaben/Sinn einer Nachricht zwischen den REST Komponenten. Es ist damit auch möglich das Cacheverhalten zu verändern, in dem es in der Antwort oder der Anfrage mit eingebunden wird. Innerhalb der Ressourcemetadaten kann auch das Datenformat der Repräsentation angegeben werden. Dieses ist auch als media type bekannt, siehe Abschnitt 2.2 (vgl. Fielding 2000, S. 91).

2.3.5 Richardson Maturity Model

Das "Richardson Maturity Model" wurde von Leonard Richardson entwickelt und teilt die REST-Bedingungen (siehe Abschnitt 2.3.1) in vier Level auf. Laut Fielding ist Level drei eine Vorbedingung für REST. Das Model an sich ist nur zur Veranschaulichung welche Elemete innerhalb REST existieren. Es ist daher keine Definition von Leveln von REST. Mithilfe des Models kann Schritt für Schritt verstanden werden welche Ideen sich hinter REST bzw. *RESTful* verbergen. (vgl. Fowler 2013, The Meaning of the Levels).

Level null

Level null beinhaltet nur die Benutzung eines Transportprotokolls (z.B HTTP) für entfernte Interaktionen. In diesem Level wird das Transportprotokoll sozusagen als Tunnel-Mechanismus benutzt um entfernte Interaktionen durchzuführen, wie es bei RPC der Fall ist. SOAP und RPC sind auf diesem Level null (vgl. ebd., Level 0). Wird HTTP als Transportprotokoll in Level null benutzt, so werden keine Eigenschaften innerhalb dessen genutzt. Es wird sozusagen nur eine Methode und eine URI als Endpunkt benutzt (vgl. Thijssen 2013, Level 0).

Level eins

Das Level eins beinhaltet dann die Unterstüzung von Ressourcen, sodass die Kommunikation sich nicht mehr auf einen Endpunkt konzentriert, sondern auf Ressourcen aufgeteilt wird (vgl. Fowler 2013, Level 1). Auf diesem Level wird aber weiterhin nur eine Methode benutzt z.B. POST (vgl. Thijssen 2013, Level 1).

Level zwei

Innerhalb des nächsten Levels (Level zwei) werden die HTTP Eigenschaften, so wie es in HTTP definiert ist, benutzt. D.h. GET zum Abfragen von Informationen, DELETE zum Löschen und Antworten auf fehlerhafte Anfragen, beinhalten andere HTTP-Statuscodes als erfolgreiche Anfragen usw. (vgl. Fowler 2013, Level 2). Das Problem bei diesem Level ist die Beschränkung auf HTTP, da REST auch mit anderen Transportprotokollen funktioniert. Denn REST ist unabhängig davon, welches Transportprotokoll benutzt wird. Um es allgemein auch für andere Protokolle zu definieren, muss gesagt werden, dass in dem Level zwei die Eigenschaften des Transportprotokolls voll ausgenutzt werden sollten (vgl. Thijssen 2013, Level 2).

Level drei

Das letzte Level (Level drei) beinhaltet die Hypermedia as the engine of application state (HATEOAS) Bedingung, siehe dazu auch Abschnitt 2.3.1.

Durch Hypermedia ist es möglich dem Client mitzuteilen wie die URI der Ressource aussieht, die verändert oder abgefragt wer kann. Der Vorteil dabei ist auch das die URIs im Nachhinein geändert werden können ohne das die Clients somit ausgesperrt werden. Da mithilfe von Hypermedia diese neuen URIs dann auch ausgeliefert werden (vgl. Fowler 2013, Level 3).

2.3.6 Zusammenfassung

REST liefert einige Architekturkriterien, die wenn sie erfüllt werden, die Skalierbarkeit von Komponenten Interaktionen, Schnittstellen Allgemeinheiten und die Unabhängigkeiten von eingesetzten Komponenten hervorheben. Zudem können die vermittelten Komponenten (proxies, gateways) Interaktionslatenzen vermindern, die Sicherheit durchsetzen und veraltete Systeme einkapseln (vgl. Fielding 2000, S. 105). Laut Fielding wurde REST entwickelt weil für das Web ein Model benötigt wurde, welches das Aussehen bzw. das Arbeiten des Webs, wie es sein sollte, beschreibt. REST ist eine koordinierte Sammlung von Architektur-Kriterien die versuchen die Latenz und Netzwerkkommunikation zu minimieren und gleichzeitig die Unabhängigkeit und Skalierbarkeit von Komponenten zu maximieren. REST ermöglicht das *caching* und die Wiederverwendung von Interaktionen, die dynamische Ersetzbarkeit von Komponenten und Verarbeitung von Aktionen durch Vermittler (proxy,gateway). REST deckt dabei den Bedarf von Internet skalierten verteilten Hypermedia-Systemen ab (vgl. ebd., S. 148). Zusammenfassend ist zu sagen das eine REST-konforme Architektur alle Bedingungen aus Abschnitt 2.3.1 erfüllen muss. Des Weiteren muss eine REST-konforme Architektur alle Komponenten aus Abschnitt 2.3.3 und Elemente aus Abschnitt 2.3.4 zusammenfassen und unterstützen. Folgendes Zitat von Richardson und Ruby beschreibt im kurzem die Namensherkunft von REST:

> »The client manipulates resource state by sending a representation as part of PUT or POST request. (DELETE requests work the same way, but there's no representation.) The server manipulates client state by sending Repräsentationen in response to the client's GET requests. This is where the name "Representational State Transfer" comes from.«
>
> Richardson und Ruby 2007, S. 218

Ein Beispiel für eine REST-konforme bzw. *RESTful* Webservice-Architektur und deren Anwendung folgt in der Abschnitt 2.4.

2.4 ROA

ROA beschreibt eine *RESTful* Webservice-Architektur. Da REST sehr allgemein und nicht direkt an das Web gebunden ist, wurde von Richardson und Ruby diese Architektur entwickelt, um eine REST-konforme Webservice-Architektur direkt an die Technologien des Web's zu binden. Denn REST hängt nicht von den Mechanismen HTTPs oder der Struktur von URIs ab. ROA benutzt die Technologien des Web's wie z.B. HTTP oder URI unter den Bedingungen von REST. ROA bringt vier Eigenschaften mit sich: Adressierbarkeit ("addressability"), Zustandslosigkeit ("statelessness"), Verbundenheit ("connectedness") und eine einheitliche Schnittstelle (*uniform interface*). Mithilfe der Web Technologien werden diese umgesetzt. Einige dieser Eigenschaften basieren auf den Bedingungen von REST und wurden bereits in dem Abschnitt 2.3.1 beschrieben. Sie wurden jedoch noch etwas erweitert (vgl. Richardson und Ruby 2007, S. 81 ff.). Bevor jedoch die Eigenschaften von ROA erklärt werden, müssen erst die wichtigen Elemente Ressource und Repräsentation, die schon in dem Abschnitt 2.3.4 innerhalb REST definiert wurden, genau erläutert werden.

2.4.1 Ressourcen

Die Definition einer Ressource gleicht dem in dem Abschnitt 2.3.4 beschriebenen. Mit dem Unterschied das nun die Verwendung von URIs mit einbezogen werden. Laut Richardson und Ruby hat jede Ressource mindestens eine URI. Im Grunde ist sozusagen eine URI ein *resource identifier* (siehe Abschnitt 2.3.4). Diese URI ist der Name und die Adresse der Ressource. Hat ein Teil einer Information keine URI, so ist dieser Teil auch keine Ressource und somit auch nicht innerhalb des Web's. Der Sinn warum URIs als *resource identifier* eingesetzt werden ist ziemlich simpel. Es soll jedem möglich sein eine URI weiterzugeben und somit den Zugriff auf eine bestimmte Ressource für andere zu gewährleisten. Dies beschreibt das Prinzip der Adressierbarkeit (siehe Abschnitt 2.4.3). Die URI einer Ressource sollte selbsterklärend sein und eine gewisse Struktur besitzen. Dies ist zwar keine Bedingung aber zeugt laut Richardson und Ruby von gutem Webdesign. Denn dadurch kann ein Client, falls er den Aufbau dieser URIs kennt, sich eigene Eintrittspunkte verschaffen. Das macht es für Clients einfacher den Service zu nutzen (vgl. ebd., S. 84 f.).

2.4.2 Repräsentation

Wie auch schon bei den Ressourcen der ROA in Abschnitt 2.4.1 ist die Definition der Repräsentation einer Ressource gleich dem, was in dem Abschnitt 2.3.3 beschrieben wurde. In diesem Abschnitt wird die Repräsentation einer Ressource weiter detailliert. Eine Repräsentation ist die Information über den derzeitigen *resource state* (engl. Ressourcenzustand siehe Abschnitt 2.4.4) einer Ressource (vgl. Richardson und Ruby 2007, S. 93). Es ist möglich für eine Ressource verschiedene Repräsentationen zu erhalten. Ein Beispiel wäre wenn eine Ressource einmal eine XML-Repräsentation ausliefert und einmal eine in JSON Format, abhängig von der Anfrage. Ein Client kann auch eine Repräsentation an einen Webservice senden um den *resource state* zu verändern. Durch das Senden einer Repräsentation einer Ressource an den Webservice, kann der Client entweder eine neue Ressource auf dem Server erstellen oder eine bestehende Ressource verändern (vgl. ebd., S. 94). Damit man von einer Ressource verschiedene Repräsentationen erhalten kann, ist es möglich den HTTP-*Header* Accept auf verschiedenen media types zu setzen. Richardson und Ruby empfehlen jeder Art von Repräsentation eine eigene URI zu geben. Der Grund dahinter ist das eine URI schnell von Person zu Person (oder Client zu Client) weitergegeben werden kann. Somit muss nicht erst ein *Header* gesetzt werden um die bestimmte Repräsentation zu erlangen. Denn wenn eine URI weitergegeben wird, gehen die Metadaten meist verloren.

2.4.3 Adressierbarkeit

Die Adressierbarkeit ist einer der vier wichtigen Eigenschaften von ROA. Eine Applikation ist adressierbar wenn sie ihre Daten und Informationen in Ressourcen abbildet. Dadurch das jede Ressource mindestens eine URI besitzt, hat eine adressierbare Applikation für jede Information eine URI. Für den Endbenutzer oder Client ist die Adressierbarkeit der wichtigeste Aspekt einer Webseite oder Applikation. Ein Beispiel für die Adressierbarkeit wäre folgende URI: ”https://www.google.de/search?q=rest”. Wäre HTTP oder die Google Suchmaschine nicht adressierbar, wäre es nicht möglich diese URI zu verbreiten. Man müsste eine genaue Beschreibung der Google Suche und des Begriffes geben um das selbe Ergebnis zu ermöglichen. Das vollkommene Gegenteil dazu sind Ajax-Applikationen. Diese sind meist nur Webservice Clients. Dadurch das bei Ajax-Applikationen meist Teile der Webseiten nachgeladen bzw. verändert werden, sind sie nicht adressierbar. Würde man eine URI von einer Webseite weitergeben, die Ajax benutzt, so würde derjenige der diese URI z.B. im Webbrowser aufruft, wahrscheinlich nicht das sehen weswegen sie eigentlich geteilt wurde. Der Benutzer würde sozusagen wieder am Anfang der Applikation stehen (vgl. ebd., S. 87 ff.).

2.4.4 Zustandslosigkeit

Eine zweite Eigenschaft der ROA ist die Zustandslosigkeit. Diese ist mit der REST-Bedingung "statelessness" gleichzusetzen, die in dem Abschnitt 2.3.1 beschrieben ist. Laut Richardson und Ruby bedeutet die Zustandslosigkeit das auch alle möglichen Zustände des Servers Ressourcen sind und somit ihre eigene URI besitzen. Somit muss der Client den Server nicht erst dazu bringen auf eine bestimmte Anfrage zu hören. Als Beispiel dafür wäre eine Desktop-Applikation. Diese ist meist nicht zustandslos. Will man z.B. auf ein Screen C kommen muss man erst über Screen A und B. Bei einem zustandlosen Webservice oder einer Applikation kann man direkt auf Screen C zugreifen (vgl. Richardson und Ruby 2007, S. 89). Diese Zustandslosigkeit macht es einfacher einen Webservice auf mehrere last ausgeglichenen ("load balanced") Servern zu verteilen, da kein Anfrage von einer anderen abhängt. Da HTTP ein zustandsloses Protokoll ist, ist die Zustandslosigkeit eines Webservices, wenn dieser HTTP benutzt, eine vorgegebene Eigenschaft. Laut Richardson und Ruby muss man erst etwas unternehmen um einen nicht zustandslosen Webservice zu schreiben (vgl. ebd., S. 91). Zustandslosigkeit bzw. *statelessness* besagt das es nur einen Zustand gibt und dieser sollte nicht auf dem Server liegen. Doch nach Richardson und Ruby gibt es zwei Zustände. Einmal den *application state*, welcher auf dem Client liegen sollte und den *resource state*, welcher auf dem Server liegt. Das bedeutet wenn ein Client eine Anfrage an einen Webservice sendet, schickt er alle nötigen *application states* die der Server braucht um die Anfrage zu beantworten. Danach "vergisst" der Webservice alles über den Client und seinen derzeitigen *application state*. Das ist für Richardson und Ruby *stateless*. Der *application state* ist somit vom Client abhängig. Jeder Client besitzt ein anderen Zustand. Der *resource state* jedoch ist für jeden Client gleich, da dieser auf dem Server liegt. Als Beispiel wäre denkbar das ein Bild oder eine Information an einen Webservice gesendet und dafür eine neue Ressource erstellt wird. Die neue Ressource ist dann für jeden Client über eine URI erreichbar (vgl. ebd., S. 92 f.).

2.4.5 Verbundenheit

ROA hat als weitere wichtige Eigenschaft die Verbundenheit. Mit Verbundenheit oder engl. *connectedness* ist gemeint das die Repräsentationen in einem Hypermediaformat geliefert werden. Also Dokumente die nicht nur Informationen enthalten, sondern auch Links (Verweise) zu anderen Ressourcen. Mit diesen Links kann der Client auf andere Ressourcen aufmerksam gemacht und weitergeleitet werden. Dies stammt schon aus der REST-Bedingung "Hypermedia as the engine of application state" (siehe Abschnitt 2.3.1).

Laut Richardson und Ruby ist die Bedingung wie folgt zu deuten: Der derzeitige Zustand der HTTP "session" ist nicht als *resource state* auf dem Server gespeichert sondern beim Client als *application state*. Dieser beinhaltet den derzeitigen Pfad im Web. Der Server leitet anhand von Repräsentationen im Hypertextformat den Client durch das Web. Diese beinhalten Hyperlinks und Formulare (vgl. Richardson und Ruby 2007, S. 96 f.). Der Server zeigt dem Client einen Weg von dem derzeitigen Zustand zu einem ähnlichen. Als Beispiel wäre ein Link der auf eine Ressource verweist die ähnliche Information oder weitere Ergebnissen zu einer Anfrage beinhaltet. Diese Eigenschaft der Rückgabe von Links und der Weiterleitung des Clients zu anderen Ressourcen, wird von Richardson und Ruby als *connectedness* bezeichnet (vgl. ebd., S. 98).

2.4.6 Einheitliche Schnittstelle

Eine weitere Eigenschaft von ROA ist die einheitliche Schnittstelle (engl. *uniform interface*) welche, mit der gleichnamigen Bedingung aus REST, dem *uniform interface*, einhergeht (siehe Abschnitt 2.3.1). Das *uniform interface* der ROA wird mithilfe von HTTP umgesetzt, da HTTP bereits ein *uniform interface* besitzt. Die HTTP-Methoden (siehe Tabelle 2.2) stellen das *uniform interface* der ROA dar und werden benutzt um mit einem ROA-Webservice zu kommunizieren. (vgl. ebd., S. 99)

HTTP-Methode	**Beschreibung**
GET	GET an einer existierenden Ressource liefert deren Repräsentation
POST	POST an einer existierenden Ressource, erstellt eine neue untergeordnete Ressource; POST an einer existierende Ressource hängt Informationen an
PUT	PUT an einer neuen URI erstellt eine Ressource; PUT an einer existierenden Ressource verändert diese
DELETE	DELETE an einer existierende Ressource löscht diese
HEAD	HEAD an einer existierende Ressource liefert deren Metadaten
OPTIONS	OPTIONS an einer existierende Ressource liefert die unterstützen HTTP-Methoden der Ressource

Tabelle 2.2.: HTTP-Methoden
vgl. ebd., S. 99 ff.

In den folgenden Abschnitten wird das *uniform interface* der ROA bzw. die HTTP-Methoden mithilfe von ebd., S. 99 ff. erklärt. Die HTTP-Methoden TRACE und Connect werden nicht mit ROA benutzt und deshalb in diesen Abschnitten auch nicht erklärt.

In der Tabelle 2.2 sind alle HTTP-Methoden, die innerhalb der ROA benutzt werden, zusammengefasst und kurz erklärt. Eine Zusammenfassung der verwendeten HTTP-*Header* und Statuscodes, innerhalb dieser Arbeit, befindet sich im Anhang. Siehe dazu die Tabelle A.2 für die HTTP-*Header* und die Tabelle A.1 für die verwendeten HTTP-Statuscodes.

GET

Die GET-Methode wird benutzt um eine Repräsentation einer Ressource zu erhalten. Die Repräsentation wird auf eine GET-Anfrage innerhalb des *entity body* der Antwort versendet (vgl. Richardson und Ruby 2007, S. 99). Eine Form von GET ist die conditional GET-Methode. Diese Methode wird mit den HTTP-Antwort-*Header* Last-Modified und/oder Etag und den Anfrage-*Headers* If-Modified-Since und/oder If-None-Match bewerkstelligt. Diese Form der GET-Methode erspart dem Server und Client Zeit und Bandbreite. Die Methode wird wie folgt bewerkstelligt: der Server sendet mit der Repräsentation den HTTP-*Header* Last-Modified, der die Zeit enthält wann die Repräsentation das letzte mal geändert wurde. Die angegebene Zeit kann vom Client gespeichert und bei der nächsten GET-Anfrage per If-Modified-Since zum Server gesendet werden. Empfängt der Server diese Anfrage mit dem If-Modified-Since-*Header*, wird die Zeit mit der letzten Änderung geprüft. Wurde seitdem eine Änderung an der Repräsentation vorgenommen, so sendet der Server als Antwort die neue Repräsentation und den HTTP-Statuscode 200 ("OK"). Wurde die Repräsentation aber seitdem nicht geändert, antwortet der Server mit dem HTTP-Statuscode 304 ("Not Modified"). Die Repräsentation wird dabei nicht gesendet. Somit weiß der Client das er die alte Repräsentation weiter benutzen kann, da diese immer noch aktuell ist (vgl. ebd., S. 140 f.). Der Unterschied zu dem Etag-*Header* (steht für *entity tag*) ist das dieser *Header* ein String beinhaltet, der immer verändert wird wenn die Repräsentation sich ändert. Der Etag kann z.B. den MD5 *hash* der Repräsentation beinhalten. Ähnlich wie bei dem Last-Modified-*Header* wird der Etag-*Header* mit der Antwort gesendet. Der Client kann diesen String speichern und bei der nächsten Anfrage als If-None-Match-*Header* senden. Stimmt z.B. der *hash* nicht mehr mit dem *hash* der Repräsentation überein, so wird, wie bei dem If-Modified-Since-*Header*, die Repräsentation und ein HTTP-Statuscode 200 als Antwort gesendet. Stimmen die *hashs* überein, wird der HTTP-Statuscode 304 als Antwort ohne Repräsentation gesendet. Der Etag- und der Last-Modified-*Header* sind miteinander kombinierbar. Der Etag-*Header* wurde eingeführt da die Zeit innerhalb des Last-Modified-*Header* nicht immer akkurat genug gewesen ist. Laut Richardson und Ruby sollte ein Server, der beide *Header* unterstützt, nur dann eine Repräsentation senden, wenn die Repräsentation und der Etag sich geändert haben (vgl. ebd., S. 244 ff.).

DELETE

Die DELETE-Methode ermöglicht es eine existierende Ressource zu löschen. Der *entity body* der Antwort auf eine DELETE-Anfrage kann entweder eine Statusnachricht oder gar nichts enthalten (vgl. ebd., S. 99).

PUT

Um eine Ressource zu erstellen oder zu verändern, sendet der Client eine PUT-Anfrage. Diese Anfrage enthält in den meisten Fällen eine neue Repräsentation der Ressource, innerhalb des *entity body*. Dies ist aber keine Bedingung für PUT. Sendet der Client eine PUT-Anfrage, ist dies der Punkt an dem der *application state* zum *resource state* wird. Denn mithilfe der Repräsentation oder der Scoping-Information, innerhalb der URI, wird eine neue Ressource erstellt oder eine alte ersetzt/verändert (vgl. Richardson und Ruby 2007, S. 99 f.).

HEAD

Als weitere HTTP-Methode wird HEAD innerhalb des *uniform interface* von ROA benutzt. Durch eine HEAD-Anfrage kann ein Client nur die Metadaten einer Ressource abfragen. Diese Anfrage liefert als Antwort das selbe Ergebnis wie eine GET-Anfrage, bis auf den *entity body*, dieser bleibt leer. HEAD kann auch dazu benutzt werden um herauszufinden ob eine Ressource überhaupt existiert (vgl. ebd., S. 100).

OPTIONS

Die OPTIONS-Methode lässt den Client herausfinden welche HTTP-Methoden eine Ressource unterstützt. Die Antwort auf eine OPTIONS-Anfrage beinhaltet ein HTTP-*Header* namens Allow. Eine Beispiel Antwort auf eine OPTIONS-Anfrage wäre folgender Allow-*Header*: Allow: OPTIONS, GET, HEAD. Dieser Allow-*Header* bedeutet das nur die Methoden GET, HEAD und OPTIONS auf die Ressource angewendet werden können. Ansonsten wird der HTTP-Statuscode 405 zurückgesendet. Der Statuscode 405 besagt "Method Not Allowed", also das die versuchte HTTP-Methode nicht an der Ressource angewandt werden kann/darf. Die Ressource aus dem Beispiel ist somit read-only, man hat also nur lesenden Zugriff. Falls die Ressource über eine Autorisierung geschützt ist und man ein OPTIONS-Anfrage ohne Authorization-*Header* ausführt, dann werden nur die HTTP-Methoden zurückgegeben, die für den unautorisierten Zugriff auch erlaubt sind. Würde man die OPTIONS-Anfrage aus dem vorherigen Beispiel noch einmal mit Authorization-*Header* ausführen, wäre es möglich das der Allow-*Header* innerhalb der Antwort folgenden Inhalt besitzt: Allow: OPTIONS, GET, HEAD, POST, PUT. Somit lässt sich mithilfe einer OPTIONS-Anfrage die Zugriffskontrolle überprüfen (vgl. ebd., S. 101).

POST

Die letzte HTTP-Methode die innerhalb des *uniform interface* von ROA benutzt wird ist POST. POST ist laut Richardson und Ruby die meist missverstandenste Methode. In einem *RESTful* Design wird die POST-Methode benutzt um untergeordnete Ressourcen zu erstellen. Eine untergeordnete Ressource ist eine Ressource die in einer Beziehung zu einer "Eltern" Ressource existiert. Als Beispiel wäre eine Ressource namens "Tagebuch". Eine untergeordnete Ressource wäre dann in dem Fall ein "Tagebucheintrag".

Um einen "Tagebucheintrag" zu erstellen müsste man eine POST-Anfrage an die "Eltern" Ressource "Tagebuch" senden (POST an example.de/tagebuch). Die POST-Anfrage kann wie bei der PUT-Anfrage eine Repräsentation enthalten. Auch hier tritt wieder der Fall ein das der *application state* zum *resource state* wird. Die Antwort auf solch eine POST-Anfrage beinhaltet meist ein HTTP-Statuscode von 201 ("Created") und einen HTTP-*Header* namens Location. Der Location-*Header* beinhaltet dann die URI zur neu erstellten, untergeordneten Ressource. In unserem Beispiel würde der Location-*Header* wie folgt aussehen: Location: example.de/tagebuch/1. Die erstellte, untergeordnete Ressource "Tagebucheintrag" ist nun über "example.de/tagebuch/1" erreichbar. Die zweite Benutzung von POST ist nicht das Erstellen von Ressourcen, sondern das Anhängen von Information an bestehende Ressourcen. Mithilfe von POST an die Ressource "/tagebuch/1" wäre es z.B. möglich den Tagebucheintrag mit der Identifikationsnummer (ID) 1 zu erweitern. Somit wird mit der gesendeten Information keine untergeordnete Ressource erstellt, sondern diese Information wird an die "Eltern" Repräsentation gehängt (vgl. Richardson und Ruby 2007, S. 101 ff.).

POST vs. PUT

Der Grund warum POST meist so missverstanden wird, ist die Ähnlichkeit mit PUT, da mit beiden Methoden Ressourcen erstellt werden können. Doch es gibt einen gravierenden Unterschied zwischen POST und PUT. POST wird für das Erstellen von Ressourcen verwendet wenn der Client nicht weiß wie die URI zu der erstellenden Ressource aussehen soll. Dies ist meist der Fall wenn die untergeordneten Ressourcen in ihrer URI ihre Datenbank ID besitzen, wie aus dem vorherigen Beispiel zu sehen ist ("example.de/tagebuch/1"). Dem Client ist es in diesem Fall nicht möglich zu ermitteln oder zu wissen welche ID "noch frei" ist oder als nächstes kommen müsste. PUT hingegen setzt voraus, dass der Client die URI für die zukünftige Ressource kennt (vgl. ebd., S. 102). Um auf das vorherige Beispiel zurückzukommen, würde der Client via PUT einen "Tagebucheintrag" erstellen wollen, müsste er eine PUT-Anfrage direkt an die URI "example.de/tagebuch/1" senden. Falls nun ein "Tagebucheintrag" mit der ID 1 noch nicht existiert, würde dieser erstellt werden. Falls er aber schon existiert würde er ersetzt werden. Meist werden PUT-Methoden an Ressourcen benutzt die nicht ihre Datenbank ID innerhalb der URI besitzen, sondern eventuell ein Datum, wann sie erstellt wurden, oder einen Namen. Das macht es für den Client leichter die URI der neuen Ressource zu bestimmen. Wenn sie mit einer PUT-Anfrage gesetzt werden soll, müsste die "Tagebucheintrag" Ressource in dem Fall folgende URI besitzen: "example.de/tagebuch/2013-07-11" (vgl. ebd., S. 220).

Safe & Idempotenz

Wurde bei der Erstellung des Webservice auf das HTTP *uniform interface* geachtet und die HTTP-Methoden korrekt implementiert, so bringen diese noch ein paar Eigenschaften mit sich. So ist z.B. GET und HEAD als *safe* zu beachten.

Das bedeutet das diese Anfragemethoden nur zum lesen/abrufen von Daten benutzt werden, nicht zum verändern. Ein Client kann somit eine GET- oder HEAD-Anfrage 10 Mal ausführen und es wird das selbe sein, als wenn er diese einmal oder gar nicht ausgeführt hat (vgl. Richardson und Ruby 2007, S. 104).

Die zweite Eigenschaft ist die Idempotenz, die die Methoden GET, HEAD, PUT und DELETE innehaben. Idempotenz kommt aus der Mathematik und bedeutet das eine idempotente Operation das selbe Ergebnis hat wenn sie einmal oder mehr als einmal ausgeführt wird. Als Beispiel geben Richardson und Ruby eine Multiplikation mit 0 an. Diese Operation ist idempotent da $4*0*0*0$ das selbe ergibt wie $4*0$. Das bedeutet eine idempotente Operation auf eine Ressource ist wenn eine Ressource nach einer Anfrage einen bestimmten *resource state* besitzt und dieser sich nicht nach der selben mehrmaligen Anfrage verändert . Die DELETE-Methode ist demnach idempotent. Denn wird laut Richardson und Ruby eine Ressource gelöscht, verschwindet diese. Wird sie erneut gelöscht ist sie immer noch verschwunden. Idempotenz der PUT-Methode zeichnet sich dadurch aus, dass wenn eine Ressource via PUT erstellt wird und diese PUT-Anfrage wiederholt wird, existiert diese Ressource immer noch mit den selben Werten. Auch bei der Veränderung von existierenden Ressourcen via PUT ist es das selbe. Wiederholt man die PUT-Anfrage hat die Ressource immer noch die selben Werte wie nach der ersten PUT-Anfrage (vgl. ebd., S. 105). Dadurch das die Methoden GET und HEAD *safe* sind, sind sie gleichzeitig auch idempotent. Da sie diesen nicht verändern, bleibt der *resource state* der selbe. Egal ob man sie nun einmal oder mehrmals ausführt. Der Vorteil dieser beiden Eigenschaften ist, dass ein Client innerhalb eines unzuverlässigen Netzwerkes seine Anfragen ohne Bedenken wiederholen kann. Zum Beispiel wenn eine Anfrage verloren gegangen ist, kann der Client diese einfach wiederholen. Selbst wenn dann doch noch die erste Anfrage übermittelt wird ergibt es das selbe Resultat als wenn er diese einmal gesendet hat. Dies gilt aber nicht für die POST-Methode. Sie ist weder *safe* noch idempotent. Zwei gleiche POST-Anfragen an eine "Eltern" Ressource ergeben zwei neu erstellte Ressourcen (vgl. ebd., S. 105 f.).

2.4.7 Entwurfsverfahren

Richardson und Ruby haben ein generisches Entwurfsverfahren entwickelt mit dem man einen ROA Webservice entwickeln kann. Dieses Verfahren beinhaltet folgende Schritte:

1. Figure out the data set
2. Split the data set into resources
   For each kind of resource:
3. Name the resources with URIs
4. Expose a subset of the uniform interface
5. Design the representation(s) accepted from the client
6. Design the representation(s) served to the client
7. Integrate this resource into existing resources, using Hypermedia links and forms
8. Consider the typical course of events: whats supposed to happen?
9. Consider error conditions: what might go wrong?

vgl. Richardson und Ruby 2007, S. 148

Schritt eins

Laut Richardson und Ruby sollte ein Entwurf eines Webservices mit der Definition der Daten/Informationen, die ausgeliefert oder empfangen werden sollen, begonnen werden (vgl. ebd., S. 112). Das beinhaltet auch der Schritt 1 des Entwurfsverfahrens.

Schritt zwei

Nachdem herausgefunden wurde welche Daten ausgeliefert oder gespeichert werden, müssen diese Daten in Ressourcen aufgespalten werden. Dies stellt den Schritt zwei des Entwurfsverfahren dar. Zur Erklärung von Ressourcen siehe Abschnitt 2.4.1. Richardson und Ruby teilen Ressourcen in drei verschiedenen Arten von Typen auf, die bei dem Entwurf beachtet werden müssen. Die erste Art von Typ beinhaltet die einmalig vordefinierten Ressourcen. Diese Ressourcen sind ähnlich wie ein directory oder eine "Eltern" Ressource, die andere Ressourcen beinhalten und auf diese verweisen. Das bedeutet eine GET-Anfrage auf solch eine Ressource liefert als Antwort eine Liste von existierenden untergeordnete Ressourcen. Für ein Beispiel siehe Abschnitt 2.4.6 POST-Methode. Die meisten Webservices besitzen wenig oder gar keine dieser Art von Ressource. Der zweite Typ von den Ressourcen beinhaltet alle Ressourcen die ein Objekt repräsentieren.

Die meisten Webservices besitzen eine große Anzahl oder gar unendlich viele dieser Art von Ressourcen. Das liegt daran das dieser Typ von Ressourcen vom Client via Repräsentation auf dem Webservice erstellt und verändert werden kann. Ein Beispiel wäre das ein Client eine Notiz durch eine Repräsentation auf einem Webservice erstellen kann. Nicht nur ein Client kann solch eine Ressource erstellen, sondern mehrere Clients. Jede erstellte Notiz ist selbst eine Ressource die abgerufen, verändert oder gelöscht werden kann. Dadurch kann es zu einer großen Anzahl dieser Typen von Ressourcen kommen. Der letzte Typ beinhaltet die Ressourcen die das Ergebnis eines Algorithmus darstellen. Darunter fallen z.B. die, die die Ergebnisse von Querys liefern. Die meisten Webservices besitzen unendlich viele oder keine davon. Ein Beispiel von Richardson und Ruby ist eine Suchmaschine die eine unendliche Anzahl von algorithmischen Ressourcen besitzt. Dadurch das für jede Suchanfrage bzw. deren Ergebnis eine eigene Ressource existiert (vgl. Richardson und Ruby 2007, S. 114 ff.). Bei der Aufspaltung der Informationen sollten diese drei Typen beachtet werden. Wurden die Informationen in Ressourcen aufgespalten, müssen die Punkte drei bis neun des Entwurfsverfahrens (siehe Zitat 2.4.7) für jede Ressource wiederholt werden.

Schritt drei

Der Schritt drei beinhaltet das Benennen der Ressourcen. Das bedeutet, dass die Ressourcen eine eindeutige URI erhalten. Innerhalb eines *resource-oriented* Service sollen die URIs die Scoping-Informationen enthalten. Laut Richardson und Ruby gibt es drei Basisregeln für das URI Design. Zum einen sollte man Pfadvariablen für das Aufbauen von Hierarchien benutzen (z.B. ”/verein/mitglied”). Dort wo keine Hierarchien existieren sollte man Pfadvariabeln vermeiden und Interpunktionszeichen benutzen, wie z.B ein Semikolon (z.B. ”/verein/mitglied1;mitglied2”). Nach Konvention sollten Kommas benutzt werden, wenn die Reihenfolge der Scoping-Information eine Rolle spielt und Semikolons wenn sie es nicht tut. Query-Variablen sollten zum Andeuten von Parametern in Algorithmen benutzt werden (z.B. ”/verein/mitglieder?name=hannes&alter=9”) (vgl. ebd., S. 119 ff.). Die einmalig vordefinierten Ressourcen können eindeutig per URI identifiziert werden und bekommen daher im Entwurf eine ”feste” URI. Ist die vordefinierte Ressource von einer Objekt-Ressource abhängig, so existieren sie für jede Objekt-Ressource. Da eine Objekt-Ressource wie zuvor beschrieben für jedes Objekt existiert, können diese URIs innerhalb des Entwurfs nicht eindeutig definiert werden. Daher wird innerhalb des Entwurfs das sogennante URI Template benutzt {URI}. Dieses Template kann für jedes beliebige Objekt ersetzt werden. Innerhalb des Templates wird meist definiert wofür es ersetzt wird. Z.B. ”/users/{id}” definiert eine URI für eine User-Ressource mit einem Template. Das Template kann durch eine ID ersetzt werden. Sodass ein User über eine ID, die in der URI eingesetzt wird, eindeutig identifizierbar ist.

Schritt vier

Der nächste Schritt (Schritt vier) ist nur nötig für Ressourcen die nicht *read-only* sein sollen. Da *read-only* Ressourcen nur die Methoden GET, HEAD und OPTIONS unterstüzen. Der Schritt vier klärt ob und wie eine Ressource erstellt werden soll und ob sie Löschbar, Veränderbar und Abrufbar sein soll. Für die Definition welche HTTP-Methoden die Ressourcen unterstützen, sollen die Antworten auf die fünf folgenden Fragen helfen:

1. Möchten Clients neue Ressourcen dieser Art erstellen?
2. Wenn ein Client eine neue Ressource erstellt, wer bestimmt die neue URI? Der Client oder der Server?
3. Möchten Clients Ressourcen dieser Art verändern?
4. Möchten Clients Ressourcen dieser Art löschen?
5. Möchten Clients die Repräsentationen dieser Art von Ressourcen abfragen?

Wird für eine Ressource die erste Frage mit Nein beantwortet, so ist die zweite Frage damit hinfällig. Ist die Antwort darauf aber Ja, muss diese beantwortet werden. Soll der Client die URI bestimmen können, kann die Ressource via PUT-Methode erstellt werden. Ist der Server aber für die URI verantwortlich so muss die Ressource an einer "Eltern" Ressource via POST-Methode erstellt werden. Zur Erklärung siehe Abschnitt 2.4.6. Wird die dritte Frage mit Ja, beantwortet so muss die Ressource die PUT-Methode zum Aktualisieren der Ressource unterstüzen. Die vierte Frage mit Ja zu beantworten bedeutet das die Ressource die DELETE-Methode unterstützen muss. Wird die letzte Frage mit Ja, beantwortet so muss die Ressource die Methoden GET und HEAD unterstützen (vgl. Richardson und Ruby 2007, S. 149 f.). Nach dem die Fragen aus Schritt vier des Entwurfsverfahren beantwortet wurden, ist es erforderlich das zu Schritt fünf übergegangen wird.

Schritt fünf bis sieben

Schritt fünf und sechs beinhalten das Design der Repräsentation für die jeweiligen Ressourcen. D.h. in welchem Datenaustauschformat sie ausgeliefert und empfangen und wie die Repräsentation strukturiert werden soll. Die Repräsentationen werden nicht nur zur Auslieferung des sogenannten *resource states* einer Ressource genutzt, sondern auch zum Verändern des *resource states*. Wenn die ausgelieferten Repräsentationen auf andere ähnliche *resource states* oder Ressourcen verweisen, ermöglichen sie somit die *connectedness* (siehe Abschnitt 2.4.5). *Connectedness* verhilft zum Wechseln von Ressource zu Ressource anhand von Links, ähnlich wie auf einer Webseite (vgl. ebd., S. 126 f.). Laut Richardson und Ruby sollten die gleichen Repräsentationsformate gesendet und akzeptiert werden, wenn der *resource state* zu komplex ist (vgl. ebd., S. 151).

Die zuvor erwähnte *connectedness*, die durch die Repräsentationen ermöglicht wird, soll in dem Schritt sieben des Entwurfsverfahren in die Repräsentationen eingebaut werden. Das bedeutet die Links zu den bestehenden *resource states* oder Ressourcen, die im Kontext verwandt sind, in die Repräsentationen einzubauen (vgl. Richardson und Ruby 2007, S. 137 f.). Dies ist somit auch die Erfüllung der HATEOAS-Bedingung (siehe Abschnitt 2.3.1) und des Level drei des "Richardson Maturity Model" (siehe Abschnitt 2.3.5).

Schritt acht & neun

In den Schritten acht und neun wird für jede Ressource und jede unterstützte Methode definiert, was passieren soll wenn die Anfrage akzeptiert oder abgelehnt wird bzw. es zu einem Fehler kommt. Es wird definiert was als HTTP-*Header* und HTTP-Statuscode gesendet werden soll. Zudem kann für eventuelle GET-Anfragen definiert werden ob die conditional GET-Methodenform durch deren HTTP-*Headers* unterstützt wird (vgl. ebd., S. 139 f.). In dem Schritt neun ist es notwendig zu planen wann eine Anfrage evtl. abgelehnt werden sollte oder es zu einem Fehler kommen kann und was dann als Antwort gesendet werden soll. Ein Beispiel wäre wenn eine Ressource nicht gefunden wurde. Dann sollte ein HTTP-Statuscode 404 ("Not Found") gesendet werden (vgl. ebd., S. 141). Ein anderes Beispiel wäre aber auch das ein Client nicht autorisiert ist eine Ressource zu löschen (HTTP-Statuscode 401) oder das falsche Datenaustauschformat gesendet hat (HTTP-Statuscode 415). Auf diese Fälle muss auch geeignet reagiert werden (vgl. ebd., S. 156).

2.5 SOA

Service-Ortiented Architecture (SOA) ist eine Sammlung von Services die gemeinsam als größerer Service zusammenarbeiten. Eines der zentralen Ziele der Serviceorientierung ist den beteiligten Services die Grundmerkmale Verfügbarkeit, Zuverlässigkeit sowie die Fähigkeit die selbe Sprache zu sprechen, zu verschaffen, sodass die Services effektiv miteinander bzw. zusammen arbeiten können (vgl. Erl 2008, S. 84). Wie zuvor erwähnt kommunizieren die Services untereinander. Diese Kommunikation kann entweder genutzt werden um einfachen Datenaustausch oder eine Koordinierung von mehren Services für eine bestimmte Aufgabe/Tätigkeit zu gewährleisten (Barry 2013, vgl.). Laut den W3C Autoren Booth u. a. ist SOA eine Architektur für verteilte Systeme, die durch folgende Eigenschaften charakterisiert werden: Der Service, der mithilfe von SOA umgesetzt wurde, ist abstrakt gesehen eine logische Sicht ("logical view") auf die aktuellen Programme, Datenbanken und Buisnessprozesse.

Zudem ist der Service nachrichtenorientiert ("message orientation"). Das heißt das der *provider agent* und der *requester agent* über Nachrichten kommunizieren können, ohne zu wissen wie ein *agent* einen Service umsetzt. Ein Service, der Teil der SOA ist, ist mit maschinenverarbeitbarendaten beschrieben ("description orientation") und benutzt nur eine kleine Anzahl von Operationen mit relativ großen und komplexen Nachrichten ("granularity"). Eine nicht bedingte Voraussetzung ist die Benutzung der Services über ein Netzwerk ("network orientation"). Die Nachrichten werden in einem plattformunabhängigen und standardisierten Format zwischen den Schnittstellen übertragen ("platform neutral"). XML ist ein Format das diesen Bedingungen am meisten entspricht (vgl. Booth u. a. 2013b, Abschnitt 3.1). Laut Richardson und Ruby ist SOA ein eher weniger gut definierter Begriff. Jedoch ist einer der Aspekte von SOA die Unabhängigkeit der Architektur. So können die Services innerhalb SOA ressourcenorientiert, RPC lastig oder auch gemischte Architekturen besitzen (vgl. Richardson und Ruby 2007, S. 314).

Unterschied zu ROA

ROA setzt, wie in den vorherigen Abschnitten bereits erläutert und auch von Richardson und Ruby beschrieben, jegliche Information als Ressource um, die dann über eine URI erreichbar ist. SOA dagegen, wie laut Booth u. a. bereits zuvor beschrieben, besteht nicht aus Ressourcen sondern aus verschiedenen Services. Wobei die Sammlungen von Services auch *RESTful* Webservices beinhalten können. Für bestimmte Dienstleistungen oder Informationen die ein Client als Anfrage fordert, werden dann bestimmte Services innerhalb der SOA angesprochen. Die Anfragen an einen SOA Service sind meist sehr RPC lastig (vgl. ebd., S. 314). Bei ROA hingegen kann der Client nur die bestehenden Ressourcen anfragen (GET, HEAD), neue hinzufügen (POST o. PUT) oder alte löschen (DELETE). ROA ist eine *RESTful* Webservice-Architektur. Daher werden die Anfragen innerhalb eines *uniform interface* gestellt. Laut Richardson und Ruby ist SOA sozusagen auf einem höherem Level als ROA, bzw. eines *RESTful* Webservices (vgl. ebd., S. 20).

2.6 Servlet

Der bestehende Webservice, wie bereits zuvor schon erwähnt, wurde mithilfe von *Servlets* umgesetzt. In diesem Abschnitt wird kurz die *Servlet* Technologie erklärt. Das *Servlet* ist eine Java Technologie um plattformunabhängige Komponenten und webbasierende Applikation zu erstellen, ohne die Performance Probleme von CGI-Programmen.

Sie sind nicht nur plattform- sondern auch serverunabhängig und haben den vollen Zugriff auf alle anderen Java Klassen bzw. Bibliotheken (vgl. Oracle 2013d). Ein *Servlet* ist ein Java Programm das innerhalb eines Webservers läuft. Das *Servlet* erhält und reagiert auf Anfragen von Webclients, üblicherweise über HTTP (vgl. Oracle 2013e). Ein *Servlet* wird von einem *Servlet*-Container verwaltet. Der *Servlet*-Container verwaltet den Lebenszyklus eines *Servlet*, sowie deren Anfragen und Antworten. Um das *Servlet* dem *Servlet*-Container bekannt zu machen werden Deployment-*Deskriptor* verwendet (web.xml). Ein Webserver mit *Servlet*-Container ist z.B. der Apache Tomcat, mit dem *Servlets* depolyed und übers Web ansprechbar gemacht werden können (vgl. Rittmeyer 2013).

2.7 Java-REST-frameworks

Für die Umsetzung eines *RESTful* Webservice in Java gibt es verschiedene Vorgehensweisen und auch Technologien die eingesetzt werden können. So kann ein *RESTful* Webservice mit *Servlets* (siehe Abschnitt 2.6) aber auch mit *frameworks* umgesetzt werden. In dem folgenden Abschnitt werden mögliche Java-REST-*framework* verglichen, sodass das geeignetste *framework* innerhalb dieser Arbeit verwendet werden kann. Die *frameworks* ermöglichen das Erstellen eines *RESTful* Webservice anhand der JAX-RS Spezifikation. JAX-RS ist eine Java Spezifikation, die Entwicklern ermöglicht REST-konforme Web-Applikationen zu erstellen. Mithilfe der *Annotation*s, die in JAX-RS spezifiziert sind, ist es möglich ein *RESTful* Webservice leicht zu entwickeln (vgl. Potociar und Oracle 2013). Als Referenzimplementierung von JAX-RS gilt Jersey. Es wird auch namentlich in dem Java EE Tutorial von Oracle erwähnt (vgl. Oracle 2013c). Doch Jersey (vgl. Oracle 2013f) ist nicht das einzige *framework* was die JAX-RS Spezifikation implementiert. Unter anderem gibt es noch die *frameworks* RESTEasy (vgl. JBoss 2013b) und Apache CXF (vgl. Apache 2013). Ein weiteres Java *framework* für *RESTful* Webservices ist Restlet (siehe Restlet 2013), welches auch von Richardson und Ruby für Java vorgestellt wird (vgl. Richardson und Ruby 2007, S. 345 ff.). Restlet war eines der ersten REST-*frameworks*. Der Gründer von Restlet war auch an der JAX-RS Spezifikation beteiligt. Die JAX-RS Spezifikation wird auch von RESTlet, mit einer Erweiterung unterstützt (vgl. FAQ - Abschnitt 7 Pawson 2013). Die genannten *frameworks* wurden in der Tabelle 2.3 anhand bestimmter Eigenschaften verglichen. Die Dokumentation der *frameworks* wurde, um die Dokumentation besser zu vergleichen, in vier Sektionen unterteilt.

Eigenschaften	**Jersey**	**RESTeasy**	**RESTlet**	**CXF**
Erstes release	2007	21.01.2009	2005	JAX-RS support ab version 2.2.x
Letztes release	13.06.2013	30.05.2013	07.03.2013	15.05.2013
JAX-RS 2.0 support	Ja	Ja, aber nur in der beta	Nein, nur 1.0	Ja
Version	2.0	3.0-beta / 2.3.6-final	2.2-beta	2.7.5/2.6.8
Tomcat support	Ja	Ja	Ja	Ja
Applikationsserver support	Ja	Ja	Ja	Ja
Built-in HTTP-Server	Ja	Nein	Ja	Nein
Filter/Interceptor	Beides	Beides	Filter	Beides
Dokumentation				
Gegliedert	Ja	Ja	Ja	Ja
Beispiele	Ja	Ja	-	Ja
Projektbeispiele	Ja	Ja	-	Nein
Übersichtlich	Ja	Ja	-	Nein
Caching	anhand von Filtern	HTTP-*caching* Server *cache* und ETag Generation	Ja	Cachable-*Annotation*
Tests	Testframework	mithilfe von JUnit	Testframework	mithilfe von JUnit

Tabelle 2.3.: Java REST-frameworks (stand 16.06.2013)

Die Dokumentation wurde in die folgenden Sektionen unterteilt: ”Gegliedert” gibt an ob die Dokumentation gegliedert war, d.h. mit Inhaltsverzeichnis bzw. Kapiteln/Abschnitten, ”Beispiele” ob die Dokumentation Beispiele besaß die mit zur Erklärung benutzt wurden, ”Projektbeispiele” ob sie Projektbeispiele besaß , die man sich zum Ausprobieren herunterladen konnte und ”Übersichtlichkeit” ob sie anhand von einzelnen Seiten und Verweisen schnell lesbar und erfassbar war. So konnte man innerhalb eines Abschnittes trotzdem zum nächsten Abschnitt kommen oder zum Inhaltsverzeichnis zurück. Wie bereits zuvor schon erwähnt ist Jersey die Referenzimplementierung von JAX-RS. Wie aber auch aus der Tabelle 2.3 heraus geht, ist Jersey das aktuellste *framework*. Jersey und CXF sind die einzigen *frameworks* die als *stable release* die JAX-RS 2.0 Spezifikation implementiert haben. RESTlet war zwar eines der ersten REST-*frameworks*, aber es besaß eine nicht hinreichende Dokumentation, da die meisten Links nicht funktionierten. Somit konnte die Dokumentation auch nicht unter den Kriterien: besitzt es Beispiele, Projektbeispiele und ob die Dokumentation übersichtlich gestaltet ist, bewertet werden.

Durch die nicht erreichbare Dokumentation konnte auch nicht genau ermittelt werden wie das Testframework oder das *caching* funktioniert. RESTeasy hatte zwar die JAX-RS 2.0 Implementierung inne, aber nur in der *beta version*. Das *caching* von RESTeasy beinhaltete Server *cache*/HTTP-*caching* und Etag Generierung. Aber durch das Fehlen eines eigenem Testframeworks wurde es ausgeschlossen. Die Dokumentation des *frameworks* Apache CXF war nicht übersichtlich, da es schwer war sich auf der Homepage zurechtzufinden. Zudem besaß es keine Projektbeispiele bzw. waren diese nicht zu finden. Ein weiteres Problem bei CXF ergab sich durch das Fehlen eines eigenen Testframeworks, sowie durch das *deployen* auf einen Applikation Server. Wie z.B. Glassfish, das nur mit Konfigurationsaufwand möglich war. Durch die angestellten Vergleiche und der Dokumentation der *frameworks* wurde für das *framework* Jersey entschieden, welches im Laufe der Arbeit benutzt wird. Es war das einzige *framework*, welches allen Bedingungen entsprach. Zudem würde es nicht mehr Aufwand durch Konfiguration bei der Implementierung beanspruchen.

2.8 Datenbanksystem

Für die Speicherung der Daten, die vom Webservice ausgeliefert und auch empfangen werden, wird in dieser Arbeit ein Datenbanksystem (DBS) verwendet. Ein DBS setzt sich aus zwei Komponenten zusammen, siehe Abbildung 2.1, der Datenbank (DB) und dem Datenbankmanagementsystem (DBMS). Die Abbildung 2.1 wurde mithilfe von Kemper und Eickler 2009, S. 29 erstellt, jedoch stark vereinfacht. Laut Garcia-Molina, Ullman und Widom ist eine Datenbank nicht mehr als eine Sammlung von Informationen, die über eine lange Zeitperiode existiert. Die Datenbank wird gebräuchlich auch als eine Sammlung von Daten, die durch ein DBMS gemanagt werden, definiert (vgl Garcia-Molina, Ullman und Widom 2013, S. 1). Das Oxford Dictionary definiert "database" als eine organisierte Sammlung von Daten, die innerhalb eines Computes gespeichert sind. Diese Daten können über verschiedene Wege angesehen und benutzt werden (vgl. Wehmeier u. a. 2005, database). Das DBMS ist laut Oxford Dictionary ein System zum Organisieren und Managen von großen Datenmengen (vgl. ebd., database managment system). Garcia-Molina, Ullman und Widom sagen dem DBMS noch einige weitere Eigenschaften zu. Das DBMS erlaubt es Nutzern neue Datenbanken zu erstellen und deren Schemen, anhand einer data-definition language (kurz DDL), zu definieren. Schmenen bzw. ein Schema ist die logische Struktur von Daten. Das DBMS gibt den Benutzern die Möglichkeit bestimmte Daten/Informationen abzufragen oder mithilfe einer sogenannten data-manipulation language (kurz DML), zu verändern.

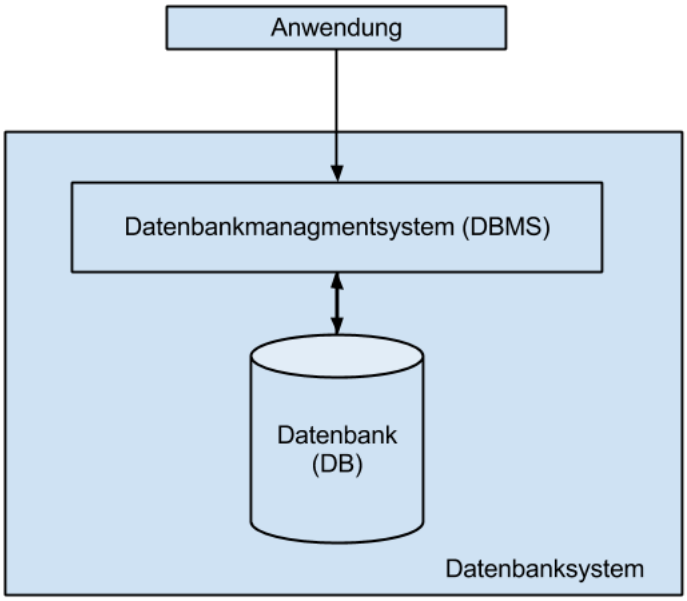

Abbildung 2.1.: Datenbanksystem

Die Speicherung von großen Datenmengen sowie deren Dauerhaftigkeit und das Zurücksetzen der Daten bei Fehlern, wird durch das DBMS ermöglicht. Der Zugriff für die Benutzer wird wiederum durch das DBMS gewährleistet. Zudem entstehen keine unerwarteten Änderungen durch die Benutzer (vgl. Garcia-Molina, Ullman und Widom 2013, S. 1 f.). Laut Kemper und Eickler ist ein Datenbanksystem in drei Abstraktionsebenen eingeteilt. Die physische Ebene, in der bestimmt wird wie die Daten gespeichert werden. In der logischen Ebene wird das zuvor erwähnte Datenbankschema festgelegt (welche Daten abgespeichert werden). Die dritte Ebene bilden die Sichten, die nur eine Teilmenge für die bestimmte Benutzer/Gruppen aus der definierten Gesamtinformationsmenge bereitstellen. Die Sichten sind auf die Bedürfnisse der jeweiligen Benutzer bzw. Benutzergruppen zugeschnitten (vgl. Kemper und Eickler 2009, S. 21 f.).

2.8.1 Relationales Datenmodel

DBMS basieren auf bestimmten Datenmodellen. Ein Datenmodel stellt die Struktur für die Modellierung der Daten zur Verfügung. Laut Kemper und Eickler ist das Datenmodel analog zu einer Programmiersprache, da es die Strukturen und Operatoren festlegt, die zur Modellierung benutzt werden können (vgl. ebd., S. 23). Das relationale Datenmodel ist einfach strukturiert, es existieren in diesem Model grundlegend nur Tabellen, auch Relationen genannt. Die Zeilen der Tabellen entsprechen den gespeicherten Datenobjekten (vgl. ebd., S. 71). Innerhalb eines relationalen Datenbanksystems werden die Daten dem Benutzer, wie bereits zuvor erwähnt, als Tabellen (Relationen) dargestellt.

Im Hintergrund, vor den Augen des Benutzers versteckt, ist eine komplexe Datenstruktur, die das schnelle Ausführen von verschiedenen Anfragen ermöglicht. Die Anfragen können in einer "high-level language" ausgedrückt werden. Die *structured query language* (kurz SQL) ist die wichtigiste *query language*, die auf dem relationalen Model basiert (vgl. Garcia-Molina, Ullman und Widom 2013, S. 3). Anhand von SQL-Anfragen können innerhalb eines relationalen Datenbanksystems Informationen aus der Datenbank extrahiert werden (vgl. Kemper und Eickler 2009, S. 109). Laut Kemper und Eickler sind die relationalen Datenbanksysteme die marktbeherrschenden Datenbanksysteme. Doch es existieren noch andere wie z.B. deduktive oder objektorientierte Datenbanken (vgl. ebd., S. 27).

2.8.2 MySQL

Für diese Arbeit wurde durch Vorgabe des Auftraggebers, MySQL als Datenbanksoftware benutzt. MySQL ist ein Open Source Projekt von Oracle, welches die MySQL Datenbank, das DBMS, die Datenbankanwendung/Client (Workbench) und andere Softwareprojekte beinhaltet (vgl. Oracle 2013g). Die MySQL Datenbanksoftwareprodukte basieren alle auf dem relationalen Datenmodel und unterstützen als Anfragesprache SQL (vgl. Oracle 2013a). Die MySQL Produkte werden in Client und Server unterteilt. Wobei der Server die Datenbank und das DBMS beinhaltet und der Client sich z.B. via MySQL connector oder TCP/IP sockets mit der Datenbank bzw. dem DBMS verbinden kann (vgl. Oracle 2013b). MySQL führt als weiteres Produkt die MySQL Workbench, die es ermöglicht sich mit dem DBMS per remote über TCP/IP oder über TCP/IP über SSH zu verbinden und die Datenbanken und Informationen abzufragen und zu verändern. Mithilfe der Workbench ist es des Weiteren möglich ein Enhanced Entity-Relationship (EER) Model zu erstellen und dafür direkt den SQL-Code zu generieren. Die MySQL Workbench wird somit auch innerhalb dieser Arbeit für den Zugriff zur Datenbank und zur Modellierung des Datenbankschemas benutzt. Ein EER-Model ist eine Erweiterung des normalen "Entity-Reltationship Model". Das EER-Model bringt neue Möglichkeiten mit, wie z.B. die subclass, die superclass, welche Vererbungen von Attributen und Beziehungselementen ermöglicht und anderes (vgl. Elmasri und Navathe 2013). Der Grund warum das EER-Model innerhalb des Entwurfs verwendet wird, resultiert daher das die MySQL-Workbench nur dieses Diagramm unterstützt. Das EER-Model erleichtert den Entwurf der Datenbank um einiges. Denn die N:M Beziehungen innerhalb des Diagramms werden automatisch von der MySQL-Workbench als Tabelle erstellt. Wird außerdem eine 1:N oder N:1 Beziehung erstellt, werden diese Relationen so "eliminiert" dass die Tabelle mit der N Beziehung einen Fremdschlüssel erhält.

2.8.3 Hibernate

Hibernate ist ein *framework* für Java und .NET, welches Object Relation Mapping (ORM) von Daten bzw. Informationen in Objekten ermöglicht. D.h. für relationale Datenbanken ist es möglich für eine Zeile einer Tabelle z.B. ein Java Objekt zu erstellen. Das *framework* kümmert sich um das mapping von Java Klassen zu Datenbanktabellen und von Java Datentypen zu SQL Datentypen (vgl. JBoss 2013a, Preface). Das Speichern von Java Objekten in relationalen Datenbanksystemen ist durchaus auch mit anderen *frameworks* möglich, wie z.B. Data Nucleus (vgl. DataNucleus 2013). Wie bereits zuvor beschrieben ermöglicht Hibernate das mapping von Daten bzw. Informationen aus einer relationalen Datenbank in ein Java Objekt. Die Java Klassen, die mit den Informationen aus den Tabellen der Datenbank gemappt werden sollen, werden auch als *entity*-Klassen bezeichnet. Hibernate ermöglicht das mapping von relationalen Tabellen in Java Klassen unter zwei verschiedenen Wegen. Der erste Weg ist die Benutzung von *Annotation*s, welche in der entsprechenden Java bzw. *entity*-Klasse platziert werden müssen. Der zweite Weg ist das mapping über XML mapping files. Innerhalb dieser Arbeit wird der Weg über die mapping files bevorzugt, da somit das mapping angepasst werden kann ohne das die Java Klasse verändert werden muss. Ein weiterer Grund ist die Verwendung von Jersey und der JAXB *Annotation*s, siehe dazu Abschnitt 2.7, was den Code bei mehreren *Annotation*s unübersichtlich macht. Damit Hibernate eine Verbindung zu der gewünschten Datenbank aufbauen kann, wird eine Hibernate configuration file benötigt. Innerhalb dieses files wird die Datenbankverbindung, der Datenbankdialekt, z.B. MySQL und andere Eigenschaften angegeben, wie z.B. die connection pool library. Für diese Arbeit wird die c3p0 library als connection pool benutzt, da diese innerhalb der Hibernate Dokumentation empfohlen wird (vgl. JBoss 2013a, 1.2.1. c3p0 connection pool). Um Zugriff auf die Datenbank zu bekommen, muss eine SessionFactory erstellt werden, die bei der Erstellung die configuration file einliest. Diese factory gibt die Möglichkeit sessions, die den Datenbankzugriff ermöglichen, zu erstellen. Durch eine Hibernate session kann ein *entity*-Klassen Objekt aus der Datenbank gelesen, gespeichert oder aktualisiert werden. Hibernate definiert für die Benutzung von sessions in Applikationen bestimmte Pattern, wie z.B. das *session-per-request* Pattern, welches in dieser Arbeit implementiert und benutzt wird. Ein Pattern ist ein Entwurfsmuster, welches für einen bestimmten Fall benutzt werden sollte. Wie der Name schon andeutet, wird für jede Anfrage an den Webservice eine eigene session benutzt. D.h. bei einer Anfrage, wird eine Session geöffnet bzw. aus einem Hibernate internen pool genommen und eine Hibernate eigene transaction mithilfe der session erstellt. Alle nötigen Operationen für diese Anfrage werden innerhalb dieser transaction ausgeführt. Danach wird die transaction commited, was bedeutet das alle Änderungen in der Datenbank übernommen werden und die session geschlossen wird. Tritt ein Fehler auf, wird die erstellte Hibernate transaction zurückgerollt und es werden keine Änderungen an der Datenbank vorgenommen, ähnlich wie bei einer Datenbanktransaktion.

Dieses Pattern ist bei Systemen, die auf Anfragen von Benutzern bzw. Clients reagieren müssen, durchaus sinnvoll (vgl. JBoss 2013a, 2.4.2. Session-per-request pattern).

2.9 Entwicklungswerkzeuge

In diesem Abschnitt werden die Werkzeuge beschrieben, die zur Entwicklung des REST-konformen Webservices benutzt werden und noch nicht bereits erwähnt bzw. erklärt wurden.

2.9.1 Netbeans

Netbeans ist eine Open Source integrated development environment (kurz IDE) die das Entwickeln von Java Desktop Applikationen, mobilen Applikationen und Web Applikationen ermöglicht. Eine IDE wie Netbeans, ermöglicht das Entdecken von Fehlern während des Schreibens. Z.B. das Fehlen von Klammern, das Hervorheben von Schlüsselwörten des Codes und anderen Dingen. Eine IDE kann zudem coding templates, coding tips oder refactoring tools anbieten. Anhand einer IDE kann man ein erstelltes Programm debuggen, was z.B. die Sicht auf die Werte von Variablen während der Laufzeit zu bestimmten breakpoints ermöglicht (vgl. Oracle 2013h).

2.9.2 Maven

Apache Maven ist ein Software Projektmanagmenttool, welches das Erstellen und Verwalten von Projekten ermöglicht. Netbeans bietet eine Anbindung für die Erstellung von Maven Projekten an. Durch eine pom file (pom.xml) oder auch Projekt Objekt Model genannt, kann Maven konfiguriert werden. So können z.B. bestimmte dependencies für das Projekt angelegt werden. D.h. das bestimmte libraries zur Ausführung bzw. compilierung benötigt werden. Diese angegebenen dependecies werden beim building Prozess des Projekts versucht aufzulösen. D.h. die angegebenen libraries werden in das locale repository geladen. Nur mithilfe des Source-Codes und der pom.xml wird das Ausführen eines Maven Projekts auf mehreren Rechnern ermöglicht. Die nötigen libraries werden beim build Prozess mit eingebunden. Maven ermöglicht noch einige weitere Sachen, wie das Ausführen von Tests oder das Erstellen einer Dokumentation (vgl. Maven 2013).

2.9.3 GIT

Git ist ein freies und open source verteiltes Versionskontrollsystem, für kleinere oder größere Projekte. Git ermöglicht das Entwickeln in verschiedenen, lokalen branches (Zweigen) und das Speichern oder Zurücksetzen von bestimmten Änderungen bzw. Ständen. Es ist möglich ein Git repository auf einem Server zu platzieren, sodass dort alle Änderungen und branches gespeichert werden. Arbeiten mehrere Projektmitglieder auf einem branch oder an einer Datei, so ist es möglich die Stände der Mitglieder zu *mergen*. D.h. zusammenzufassen und somit zu überprüfen ob die richtigen Änderungen übernommen werden. Das repository auf einem Server kann durch die Projektmitglieder aktualisiert werden. Besitzt ein Mitglied nicht den neuesten Stand des Projekts, so stellt der Server diese Version bereit und ermöglicht das Aktualisieren des lokalen repositories (vgl. Git 2013).

Analyse
3

In dem folgenden Kapitel wird eine IST-Analyse anhand des bestehenden Webservice erstellt. Danach wird aufgrund der bevorstehenden Implementation eines *RESTful* Webservices und der Verwendung von *Servlets* im bestehenden Webservice die Implementierung von *Servlets* und dem Jersey-*framework* analysiert und verglichen. Sodass nach dem Vergleich eine Technologie bevorzugt für die Implementierung verwendet werden kann. Jersey wird als *framework* benutzt da, anhand des Vergleiches in Abschnitt 2.7, Jersey sich als geeignetstes *framework* herausstellte. Da Daten innerhalb des Webservice auch persistiert werden müssen, wird innerhalb des letzten Abschnittes dieses Kapitels, Hibernate mit dem verwendeten *framework* aus dem bestehenden Webservice verglichen. Dies soll die Wahl eines geeigneten ORM-*frameworks* ermöglichen.

3.1
IST-Analyse

Diese Analyse soll alle Funktionalitäten des MoCCha Webservices aufzeigen sowie deren eventuelles Vorgehen. Es werden nicht nur die Funktionalitäten beschrieben sondern auch die ansprechbaren Ressourcen und deren Datenaustauschformate.

3.1.1
MoCCha

Der Webservice der umstrukturiert werden soll gehört dem MoCCha Projekt an und beliefert die gleichnamige Mobile Applikation mit Daten bzw. Informationen. Die MoCCha-App ist für Studierende gedacht und bietet einige Angebote/Services an. Zum einen ist es möglich mithilfe der Mobilen Applikation die Vorlesungsverzeichnisse für TU und Universität der Künste (UdK) sowie die Speisenpläne der Mensen des Studentenwerks einzusehen. Für die Mensa ist es unter anderem möglich mithilfe der App sich zu verabreden. Des Weiteren bietet MoCCha einen Veranstaltungskalender für die TU sowie das Programm der Staatsoper im Schillertheater. Die Neuigkeiten der TU über Twitter sind auch anhand von MoCCha verfolgbar (vgl. TLabs 2013a).

Der Webservice der für die Umstrukturierung vorgesehen ist liefert aber nicht alle Features (Daten/Information), die die MoCCha-App besitzt. Die Daten und Information die vom Webservice bisher ausgeliefert werden, werden in dem Abschnitt 3.1.2 benannt.

3.1.2 Webservice

Der MoCCha-Webservice wird auf einem Debian Linux Server der Version 6.0.6 innerhalb eines Apache Tomcat's der Version 6.0.35.0 ausgeführt. Der Debian-Server läuft innerhalb einer virtuellen Maschine und wird von den "Quality and Usability"-Labs verwaltet. Die Daten die für die MoCCha-App bereitgestellt werden, werden aus einer MySQL Datenbank gelesen. Die MySQL Datenbank hat die Version 5.1.63 und befindet sich auf dem Debian Server wie der Apache Tomcat. Der bestehende Webservice benutzt zur Verbindung mit der MySQL Datenbank ein ORM-*framework*. Bei dem *framework* handelt es sich um Data Nucleus, welches die JDO Java Spezifikation implementiert. Da der Webservice mithilfe von *Servlets* umgesetzt wurde, ist ein Deployment-Deskriptor (web.xml) in Verwendung. Anhand des Deployment-Deskriptors werden die *Servlets* dem *Servlet*-Container innerhalb des Tomcat's bekannt gemacht und die URIs über die sie erreichbar sind festgelegt. Für die Beschreibung des Deployment-Deskriptors siehe Abschnitt 2.6. In diesem Abschnitt werden alle bestehenden *Servlets* des MoCCha-Webservices kurz vorgestellt. Die weiteren Klassen die zum Webservice gehören bzw. die die Logik des Webservices ausmachen, werden nicht weiter erläutert. Denn das Ziel dieser Arbeit ist das Umstrukturieren des Webservices auf eine REST-konforme Architektur. Dabei wird versucht die Programmlogik außen vor zu lassen. In der Abbildung 3.1 werden die bestehenden *Servlets* des MoCCha-Webservices aufgelistet. Wie aus der Abbildung 3.1 zu entnehmen ist, ist eines der bestehenden *Servlets* das "AppDataServlet". Es ist für die Synchronisation der IOS und Android Applikationen zuständig. Denn durch das "AppDataServlet" ist es möglich die gespeicherten App-Daten abzurufen oder zu speichern. Das *Servlet* unterstützt die Methoden GET und POST (siehe Tabelle 3.1).

- AppDataServlet
- StaatsopernServlet
- CanteenServlet
- TUEventServlet
- CourseServlet
  - Friends
- DoodlePollServlet
  - PollServlet
- Communicator
- MoCChaInfo
- FileServlet

Abbildung 3.1.: MoCCha - Servlets

Die URIs, sowie deren mögliches Ergebnis bei einer POST- oder GET-Anfrage, sind in der Tabelle 3.1 aufgelistet. Das URI-Template "{user}", "{app}" oder "{version}" be-

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/appData/v1/{user}	liefert alle App's des Benutzers
	/appData/v1/{user}/{app}	liefert die App-Daten
	/appData/v1/{user}/{app}/timestamp	liefert den Timestamp der letzten App-Daten
	/appData/v1/{user}/{app}/version	liefert die App-Version
POST	/appData/v1/?appVersion=version	aktualisiert oder erstellt App-Daten

Tabelle 3.1.: AppDataServlet

deutet, dass dort ein beliebiger Wert eingesetzt werden kann. Ist dieser Wert valide, wird die Anfrage erfolgreich ausgeführt. So muss "{user}" mit einem Benutzername, "{app}" mit einem Applikationsnamen und "{version}" mit einer Versionsnummer ersetzt werden. Ein weiteres *Servlet*, siehe Abbildung 3.1, ist das "StaatsopernServlet". Es ist für das Ausliefern aller aktuellen Staatsopernevents zuständig. Dieses *Servlet* unterstützt nur die Methode GET und ist auch nur über einen Pfad erreichbar (siehe Tabelle 3.2). Für das Ausliefern aller Kantinen sowie deren Menüs ist das "CanteenServlet" zustän-

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/get/staatsoperCatalog	liefert alle heutigen Staatsopernevents

Tabelle 3.2.: StaatsopernServlet

dig. Dieses *Servlet* unterstützt wiederum nur die HTTP-Methode GET. Durch einige Query-Parameter kann man das Resultat der GET Methode beeinflussen. So liefert z.B. eine GET-Anfrage mit angegebenen Query-Parameter "type=list", die Liste aller Kantinen. Siehe dazu auch die Tabelle 3.3. Dort sind alle GET-Anfragen mit den möglichen Ergebnissen beschrieben.

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/get/canteenList?type=list	liefert alle Kantinen
	/get/canteenList?canteenID=ID	liefert das Menü der Kantine

Tabelle 3.3.: CanteenServlet

Es ist möglich mithilfe des ”TUEventServlet” alle aktuellen Events der TU Berlin abzurufen. Das *Servlet* unterstützt die HTTP-Methode GET und ist wie die anderen beiden *Servlets* zuvor, nur über eine URI erreichbar. Siehe für nähere Informationen Tabelle 3.4.

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/get/events	liefert alle Events der TU Berlin
	/get/events?eventID =ID	genauere Info's für das Event

Tabelle 3.4.: TUEventServlet

Für das ”TUEventServlet” ist zudem ein Query-Parameter ”eventID” vorgesehen. Durch die Angabe dieses Parameters soll die GET-Anfrage, wenn der Parameter gültig ist, genauere Informationen für das bestimmte Event liefern. MoCCha liefert nicht nur die Pläne der Mensen sondern auch die Kurspläne der UdK und TU. Die Kurspläne werden durch das ”CourseServlet” ausgeliefert. Durch Angabe eines Query-Parameters ”uni” kann die Universität gewählt werden. Damit für den Client nicht zuviel traffic entsteht kann dieser als zweiten Query-Parameter ”updatedSince” angeben. Der Parameter ”updatedSince” muss ein gültiges Datumsformat enthalten. Diese Angabe ermöglicht das Prüfen ob die Kurse sich seit dem geändert haben. Falls nicht, wird ein ”Not Modified” (304) Statuscode gesendet. Die angegebenen URIs aus der Tabelle 3.5 sind auch

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/get/courses	liefert alle Kurse der TU Berlin
	/get/courses?uni=0	liefert alle Kurse der TU Berlin
	/get/courses?uni=1	liefert alle Kurse der UdK
	/get/courses?updatedSince={datum}	liefert neue Kurse oder einen 304 Statuscode
	/get/courses?uni=1 &updatedsince={datum}	liefert neue Kurse der UdK oder einen 304 Statuscode

Tabelle 3.5.: CourseServlet

über den Pfad ”/courses/” erreichbar, d.h. anstelle von ”/get/courses”. Das *Servlet* ”Friends” ist eines der *Servlets*, was mehrere Funktionen miteinander vereint. Es ermöglicht das Hinzufügen eines Freundes, das Registrieren eines Benutzers, das Bestätigen einer Registrierung, das Aktualsieren des Benutzernamen, das Löschen eines Benutzers, das Hinzufügen von Kontakten sowie das Authentifizieren des Benutzers (siehe Tabelle 3.6). Diese Funktionalitäten werden durch die doPost Methode umgesetzt. Das bedeutet das das *Servlet* nur die HTTP-Methode POST unterstützt.

Die Daten die für die Verarbeitung der Funktionalitäten benötigt werden, werden als JSON innerhalb der Anfrage im *entity body* erwartet. Um es den Benutzern von MoCCha

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
POST	/post/friends	fügt einen neuen Freund hinzu
	/authenticate	aktualisiert den Devicetoken
	/user/updateDisplayName	aktualisiert den Benutzername
	/user/register	erstellt neuen Benutzer, sendet SMS nach der Erstellung auch wenn er bereits existiert
	/user/unregister	löscht einen existierenden Benutzer
	/user/validateRegistration	validiert die Registrierung
	/user/contacts	fügt Kontakte hinzu

Tabelle 3.6.: Friends

zu ermöglichen sich leichter zur Mensa zu verabreden existiert das "DoodlePollServlet", was durch das "PollServlet" abgelöst wurde. Es sind dennoch beide *Servlets* noch im Webservice vorhanden. Diese *Servlets* ermöglichen das Starten einer Umfrage, mithilfe dieser man sich Verabreden kann. Das "DoodlePollServlet" unterstützt die Methoden GET und POST. Mithilfe der GET Methode können alle Umfragen (engl. polls) für den Benutzer, der über sein Telefonhash identifiziert wird, geliefert werden. Zudem ist es möglich über die "pollID" ein bestimmten Umfrage abzufragen. (siehe Tabelle 3.7)

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/get/doodle ?pollID=ID	liefert Umfrage mit der ID
	/get/doodle?user=TeleHash	liefert alle Umfragen des Benutzers
POST	/post/doodle	Benutzer von Umfrage entfernen
		Umfrage schliessen
		Benutzer zur Umfrage hinzufügen
		Umfrage löschen
		Optionen zur Umfrage hinzufügen
		Umfrage aktualisieren
		neuen Umfrage erzeugen

Tabelle 3.7.: DoodlePollServlet

Über die POST-Methode des ”DoodlePollServlet” ist es möglich mehrere Funktionalitäten anzusprechen. Die Anfrage erfordert ein JSON-Objekt mit Umfrageinformationen. Abhängig von dem übermittelten JSON-Objekt, werden die Funktionalitäten des ”DoodlePollServlet” ausgeführt. Eine der Funktionalitäten ist z.B. das Erstellen oder das Löschen einer Umfrage, für mehr Funktionalitäten siehe Tabelle 3.7. Wie bereits zuvor erwähnt wurde das ”DoodlePollServlet” mit dem ”PollServlet” ersetzt. Das ”PollServlet” unterstützt, wie das ”DoodlePollServlet”, die Methoden GET und POST dazu jedoch noch die HTTP-Methode PUT. Das *Servlet* ist zudem nur an eine URI gemappt und zwar an ”/polls”. Um alle Umfragen eines Benutzers abzufragen muss eine GET-Anfrage an die URI ”/polls” mit dem Query-Parameter ”user” gesendet werden. Der Query-Parameter beinhaltet die gehashte Telefonnummer. Die unterstüzten HTTP-Methoden POST (doPost) und PUT (doPut) des ”PollServlet”, erwarten innerhalb der Anfrage im *entity body* eine Umfrage im JSON Format. Wird eine PUT-Anfrage mit einer Umfrage im JSON Format an die URI ”/polls” gesendet, so können die existierenden Umfragen verändert werden. Via POST-Anfrage kann eine Umfrage erstellt werden, siehe dazu auch Tabelle 3.8. Da das MoCCha-Projekt zur Erforschung des Nutzerver-

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/polls?user=TeleHash	liefert alle Umfragen des Benutzers
POST	/polls	erstellt eine neue Umfrage
PUT	/polls	aktualisiert eine existierende Umfrage

Tabelle 3.8.: PollServlet

haltens erstellt wurde, ist es erforderlich das die Logdaten der mobilen Applikationen ausgewertet werden können. Für das Empfangen und Speichern der Logdaten ist das *Servlet* ”Communicator” verantwortlich. Das *Servlet* unterstützt die HTTP-Methoden GET und POST. Es ist zudem auf die URIs ”/appID”, ”/post/logs” und ”/device” gemappt (siehe Tabelle 3.9). Wobei die URI ”/appID” nur auf eine GET-Anfrage reagiert und die anderen URIs für die POST-Methode vorgesehen sind. Eine GET-Anfrage auf die URI ”/appID” liefert konstant die ID = 527216534. Diese ist auch direkt in den Quellecode geschrieben. Das liegt daran das die Methode noch als Prototyp innerhalb des Webservices existiert. Die POST-Methode ist wie bereits zuvor erwähnt anwendbar auf die URIs ”/post/logs” und ”/device”. Eine POST-Anfrage auf die URI ”/post/logs” erwartet ein Log in JSON Format, welches dann gespeichert wird. Wird ein device im JSON Format mithilfe von POST an die URI ”/device” gesendet, so wird das device in der Datenbank gespeichert oder aktualisiert. Das *Servlet* ”MoCChaInfo” ermöglicht über eine GET-Anfrage das Liefern von Informationen über die derzeitige Anzahl an registrierten Geräten (Nutzern), sowie die Anzahl der aktiven Nutzer in den letzten 15 Minuten (siehe Tabelle 3.10). Das letzte *Servlet* aus dem MoCCha-Webservice ist das ”FileServlet”. Das *Servlet* ermöglicht das Ausliefern von Dateiinhalten als plain text.

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/appID	liefert die app ID (527216534)
POST	/post/logs	speichert die gesendeten Logs
	/device	speichert oder aktualisiert ein device

Tabelle 3.9.: Communicator

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/info	liefert die Anzahl an registrierten Geräten sowie die Anzahl der aktiven Nutzer in HTML

Tabelle 3.10.: MoCChaInfo

Die Datei die als Antwort auf eine GET-Anfrage zurückgesendet werden soll, wird über ihren Dateinamen identifiziert. Der Dateiname wird als Pfadvariable bei der GET-Anfrage übermittelt (siehe Tabelle 3.11). Das ist möglich, da das "FileServlet" innerhalb des Deployment-Deskriptors mit der URI "/files/*" gemappt ist. Der Stern steht für ein beliebigen Pfad. Dieser Pfad wird zur Identifizierung der Datei benutzt. Das "FileServlet" unterstützt zudem die HTTP-Methode HEAD. Diese Methode ist ähnlich wie GET, sie sendet nur bei der Antwort keinen *entity body*. Zur Erklärung siehe Abschnitt 2.4.6. Das "FileServlet" gehört zu den *Servlets* welche nicht mehr benutzt und mit der Umstrukturierung entfernt werden können.

HTTP-Methoden	**Pfade / URIs**	**Beschreibung**
GET	/files/{dateiname}	liefert den Dateiinhalt als Antwort, die Datei wird über ihren Namen identifiziert
HEAD	/files/{dateiname}	liefert nur die Metadaten (HTTP-*Header*)

Tabelle 3.11.: FileServlet

3.1.3 Ressourcen

Anhand der *Servlets* (siehe Abschnitt 3.1.2) und des Deployment-Deskriptors von dem MoCCha-Webservice, wurde die Abbildung 3.2 erstellt. Diese Abbildung stellt die vorhandenen Ressourcen des MoCCha-Webservice, in der bestehenden Hierarchie dar. Die Ressourcen werden durch die existierenden URIs dargestellt, siehe zur Erklärung Abschnitt 2.3.4. Die Darstellung der Ressourcen wird für den Entwurf zur Umstrukturierung des Webservices sowie für den späteren Vergleich beider Webservices benutzt.

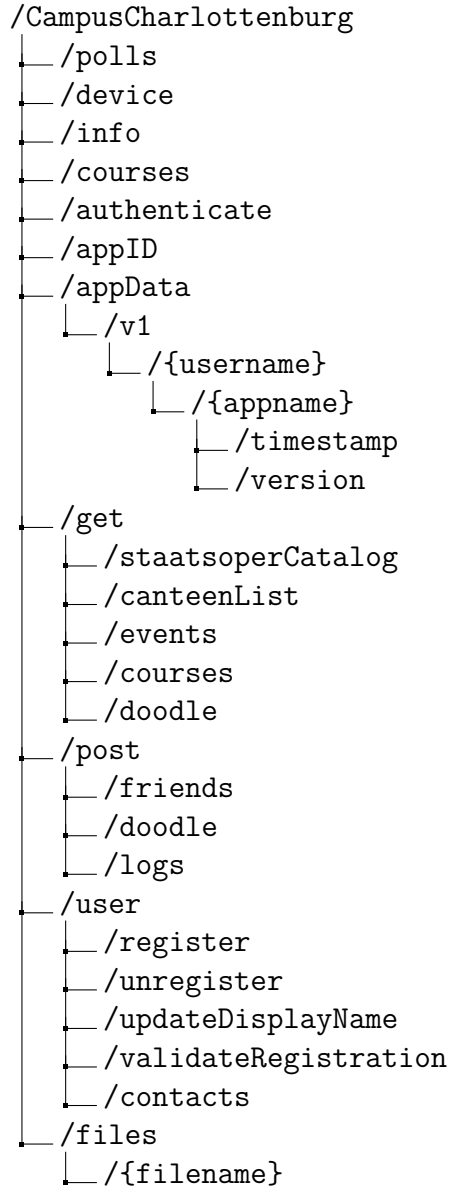

Abbildung 3.2.: MoCCha - Ressourcen

3.2 REST-Umsetzung

In dem folgenden Abschnitt wird, wie bereits zuvor erwähnt, ein *RESTful* Webservice unter zwei verschiedenen Vorgehensweisen implementiert. Dies dient dazu die Implementierung zwischen *Servlets* und einem Java-*framework* für einen *RESTful* Webservice zu vergleichen. Dieser Vergleich ist notwendig da der bestehende Webservice laut IST-Analyse (siehe 3.1) mithilfe von *Servlets* umgesetzt wurde und somit eventuell eine bessere Technologie für die Umsetzung gefunden werden kann. Zur Erklärung von der *Servlet* Technologie siehe Abschnitt 2.6. Jersey wird als Java-REST-*framework* für die Implementierung bzw. den Vergleich eingesetzt, da dieses *framework* anhand des Vergleiches in Abschnitt 2.7 ausgewählt wurde. Da der Webservice nur zur Demonstration dienen soll, wird er nur zwei Ressourcen besitzen. Die eine Ressource ermöglicht das Ausliefern aller existierenden User sowie das Erstellen eines neuen Users. Die Repräsentation dieser Ressource wird demzufolge eine Liste von User-Ressourcen besitzen und auf diese mithilfe von URIs verweisen. Dazu beinhaltet die Repräsentation die Information wie man einen User via POST-Methode an der Users-Ressource erstellt. Wurde eine POST-Anfrage an die Users-Ressource gesendet und diese beinhaltete alle nötigen Informationen, so wird eine neue User-Ressource bzw. ein neuer User erstellt. Die Antwort auf diese Anfrage beinhaltet den Statuscode 201 ("Created"), sowie den Location-*Header* der auf die erstellte Ressource mithilfe einer URI verweist. Die andere Ressource ist für jeden User individuell. Das bedeutet es handelt sich dabei um eine Objekt-Ressource. Durch diese Ressource kann man einen User verändern, auslesen und löschen. Die Informationen wie man den *resource state* der User-Ressource verändert, ist in der User-Repräsentation enthalten. Des Weiteren beinhaltet die Repräsentation eine URI, die zurück zur Users-Ressource weist. Ein User besitzt als Eigenschaften: einen Namen, ein Alter sowie eine ID, über die er eindeutig identifizierbar ist. Die ID wird innerhalb der URI enthalten sein. Die Konzentration ist einzig und allein auf die Umsetzung der HTTP-Methoden, die die Ressourcen unterstützen, gesetzt. Wie die Umsetzung zum Speichern oder Auslesen eines Users aus einer Datenbank dabei aussieht, ist irrelevant. Es wird deshalb auch nicht in der Beschreibung der Beispiele drauf hingewiesen oder weiter erklärt. Durch die Verweise und Informationen, innerhalb der Repräsentation, wie man den *resource state* einer User-Ressource verändert, ist das Level drei des Richardson Maturity Model (siehe Abschnitt 2.3.5) erreicht bzw. die HATEOAS Bedingung (siehe Abschnitt 2.3.1) erfüllt. Die Beispiel-Webservices wurden nach der ROA entwickelt, siehe Abschnitt 2.4. Die Informationen/Daten sind in Ressourcen aufgeteilt und über URIs adressierbar. HTTP wird als *uniform interface* benutzt. Es wird kein *application state* auf dem Server gespeichert und die Beispiele basieren auf dem Client-Server Model. Mit diesen Eigenschaften sind die Bedingung von ROA erfüllt. D.h. es handelt sich bei den Beispielen um ROA-Webservices. Somit sind diese auch *RESTful*.

3.2.1 Servlets

Wie zuvor bereits beschrieben wird in diesem Abschnitt ein *RESTful* Beispiel mithilfe von *Servlets* umgesetzt. Dabei sollen die Ressourcen Users und User existieren. Die Ressource Users unterstützt die Methoden GET (siehe Beispiel 3.1) sowie POST (siehe Beispiel 3.2).

```
1  @Override
2  protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
3    resp.setContentType("application/json;charset=UTF-8");
4    PrintWriter out = resp.getWriter();
5    StringBuffer uri = req.getRequestURL();
6    UserListExample users = service.getAll(uri);
7    if (users == null) {
8      URI userPath = createURI(uri.toString());
9      UserDescription userDesc = new UserDescription("NAME", "AGE");
10     users = new UserListExample(new ResourceOperation(userPath,"post",
11                                   "application/json", userDesc));
12   }
13   Gson gson = new Gson();
14   out.println(gson.toJson(users));
15 }
```

Beispiel 3.1: Servlet-Users-GET

Die Antwort auf eine GET-Anfrage an die Users-Ressource beinhaltet eine Liste von User-Ressourcen und die dazu gehörigen URIs, die auf diese Ressourcen verweisen. Die Repräsentation beinhaltet des Weiteren die Information wie man einen User via POST-Methode an der Users-Ressource erstellt. Die Repräsentationen werden als JSON zurückgegeben. Damit das JSON nicht selbst aufgebaut werden muss, wird hier eine library benutzt. Bei der library handelt es sich um GSON, die es ermöglicht Plain Old Java Objects (POJOs) in JSON-Repräsentation umzuwandeln (vgl. Unbekannt 2013). Da JSON innerhalb des *entity body* als Antwort gesendet wird, muss der "Content-Type" auf "application/json" gesetzt werden. Dies ist der MIME media type für JSON (vgl. Crockford 2006, Abschnitt 6). Damit das *Servlet* auch über den Pfad "/users" erreichbar ist, muss es entweder in einem Deployment-*Deskriptor* registriert sein oder es als *Annotation* "WebServlet" besitzen. Die Umsetzung mit der *Annotation* ist nur möglich mit der *Servlet* 3.0 Spezifikation. Das heißt der *Servlet*-Container muss Java EE 6 unterstützen. Das *Servlet*-Beispiel wurde mithilfe eins Deployment-*Deskriptor* umgesetzt (siehe Beispiel 3.7). Das Beispiel 3.2 zeigt die doPost-Methode des *Servlet* "UsersServlet". Diese Methode behandelt alle POST-Anfragen an die "/users" Ressource. Da anhand der POST-Methode ein neuer User erstellt werden soll, wird eine User-Repräsentation im *entity body* der Anfrage erwartet.

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
  if (req.getContentType().equalsIgnoreCase("application/json")) {
    PrintWriter out = resp.getWriter();
    BufferedReader reader = req.getReader();
    String line;
    StringBuilder builder = new StringBuilder();
    while ((line = reader.readLine()) != null) {
      builder.append(line);
    }
    Gson gson = new Gson();
    if (builder.toString().isEmpty()) {
      resp.setContentType("application/json;charset=UTF-8");
      out.println(gson.toJson(new ErrorResponse("The request must contain a user representation!")));
      resp.setStatus(400);
    } else {
      UserExample user = gson.fromJson(builder.toString(), UserExample.class);
      service.create(user);
      resp.setStatus(201);
      resp.setHeader("Location", req.getRequestURL().append("/").append(user.getId()).toString());
    }
  } else {
    resp.setStatus(415);
  }
}
```

Beispiel 3.2: Servlet-Users-POST

Fehlt diese Repräsentation wird mithilfe einer Fehlerantwort geantwortet. D.h. die Antwort beinhaltet ein JSON-Objekt mit einer Fehlernachricht und einen HTTP-Statuscode 400. Konnte der User erfolgreich anhand der gesendeten Repräsentation erstellt werden, so beinhaltet die Antwort ein Location-*Header* der die URI zur neuen User-Ressource angibt, sowie einen HTTP-Statuscode 201 (”Created”). Als Beispiel wäre folgender Location-*Header* denkbar: Location: http://example.de/Example-Servlet/users/240.
Für die Erklärung und das Verhalten von POST siehe Abschnitt 2.4.6. Nach dem Erstellen einer User-Ressource ist es möglich die erstellte Ressource abzufragen. Eine User-Ressource akzeptiert die HTTP-Methoden GET (siehe Beispiel 3.3), PUT (siehe Beispiel 3.5) sowie DELETE (siehe Beispiel 3.6).

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
  resp.setContentType("application/json;charset=UTF-8");
  makeRequest(req, resp, new Action() {
    @Override
    public void action(HttpServletRequest req, HttpServletResponse resp, PrintWriter out, Gson gson, Long id)
        throws ServletException, IOException {
      UserExample user = service.get(id, req.getRequestURL());
      if (user == null) {
        out.println(gson.toJson(new ErrorResponse("No user with the given ID: " + id + " exists!")));
        resp.setStatus(404);
      } else {
        out.println(gson.toJson(user));
      }
    }
  });
}
```

Beispiel 3.3: Servlet-User-GET

Sendet man eine GET-Anfrage an die User-Ressource so wird die doGET Methode des *Servlet* ”UserServlet” aufgerufen (siehe Beispiel 3.3). Das Problem hierbei ist das die User-Ressource die URI ”/users/{ID}” besitzt. Das bedeutet das ein User durch seine ID eindeutig identifizierbar ist und über /user/{ID} ansprechbar. Eine GET-Anfrage an die Ressource ”/users/5” liefert die Repräsentation des Users mit der ID 5. Die Repräsentation des Users wird als JSON im *entity body* der Antwort ausgeliefert. Wenn nun aber eine GET-Anfrage an die Ressource ”users/harry” gesendet wird, wird wieder die doGet Methode des *Servlet* ”UserServlet” aufgerufen.

Das liegt daran das innerhalb des Deployment-*Deskriptor* keine URI-Templates an *Servlets* gemappt werden können (siehe Beispiel 3.7). Somit muss jede Anfrage an die Ressource "/users/{ID}" vorher überprüft werden, siehe dazu Beispiel 3.4.

```
1  private void makeRequest(HttpServletRequest req, HttpServletResponse resp, Action act) throws ServletException,
       IOException {
2    PrintWriter out = resp.getWriter();
3    Gson gson = new Gson();
4    String path = req.getPathInfo();
5    if (!path.isEmpty() && path != null) {
6      String c = path.substring(1);
7      if (isNumber(c)) {
8        act.action(req, resp, out, gson, getID(c));
9      } else {
10       resp.setContentType("application/json;charset=UTF-8");
11       out.println(gson.toJson(new ErrorResponse("The path must contain the user ID")));
12       resp.setStatus(400);
13     }
14   } else {
15     resp.setContentType("application/json;charset=UTF-8");
16     out.println(gson.toJson(new ErrorResponse("The path must contain the user ID")));
17     resp.setStatus(400);
18   }
19 }
```

Beispiel 3.4: Servlet-User-Anfrage

Die Methode makeRequest aus dem Beispiel 3.4 wird von den Methoden doGet, doPost und doDelete vom *Servlet* "UserServlet" aufgerufen. Da somit die Überprüfung der Anfrage nicht in jeder Methode wiederholt werden muss. Wenn ein Fehler bei den Anfragen auftritt wie z.B. das eine Anfrage an die "/users/harry" URI gesendet wurde, dann wird mit einer Fehlerantwort geantwortet. Diese Antwort beinhaltet eine Fehlernachricht im JSON Format sowie ein HTTP-Statuscode 400. Mithilfe dieser Fehlerbehandlung ist es möglich die User-Repräsentation als Antwort auf eine GET-Anfrage nur dann zu senden, wenn die User-ID auch innerhalb der angefragten URI enthalten ist (siehe Beispiel 3.3).

```
1  @Override
2  protected void doPut(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
3    if (req.getContentType().equalsIgnoreCase("application/json")) {
4      makeRequest(req, resp, new Action() {
5        @Override
6        public void action(HttpServletRequest req, HttpServletResponse resp, PrintWriter out, Gson gson, Long id)
           throws ServletException, IOException {
7          BufferedReader reader = req.getReader();
8          String line;
9          StringBuilder builder = new StringBuilder();
10         while ((line = reader.readLine()) != null) {
11           builder.append(line);
12         }
13         if (builder.toString().isEmpty()) {
14           out.println(gson.toJson(new ErrorResponse("The request must contain a user representation!")));
15           resp.setStatus(400);
16         } else {
17           UserExample newUser = gson.fromJson(builder.toString(), UserExample.class);
18           newUser.setId(id);
19           UserExample oldUser = service.get(id, req.getRequestURL());
20           if (oldUser == null) {
21             service.create(newUser);
22             resp.setStatus(201);
23             resp.setHeader("Location", req.getRequestURL().toString());
24           } else {
25             service.update(newUser);
26             resp.setStatus(200);
27           }
28         }
29       }
30     });
31   } else {
32     resp.setStatus(415);
33   }
34 }
```

Beispiel 3.5: Servlet-User-PUT

Um eine User-Ressource mithilfe einer neuen Repräsentation zu ersetzen/aktualisieren oder eine neue Ressource mit gewünschter ID zu erstellen, muss eine PUT-Anfrage an die "/users/{ID}" Ressource gesendet werden. Für die Erklärung der HTTP-Methode PUT siehe Abschnitt 2.4.6. Wird eine PUT-Anfrage an die Ressource "users/{ID}" gesendet, wird die doPut Methode des "UserServlet" aufgerufen (siehe Beispiel 3.5). Die zuvor benannten Probleme existieren hier genauso, deshalb wird wiederum zur Fehlerbehandlung die makeRequest Methode aufgerufen (siehe Beispiel 3.4). Enthält die PUT-Anfrage keine Repräsentation der User-Ressource innerhalb des *entity body*, so wird, wie bei der POST-Methode der Users-Ressource, eine Fehlerantwort gesendet. Ist die Repräsentation enthalten wird die existierende User-Ressource mit der neuen Repräsentation ersetzt. Existiert die User-Ressource nicht (z.B. PUT an "/users/5"), dann wird diese an dieser Stelle mit der gegebenen Repräsentation und ID erstellt. Um eine User-Ressource zu löschen muss eine DELETE-Anfrage an die gewünschte User-Ressource gesendet werden. Diese DELETE-Anfrage ruft die doDelete Methode des *Servlet* "UserServlet" auf (siehe Beispiel 3.6).

```
1  @Override
2  protected void doDelete(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
3    makeRequest(req, resp, new Action() {
4
5      @Override
6      public void action(HttpServletRequest req, HttpServletResponse resp, PrintWriter out, Gson gson, Long id)
           throws ServletException, IOException {
7        UserExample user = service.get(id, req.getRequestURL());
8        if (user == null) {
9          resp.setContentType("application/json;charset=UTF-8");
10         out.println(gson.toJson(new ErrorResponse("No user with the given ID: " + id + " exists!")));
11         resp.setStatus(404);
12       } else {
13         service.delete(user.getId());
14       }
15     }
16   });
17 }
```

Beispiel 3.6: Servlet-User-DELETE

Die Fehlerbehandlungen für eine fehlgeleitete Anfrage werden wie bei den Methoden doGet und doPut von der Methode makeRequest erledigt. Wird eine DELETE-Anfrage an eine bestehende User-Ressource gesendet, wird diese gelöscht und es wird als Antwort ein HTTP-Statuscode 200 ("OK") gesendet. Existiert die User-Ressource jedoch nicht, so wird eine Fehlerantwort gesendet. Diese Antwort beinhaltet im *entity body* eine Fehlernachricht im JSON Format. Des Weiteren beinhaltetet die Antwort den HTTP-Statuscode 404 ("Not Found"). Das bedeutet das die gewünschte Ressource, die gelöscht werden sollte, nicht gefunden wurde. Wie bereits zuvor erwähnt wird der Deployment-*Deskriptor* dazu verwendet die *Servlets* mit dem *Servlet*-Container bekannt zu machen. Zudem wird innerhalb des *Deskriptor* die URI spezifiziert, über die das *Servlet* erreichbar ist. Das *Servlet* "UsersServlet" ist somit laut *Deskriptor* (siehe Beispiel 3.7) über die URI "/users" erreichbar. Da jeder einzelne User über seine ID erreichbar sein soll, wird das "UserServlet" auf die "/users/*" URI gemappt. Das hat aber zur Folge, wie bereits schon erwähnt, das nicht nur die Anfragen an die Ressource "/users/{ID}" dem *Servlet* gesendet werden, sondern auch Anfragen die z.B an folgende Ressource gehen "/users/harry/helm". Somit müssen etwaige Fehlanfragen behandelt werden (siehe Beispiel 3.4).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>UsersServlet</servlet-name>
        <servlet-class>de.example.servlet.UsersServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>UserServlet</servlet-name>
        <servlet-class>de.example.servlet.UserServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>UsersServlet</servlet-name>
        <url-pattern>/users</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>UserServlet</servlet-name>
        <url-pattern>/users/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Beispiel 3.7: Servlet-Web.xml

3.2.2 Vergleich

Wie zuvor bereits beschrieben und in Abschnitt 2.6 bereits angewendet, wird in diesem Abschnitt ein zweiter Beispiel-Webservice erstellt, um einen Vergleich zwischen *Servlet* und *framework* Implementierung zu erstellen. Für diesen Beispiel-Webservice wird das *framework* Jersey verwendet, für das durch den Vergleich mehrerer *frameworks* in dem Abschnitt 2.7 entschieden wurde. Der Beispiel-Webservice besitzt zwei Ressourcen. Diese zwei Ressourcen sollen User und Users heißen. Da der Beispiel-Webservice gegen einen Apache Tomcat entwickelt wurde, wurde wie in dem Abschnitt 2.6 auf den Deployment-*Deskriptor* gesetzt (siehe Beispiel 3.8). Es ist durchaus möglich einen Webservice mit Jersey zu erstellen ohne Deployment-*Deskriptor*. Aber wie bereits in dem Abschnitt 2.6 erwähnt ist dafür entweder die *Servlet* 3.0 Spezifikation nötig oder man benutzt den built-In Server den Jersey mitbringt. Die *Servlet* 3.0 Spezifikation ist z.B. bei einem normalen Applikation-Server enthalten, der die Java EE 6 Spezifikation implementiert hat (z.B. Glassfish oder JBoss). Oder auch bei dem Apache Tomcat 7.

```xml
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         version="2.5"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>ServletAdaptor</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <!--Jersey will check the given package after resources-->
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>de.example.jersey</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>ServletAdaptor</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Beispiel 3.8: Jersey-Web.xml

Mithilfe des Deployment-*Deskriptor* muss man nur den *Servlet*-Container innerhalb Jersey's registrieren und den Packagenamen der Ressourcen angeben, siehe Beispiel 3.8. Danach kann jede weitere Ressource, die für den Webservice benötigt wird, via "Path" *Annotation* markiert werden. Diese *Annotation* ermöglicht die Registrierung der Ressource innerhalb des Webservices und macht die Ressource über den angegebenen Pfad erreichbar (siehe Beispiel 3.9).

```
1 @Path("/users")
2 public class UsersResourceImpl {
3   //...
4 }
```

Beispiel 3.9: Jersey-Users-Ressource

Wird eine Klasse mit "Path" markiert, so stellt diese eine Ressource dar. Die Methoden der Klasse, die mit HTTP-Methodenmarkierungen markiert sind, gelten als die Methoden die von der Ressource unterstützt werden. Eine Methode kann auch mit "Path" markiert werden. Sie benötigt dazu noch eine HTTP-Methoden *Annotation*. Dies bedeutet das dann diese Methode für die angegebene Ressource unterstützt wird. Bei einer Anfrage mit der angegeben HTTP-Methode an der Ressource, wird dann die markierte Methode aufgerufen. (siehe Beispiel 3.10 - 3.14) Im Vergleich dazu müssen bei dem *Servlet*-Beispiel in Abschnitt 2.6 alle existierenden *Servlets* mithilfe des Deployment-*Deskriptor* registriert werden, siehe dazu Beispiel 3.7. Des Weiteren werden dann nur die HTTP-Methoden unterstützt, die auch von dem *Servlet* überschrieben wurden. Die Users-Ressource unterstützt zwei HTTP-Methoden, GET (siehe Beispiel 3.10 und POST (siehe Beispiel 3.11).

```
1   @GET
2   @Produces({MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
3   public Response getUsers() {
4     StringBuffer uri = new StringBuffer(info.getAbsolutePath().toString());
5     UserListExample users = service.getAll(uri);
6     if (users == null) {
7       UserDescription userDesc = new UserDescription("NAME", "AGE");
8       users = new UserListExample(new ResourceOperation(info.getAbsolutePath(),"post",
9                                                         "application/json", userDesc));
10    }
11    return Response.status(Response.Status.OK).entity(users).build();
12  }
```

Beispiel 3.10: Jersey-Users-GET

Eine GET-Anfrage an die Users-Ressource liefert wie bei dem *Servlet*-Beispiel 3.1 die selbige Repräsentation. Das Repräsentationsformat für die Users-Ressource oder auch für eine einzelne User-Ressource ist JSON oder XML. Anhand des Accept-*Header* kann der Client festlegen in welchem Format die Repräsentation geliefert werden soll. Die verfügbaren Formate werden innerhalb der "Produces" *Annotation* angegeben. Jersey wandelt den *entity body* der Antwort in die bestimmten Datenformate um. Sendet der Client innerhalb der Anfrage einen Accept-*Header* mit einem Datenformat das nicht verfügbar ist, so sucht Jersey das bestpassendste raus. Kann Jersey kein passendes Datenformat finden, wird automatisch als Antwort der HTTP-Statuscode 406 ("Not Acceptable") gesendet. Gibt der Client kein Accept-*Header* an, nimmt Jersey das erst angegebene Datenformat (vgl. Oracle 2013i). Bei der *Servlet*-Umsetzung ist es in dem Beispiel 3.1 nicht möglich ein anderes Datenformat zu wählen.

Um den Accept-*Header* zu unterstützen und mehrere Datenformate anzubieten bräuchte man mehrere *marshaller*, die die POJOs in das gewünschte Datenformat wandeln. Dieses Feature bringt das *framework* Jersey mit der *Annotation* "Produces" und "Consumes" mit. Die *Annotation* "Consumes" ist das Gegenstück zu "Produces" es zeigt an, welches Datenformat für die Anfrage innerhalb des *entity body* erwartet/akzeptiert wird. Via "Consumes" können auch mehrere Datenformate angegeben werden. Bei der Anfrage wird dann der HTTP-*Header* "Content-Type" geprüft, ob es sich um einen akzeptierten Content-Type handelt. Ist der Content-Type nicht in der Liste der unterstützten Datenformate, so wird als Antwort ein HTTP-Statuscode 415 ("Unsupported Media Type") zurückgegeben. Um eine User-Ressource zu erzeugen muss eine POST-Anfrage an die Users-Ressource gesendet werden. Die POST-Methode erwartet eine User-Repräsentation im JSON Format. Ist diese nicht vollständig, wird eine Fehlerantwort gesendet. Diese Fehlerantwort beinhaltet eine Fehlernachricht im JSON Format sowie den HTTP-Statuscode 400 ("Bad Request"). Wurde die Repräsentation jedoch komplett geliefert, wird die User-Ressource erstellt. Die Antwort beinhaltet einen Location-*Header*, der die URI aufweist, wo die neu erstellte Ressource zu finden ist. Außerdem wird mit der Antwort ein HTTP-Statuscode 201 ("Created") gesendet, siehe dazu das Beispiel 3.11.

```
1  @POST
2  @Consumes(MediaType.APPLICATION_JSON)
3  @Produces(MediaType.APPLICATION_JSON)
4  public Response createUser(UserExample user) {
5    if (user == null || user.getAge() == null || user.getName() == null)
6      return Response.status(Response.Status.BAD_REQUEST)
7                     .entity(new ErrorResponse("The request must contain a full user reprensentation!"))
8                     .build();
9
10   service.create(user);
11   UriBuilder builder = info.getAbsolutePathBuilder();
12   URI uri = builder.path(user.getId().toString()).build();
13   return Response.created(uri).build();
14 }
```

Beispiel 3.11: Jersey-Users-POST

Einer der großen Vorteile von Jersey, wie man hier auch wieder sieht, ist der *marshaller/unmarshaller* der die POJOs in Datenformate wandelt und auch wieder umwandelt. So wird bei dem Beispiel 3.11 innerhalb der Anfrage im *entity body* eine User-Repräsentation in JSON Format an diese Methode versendet. Jersey wandelt diese Repräsentation in ein Java Objekt und übergibt dieses Objekt als Parameter an die Methode. Das erspart dem Entwickler einiges an Arbeit, wie man aus dem *Servlet*-Beispiel 3.2 entnehmen kann. Dort wurde der *entity body* der Anfrage zuerst ausgelesen und dann die User-Repräsentation dem GSON *framework* übergeben, was daraus dann ein Java Objekt erstellt hat. Aber nicht nur das automatische *marshalling/unmarshalling* ist von Vorteil. Sondern auch die "Response" Klasse, die aus der JAX-RS Spezifikation stammt. Mithilfe dieser Klasse können einfach HTTP-Statuscodes, HTTP-*Header* und auch entities innerhalb des *entity body* zurückgegeben werden. In dem Beispiel 3.11 wird durch "return Response.created(uri).build();" eine Antwort, mit einem Location-*Header*, der die URI der User-Ressource enthält und ein HTTP-Statuscode 201 ("Created") gesendet. Die erstellte User-Ressource unterstützt wie in dem Abschnitt 2.6 drei HTTP-Methoden: GET (siehe Beispiel 3.12), PUT (siehe Beispiel 3.13) und DELETE (siehe Beispiel 3.14).

Da die User-Ressource über die Users-Ressource und ihrer eigenen ID erreichbar sein soll, d.h. "/users/{ID}", wurden die drei Methoden in die selbe Klasse platziert. Werden die Methoden nun mit "Path" markiert, werden die vorangegangenen "Path" *Annotation*s zum URI-Pfad hinzugefügt. Zur Erklärung siehe Beispiel 3.9. Dort wird der Pfad "/users" an der Klasse angegeben. Alle Methoden die nun mit HTTP-Methoden *Annotation*s markiert werden, sind unter diesem Pfad verfügbar (siehe Beispiel 3.10 oder 3.11). Hat nun aber eine Methode innerhalb dieser Klasse auch eine "Path" *Annotation*, wird dieser Pfad an die URI gehängt. Siehe Beispiel 3.12. Dort beinhaltet die "Path" *Annotation* "{id}". Nun ist diese Methode über "GET /user/{id}" erreichbar. Das bedeutet eine GET-Anfrage an die URI "/users/5" ruft die Methode aus dem Beispiel 3.12 auf. Die ID mit dem Wert 5 wird als Parameter übergeben. Dies geschieht durch die *Annotation* "PathParam". Mit dieser Markierung am Parameter und der *value* "id" kann Jersey das Template "{ID}" auflösen und den enthaltenen Wert der Methode übergeben. Dies ist, wie bereits in Abschnitt 2.6 erklärt, mit *Servlets* nicht so einfach. Dort muss dies mithilfe des Deployment-*Deskriptor* umgesetzt werden, siehe Beispiel 3.7. Dazu müssen Fehlanfragen behandelt werden die z.B. an "/users/harry" gehen. Das ist bei Jersey nicht der Fall. Jersey erkennt das ein Long in dem Beispiel 3.12 erwartet wird und leitet auch nur die Anfragen weiter, die diesem Schema entsprechen (z.B. "/users/54"). Bei einer Anfrage an "/users/harry" wird von Jersey aus eine "Not Found" Fehlerantwort gesendet. Die GET-Methode ist mit der "Produces" *Annotation* markiert und gibt somit entweder JSON oder XML zurück. Dies richtet sich, wie bereits zuvor bei den Beispiel 3.10 erwähnt, danach, was der Client im Accept-*Header* innerhalb der Anfrage angegeben hat.

```
1  @GET
2  @Path("{id}")
3  @Produces({MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
4  public Response getUser(@PathParam(value = "id") Long id) {
5    UserExample user = service.get(id, getURIStrinBuffer(info.getAbsolutePath()));
6    if (user == null) {
7      return Response.status(Response.Status.NOT_FOUND)
8                     .entity(new ErrorResponse("No user with the given ID exists!"))
9                     .build();
10   }
11   return Response.ok(user).build();
12 }
```

Beispiel 3.12: Jersey-User-GET

Eine weitere Methode die von der User-Ressource unterstützt wird, ist die PUT-Methode (siehe Beispiel 3.13). Diese Methode wird zum Ersetzen/Aktualisieren einer User-Ressource benutzt. Beziehungsweise wenn mit der gegebenen URI noch keine User-Ressource existiert wird anhand der User-Repräsentation eine neue User-Ressource an dieser Stelle erstellt. Ähnlich wie bei der POST-Methode der Users-Ressource (siehe Beispiel 3.11), wird die User-Repräsentation als JSON Format akzeptiert. Vor Aufruf der Methode wird die Repräsentation in ein Java Objekt gewandelt. Jersey ruft dann diese Methode mit den gewandelten Java Objekt als Parameter auf. Dazu kommt aber noch das die PUT-Methode auch ein "PathParam" besitzt. Dieser Parameter wird genauso wie bei der GET-Methode (siehe Beispiel 3.12) aus der URI geparst und beim Aufruf der Methode mit übergeben. Dadurch bleibt wie bereits zuvor erwähnt die ganze Parse-, die *unmarshalling*- sowie die Fehlanfragenbehandlungsarbeit, wie es bei der *Servlet*-Implementierung der Fall ist, erspart (siehe Beispiel 3.5).

```
@PUT
@Path("{id}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response updateUser(@PathParam(value = "id") Long id, UserExample user) {
  if (user == null || user.getAge() == null || user.getName() == null)
    return Response.status(Response.Status.BAD_REQUEST)
                   .entity(new ErrorResponse("The request must contain a full user reprensentation!"))
                   .build();

  UserExample oldUser = service.get(id, getURIStrinBuffer(info.getAbsolutePath()));
  user.setId(id);
  if (oldUser == null) {
    service.create(user);
    UriBuilder builder = info.getAbsolutePathBuilder();
    URI uri = builder.path(user.getId().toString()).build();
    return Response.created(uri).build();
  }
  service.update(user);
  return Response.ok().build();
}
```

Beispiel 3.13: Jersey-User-PUT

Die letzte Methode, die die User-Ressource unterstützt, ist die DELETE-Methode (siehe Beispiel 3.14). Sie ermöglicht es eine User-Ressource zu löschen. Wie die GET-Methode aus Beispiel 3.12 erwartet sie ein Parameter, die ID des Users. Diese ID wird genauso wie bei der GET- (siehe Beispiel 3.12) und PUT-Methode (siehe Beispiel 3.13) aus der URI geparst und beim Aufruf der Methode als Parameter übergeben. Auch hier ist wieder der Mehraufwand aus dem Beispiel 3.6 für die *Servlet*-Implementierung ersichtlich.

```
@DELETE
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response deleteUser(@PathParam(value = "id") Long id) {

  UserExample user = service.get(id, getURIStrinBuffer(info.getAbsolutePath()));
  if (user == null) {
    return Response.status(Response.Status.NOT_FOUND)
                   .entity(new ErrorResponse("No user with the given ID exists!"))
                   .build();
  }
  service.delete(id);
  return Response.ok().build();
}
```

Beispiel 3.14: Jersey-User-DELETE

3.2.3 Zusammenfassung

Aus den Beispielen in dem Abschnitt 2.6 und 3.2.2 sowie aus deren Erklärung ist schon ersichtlich was für ein Mehraufwand aufkommt, wenn keine *frameworks* benutzt werden. Würde bei dem *Servlet* Beispiel-Webservice auch kein JSON-*framework* benutzt werden, würde das den Code nochmal um einiges ausdehnen. Nicht umsonst wurden *frameworks* entwickelt um es den späteren Entwicklern leichter zu machen ähnliche Projekte oder Programme umzusetzen. In dem Vergleich von einer Implementierung eines *RESTful* Webservice mithilfe von *Servlets* und *frameworks*, ist ersichtlich das die Verwendung von Jersey die Wartbarkeit und Lesbarkeit erhöht und den Aufwand der Implementierung um einiges vermindert. Die Wartbarkeit und Lesbarkeit wird dadurch erhöht das mit Jersey weniger Verzweigungen (IF-Statements) und Fehlerbehandlungen nötig sind.

Dies ist anhand der GET-Methode der User-Ressource, von der *Servlet*-Implementierung (siehe Beispiel 3.3 und 3.4) und der Implementierung durch Jersey (siehe Beispiel 3.12), aufzeigbar. In der *Servlet*-Implementierung müssen die Fehlanfragen, die eventuell nicht für die Ressource bestimmt sind, behandelt werden. Der "PathParam" muss in der *Servlet*-Implementierung aus der angefragten URI selbst geparst werden. Dies ist mit Jersey durch die *Annotation*s "Path" und "PathParam" einfacher möglich. Mit den *Annotation*s, die auch in dem Abschnitt 3.2.2 genannt wurden, vermindert sich auch wie zuvor erwähnt der Aufwand der Implementierung. Ein Beispiel dafür sind auch die *Annotation*s "Produces" und "Consumes". Wird ein Datenformat mithilfe von "Produces" angegeben, so wandelt Jersey das Java Objekt, welches zurückgegeben werden soll, automatisch in das gewünschte Format. Jersey achtet dabei auch auf den Accept-*Header*, falls dieser vom Client angegeben wurde. Bei einer *Servlet*-Implementierung müsste man dieses Verhalten nach implementieren. Das selbe ist es mit der "Consumes" *Annotation*, die angibt welches Datenformat die Methode akzeptiert. Wenn bei einer Anfrage das korrekte Format übergeben wurde, werden die Daten automatisch in ein Java Objekt gewandelt. Dieses Verhalten müsste auch bei einer *Servlet*-Implementierung nach implementiert werden. Um den Aufwand dafür aufzuzeigen, siehe die Beispiele 3.2 und 3.11. So muss bei der *Servlet*-Implementierung als erstes der Content-Type HTTP-*Header* überprüft werden. Wenn das übergebene Format nicht unterstützt wird, muss eine Fehlerantwort zurückgegeben werden. Ist das Datenformat unterstützt muss der passende *unmarshaller* gewählt werden und die übergebenen Daten müssen aus dem *entity body* ausgelesen werden. Der *unmarshaller* muss dann die Repräsentation in ein Java Objekt umwandeln. Erst dann kann die eigentliche Methode bzw. die Funktionalität begonnen werden. Durch die Vorteile einer Jersey *RESTful* Webservice Implementierung wird dieses *framework* auch für die Umsetzung des Webservices in Kapitel 6 benutzt.

3.3 ORM-Vergleich

Da anhand der IST-Analyse im Abschnitt 3.1 festgestellt wurde das der bestehende Webservice Data Nucleus als ORM-*framework* benutzt, wird ein Vergleich von Hibernate und Data Nucleus innerhalb dieses Abschnittes erstellt. Siehe dazu die Tabelle 3.12. Data Nucleus implementiert die Spezifikationen JDO (siehe *JSR 243: JavaTM Data Objects 2.0 - An Extension to the JDO specification*) und JPA (siehe *JSR 338: JavaTM Persistence 2.1*). Hibernate dagegen implementiert die Spezifikationen JPA und Java Beans (siehe *JSR 220: Enterprise JavaBeansTM 3.0*). Das hat zur Folge das Data Nucleus nicht nur relationale Datenbanksysteme unterstützt, sondern auch objektbasierte.

Da in dieser Arbeit die Verwendung von MySQL gefordert ist, ist dieses Feature nachlässig. Hibernate implementiert nicht nur die Spezifikationen sondern auch eigene Features, im Gegensatz zu Data Nucleus. Als Beispiel wäre das built-in connection pool von Hibernate, was mithilfe von dem *framework* c3p0 oder Proxol auch erweitert werden kann.

Eigenschaften	**Data Nucleus**	**Hibernate**
Implementierte Spezifikationen	JDO und JPA	JPA und Java Beans
Datenbanksysteme	RDBMS und andere	RDBMS
Version	3.3	4.2.3
Letztes Release	27.06.2013	03.07.2013
Lizenz	open source	open source
Entwickler	Data Nucleus	JBoss
Cache	L2 *cache*	Query und L2 *cache*
Built-in Connection pool	Nein	Ja
Query Criteria API	JDO impl. nein, JPA Ja	Ja JPA und mit eigener API
Auto. Datentyp marshalling	Nein	Ja
Dokumentation		
Gegliedert	Ja	Ja
Ausführliche Beispiele	Ja	Ja

Tabelle 3.12.: Java ORM-frameworks (stand 25.07.2013)

Data Nucleus unterstützt zwar auch die Verwendung dieser *frameworks* aber kein built-in connection pool, sodass dies nicht ohne externes *framework* benutzt werden kann. Da das *framework* Data Nucleus die Spezifikationen JDO und JPA implementiert, muss zuvor entschieden werden welche Implementierung benutzt werden soll, da beide Implementierungen nicht simultan benutzt werden können. Im bestehenden Webservice wurde die JDO Implementierung benutzt. Innerhalb der JPA Spezifikation wurde eine "Query Criteria API" definiert. Diese API kann ohne SQL Kenntnisse verwendet werden um leichter Anfragen für das DBMS zu erstellen. Hibernate besitzt unter anderem eine weitere "Criterion API", die die "Query Criteria API" somit noch erweitert. Anhand von Hibernate ist es, wie bereits zuvor erwähnt, möglich Java Datentypen automatisch in JDBC bzw. SQL Datentypen umzumappen, sodass diese persistiert oder auch wieder ausgelesen werden können. Diese Funktionalität fehlt in Data Nucleus und muss für jedes Mapping angegeben werden. Beide *frameworks* besitzen einen *cache*, einen sogenannten "L2 Cache". Dieser ermöglicht das *caching* von ausgelesenen Java Objekten. So müssen diese nicht wiederholt ausgelesen werden. Hibernate besitzt nicht nur diesen "L2 Cache", sondern auch einen sogenannten "Query-Cache". Dieser *cache* ermöglicht das *caching* von selbst erstellten Querys wie z.B. über die "Query Criteria API" oder durch eigens erstellte Querys, mithilfe von SQL. Die Ergebnisse der Querys werden zwischengespeichert, sodass sie nicht wiederholt ausgeführt werden müssen. Nicht nur die Funktionalitäten der *frameworks* wurde verglichen, sondern auch deren Dokumentation.

Die Dokumentation wurde in zwei Teile aufgelöst, so kann sie besser bewertet werden. Unter dem Teil "Gegliedert" ist zu verstehen, ob die Dokumentation in einzelnen Abschnitten und Kapiteln gegliedert wurde. Dies traf bei beiden *frameworks* zu. Der Teil "Ausführliche Beispiele" bedeutet das es für die Dokumentation auch Beispiele gab, die das Thema genau erläutern und darstellen. Auch dies traf auf beide *frameworks* zu. Wobei Data Nucleus viele Beispiele sehr knapp gestaltete. ObjectDB Software Ltd. ermöglicht es einen Performance-Vergleich für verschiedene ORM-*frameworks* mit verschiedenen DBMS zu erstellen. Dadurch konnte Data Nucleus mit Hibernate auf einem MySQL DBMS verglichen werden. Hibernate lag bei fast allen Performance-Tests vor Data Nucleus, siehe dazu ObjectDB Software Ltd. 2013. Durch die Performance-Tests von ObjectDB Software Ltd. und dadurch das Hibernate ein besseres *cache* Verhalten, eine eigene criteria API und ein build-in connection pool besitzt, wurde für das *framework* Hibernate entschieden. Hibernate wird in dieser Arbeit als ORM-*framework* benutzt.

4 Anforderungsanalyse

In dieser Arbeit und somit in den folgenden Kapiteln, soll ein Webservice der auf *Servlets* basiert auf eine REST-konforme Architektur umstrukturiert werden. Um den Webservice allen Anforderungen entsprechend umzustrukturieren, wird in diesem Kapitel eine Anforderungsanalyse erstellt.

Analyse

Der bestehende MoCCha-Webservice soll auf eine REST-konforme Architektur umstrukturiert werden. Als REST-konforme Architektur wurde ROA gewählt, siehe dazu Abschnitt 2.4. Bei der Umstrukturierung sollen alle Funktionalitäten, die innerhalb der IST-Analyse aufgedeckt (siehe Abschnitt 3.1) wurden, erhalten bleiben, bis auf die, die explizit nicht mehr gebraucht werden. Der Webservice muss allen REST sowie ROA-Bedingungen entsprechen. Zudem sollte der Webservice nach Anforderungen der T-Labs in Java entwickelt werden. Der Webservice sollte mithilfe von Jersey umgesetzt werden, da dies vorteilhafter im Gegensatz zu einer *Servlet* Implementierung befunden wurde. Siehe dazu den Vergleich von *Servlet* und *framework* Implementierung im Abschnitt 3.2. Nach erfolgreichem Umstrukturieren sollten die Webservices, wenn möglich, verglichen werden. Beispiele für die Vergleiche wären die Wartbarkeit, die Lesbarkeit, die Erweiterbarkeit, die Komplexität, die Anzahl an Ressourcen, die eventuelle Struktur und REST-konformität. Die *Servlets* sollten durch einzelne Ressourcen ersetzt werden. Die Repräsentationen der Ressourcen sollten, nach Anforderungen der T-Labs und durch die Benutzung von Mobilen-Clients, im JSON Datenaustauschformat ausgeliefert werden. Das *Servlet* "Communicator" ist mit dem Projekt "AppTrack" zu ersetzen. Bei diesem Projekt handelt es sich um die Erstellung eines generischen Loggers der von verschiedenen mobilen Applikationen benutzt werden kann. Mithilfe dieses Loggers ist es außerdem möglich die Logdaten visuell via Diagramme darzustellen. Da der Logger von der Seite des Clients angesprochen wird, ist ausgehend vom neuen Webservice nichts weiter zu tun. Das *Servlet* "Friends", wie bereits schon aus der Tabelle 3.6 hervorgeht, vereint viele verschiedene Funktionalitäten. Diese sollten auf mehrere Ressource aufgespalten werden.

Eine der Ressourcen sollte eine Benutzer-Ressource sein, die das Erstellen bzw. Registrieren sowie das Aktualisieren eines Benutzers ermöglicht. Die Validierung der Telefonnummer eines Benutzers soll wie bei dem "Friends" *Servlet* als Ressource erstellt werden. Jedoch sollte die Logik zur Validierung dahinter in Ausblick stehen. D.h. innerhalb dieser Arbeit soll das Augenmerk nur auf das Gerüst der Validierung liegen, damit ist die Ressource gemeint. Da die Versendung einer SMS mit weiteren *frameworks* bewerkstelligt werden muss, sollte die Funktionaliät der Versendung einer SMS für die Validierung einer Telefonnummer, nach dieser Arbeit in Ausblick stehen. Dennoch um das Gerüst bereits zu implementieren, soll die Ressource einen Validierungscode akzeptieren können. Der Benutzer sollte folgende Eigenschaften besitzen: eine UUID, die das Gerät identifiziert, einen Token, einen Namen, die Telefonnummer als SHA1 *hash*, einen Registrierungscode, mit dem der Benutzer seine Registrierung validiert, ein Validierungsdatum, wann er seine Registrierung validiert hat, sowie eine Kontaktliste die der Benutzer mithilfe seines Telefonbuchs befüllt. Die Kontaktliste sollte, außer das sie erweiterbar ist, auch noch abrufbar sein. Die Möglichkeit sich für die Mensa zu verabreden sollte weiterhin bestehen. Diese Verabreden-Funktion wurde von dem *Servlet* "DoodlePollServlet" getragen, was aber wie bereits schon erwähnt, von dem "PollServlet" ersetzt wurde. Die selbe Funktionalität wie das "PollServlet" soll auch über eine Ressource erreichbar sein. D.h. es sollte einem Benutzer möglich sein eine Umfrage zu erstellen, zu löschen oder auch zu verändern. Andere Benutzer sollen zu der Umfrage hinzugefügt werden können und zudem auch weitere Optionen. Eine Option soll eine bestimmte Mensa sein, zu der man sich zum Mittag verabreden kann. Jeder eingeladene Benutzer soll die Möglichkeit besitzen die Option mit JA/NEIN oder VIELLEICHT zu beantworten, dies kann auch durch eine Bewertung dargestellt werden. Um auch die Events für die UdK liefern zu können oder später auch allgemeinere, sollte das *Servlet* "TUEventServlet" mit einer Ressource ersetzt werden. Diese soll es ermöglichen die Events für verschiedene Universitäten zu liefern. Das selbe gilt für die Auslieferung von Kursen. Die Funktionalitäten der *Servlets* "CanteenServlet", "StaatsopernServlet" sowie "AppDataServlet" soll beibehalten werden und benötigt keine Veränderung, bis auf die Umstellung auf RESTkonformität. Das Ausliefern von Dateiinhalten ist nicht mehr erforderlich. So kann das *Servlet* "FileServlet" ganz entfernt werden. Das selbe gilt für das "MoCChaInfo" *Servlet*. Jede Ressource die GET unterstützt, sollte zudem auch HEAD und conditional GET unterstützen. Zur Erklärung siehe Abschnitt 2.4.6. Die Daten die von dem Webservice ausgeliefert oder empfangen werden sollten, nach Anforderungen der T-Labs, innerhalb einer MySQL Datenbank persistiert werden. Dabei sind die Daten der Parser ausgenommen. Zur Persistierung der Daten sollte Hibernate benutzt werden, siehe dazu den Vergleich von ORM-*frameworks* im Abschnitt 3.3. Als Feature wäre denkbar das die Ressourcen über den Pfad "/v2" erreichbar sind. Dieser Pfad soll die Version anzeigen, sodass man Änderungen am Server oder neue Versionen später mit fortlaufenden IDs deployen könnte. Ein weiteres Feature wäre es die Programmlogik zur Beschaffung der Daten (Parser für die Webseiten) in ein externes Java-Programm auszulagern.

Entwurf
5

Anhand der Anforderungen die zuvor genannt wurden, wird in dem folgenden Kapitel der Entwurf für einen *RESTful* Webservice erstellt. Da der alte MoCCha-Webservice Re-Designed werden soll, wird ein *RESTful* Webservice erstellt der allen Anforderungen entspricht. Die Programmlogik zur Beschaffung der Daten bleibt dabei unberüht und wird vom "altem" Webservice daher übernommen. Dabei sind die Parser für die Events der Staatsoper, Speisepläne der Mensen usw. gemeint. Da aber laut Anforderungsanalyse (siehe Abschnitt 4) auch einige Daten gespeichert werden müssen, wird in dem folgenden Abschnitt 5.1 das Datenbankschema beschrieben.

5.1 Datenbankschema

Anhand der Anforderungsanalyse in Kapitel 4 wurde ein ER-Diagramm erstellt (siehe Abbildung 5.1). Das Diagramm beinhaltet die entities "User", "Poll" sowie "Option". Wie auch schon in Kapitel 4 beschrieben hat der "User" folgende Eigenschaften: einen Namen, eine UUID, ein Telefonhash, ein Validierungsdatum, ein Registrierungsdatum und einen Registrierungskey. Das Registrierungsdatum wurde als Eigenschaft noch hinzugefügt, da es aus Analysezwecken eventuell interessant ist. Somit kann man erkennen wann der "User" sich registriert bzw. zum ersten mal angemeldet und wann er seine Registrierung validiert hat. Der "User" hat eine "1:N" Beziehung namens "owns" zu der Tabelle "App". Diese Beziehung entsteht dadurch das ein "User" mehrere Applikationen besitzen kann und eine Applikation in diesem Kontext aber nur zu einem User gehören soll. Jede Applikation hat mehrere Versionen. Aber eine Version gehört nur zu einer Applikation. Daraus entsteht die Beziehung "has", siehe Abbildung 5.1. Wie zuvor beschrieben ist diese "has" Beziehung demzufolge eine "1:N" Beziehung. Innerhalb der Versionstabelle werden besondere Daten gespeichert die zu der bestimmten Version und Applikation zugehörig sind. Die Zeit der Speicherung der Daten wird innerhalb des timestamps gespeichert. Eine weitere Tabelle ist die "Poll" Tabelle. Ein "Poll" (eine Umfrage) kann von einem "User" erstellt werden. Wobei ein "User" mehrere "Poll's" erstellen, ein "Poll" aber nur von einem "User" erstellt werden kann. Daher ergibt sich die in Abbildung 5.1 dargestellte 1:N Beziehung "create".

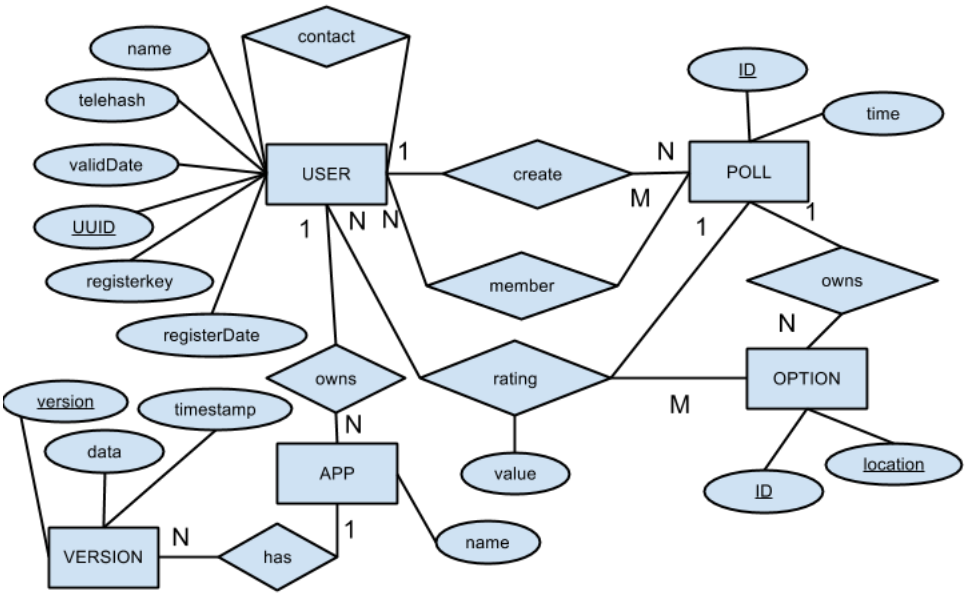

Abbildung 5.1.: MoCCha-ER-Model

Eine weitere Beziehung zwischen der ”User” und ”Poll” Tabelle ist die ”member” Beziehung. Da ein ”User” bei mehreren ”Polls” Mitglied sein kann und ein ”Poll” mehrere ”User” als Mitglied haben kann, ist dies eine N:M Beziehung. Die ”Poll” Tabelle besitzt zwei Eigenschaften: eine ID, über die sie eindeutig identifizierbar ist und ein Zeit, die definiert für wann die Umfrage erstellt wurde. Z.B. wann man sich in einer Mensa treffen möchte. Eine weitere Beziehung ist die ”owns” Beziehung zwischen ”Poll” und ”Option”. Diese ist wiederum eine 1:N Beziehung da ein ”Poll” mehrere ”Options” besitzen kann aber eine ”Option” nur einem ”Poll” angehört. Die Tabelle ”Option” besitzt zwei Eigenschaften: einen Ort sowie eine ID, über die die Option eindeutig identifizierbar ist. Die Beziehung ”rating” bezieht sich auf ”User”, ”Poll” und ”Option” und hat eine Eigenschaft ”value”, was einen Wert für die Bewertung beinhaltet. Ein ”User” kann für einen ”Poll” mehrere ”Options” bewerten. Ein ”Poll” hat mehrere ”Options”, die von mehreren ”Usern” bewertet werden können. Eine ”Option” kann innerhalb eines ”Polls” von mehreren ”Usern” bewertet werden. So ergibt sich die Beziehung ”rating” die in der Abbildung 5.1 dargestellt ist. Die Tabelle ”User” hat eine Beziehung mit sich selbst, die ”contact” Beziehung. D.h jeder ”User” kann ein Kontakt eines anderen ”Users” sein. Die Tabellen die aus dem ER-Model resultieren sind: ”User”, ”Contact”, ”App”, ”AppData”, ”PollMember”, ”Poll”, ”Option”, ”OptionRating” und ”OptionRatingValue”, siehe dazu auch Abbildung 5.2. Der Grund dafür ist das alle N:M Relationen durch eigene Tabellen repräsentiert werden sollten, da es sonst laut Kemper und Eickler zu ”Anomalien” kommen kann. Die anderen Beziehungen (1:N oder N:1) können ”eliminiert” und als Fremdschlüssel in den teilhabenden Tabellen dargestellt werden (vgl. Kemper und Eickler 2009, S. 78 ff.).

Die Abbildung 5.2 wurde mithilfe der MySQL Workbench angefertigt, da als Datenbank auch eine MySQL-Datenbank verwendet wird. Zur Erklärung von MySQL bzw. MySQL Workbench siehe Abschnitt 2.8.2.

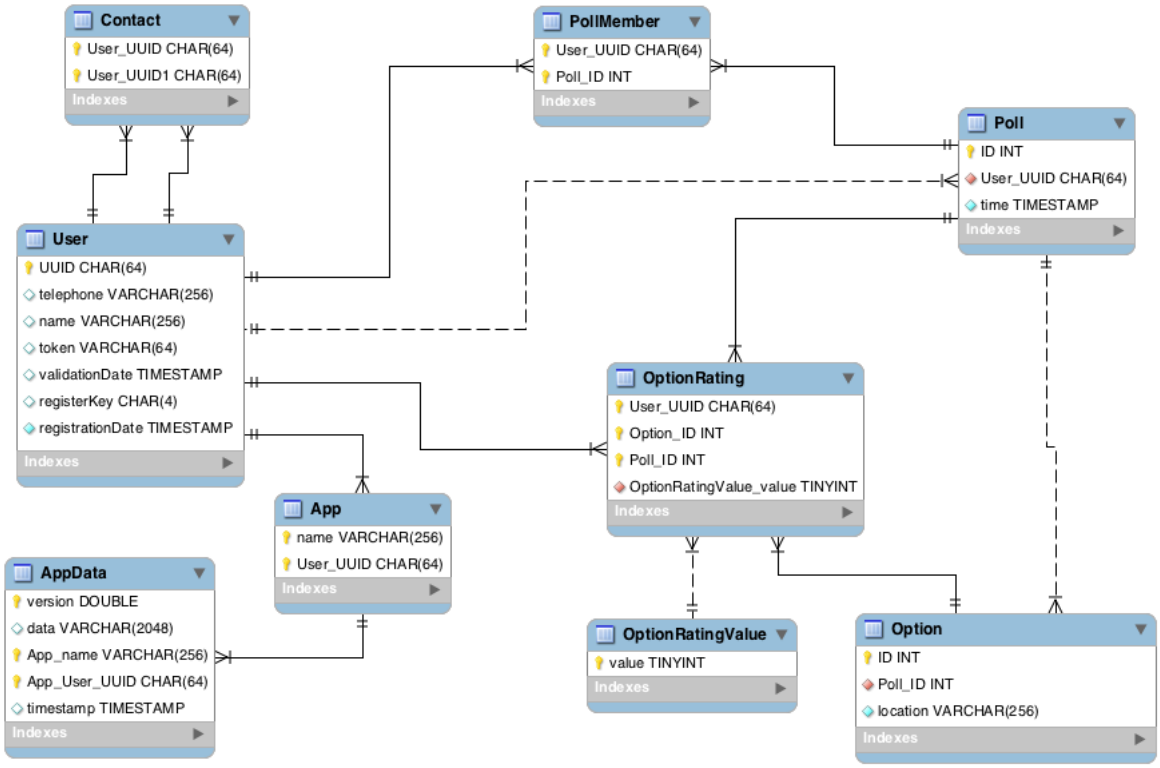

Abbildung 5.2.: MoCCha-EER-Model

5.2 ROA Entwurfsverfahren

Da es sich bei der REST-konformen Architektur, die für den Webservice benutzt wird um ROA handelt, wird für den Entwurf des ROA-Webservice das von Richardson und Ruby entwickelte generische ROA-Entwurfsverfahren benutzt. Dieses Verfahren hat wie bereits in Abschnitt 2.4.7 beschrieben folgendes Aussehen:

1. Figure out the data set
2. Split the data set into resources
   For each kind of resource:
3. Name the resources with URIs
4. Expose a subset of the uniform interface
5. Design the representation(s) accepted from the client
6. Design the representation(s) served to the client
7. Integrate this resource into existing resources, using hypermedia links and forms
8. Consider the typical course of events: whats supposed to happen?
9. Consider error conditions: what might go wrong?

Richardson und Ruby 2007, S. 148

Das Entwurfsverfahren wird in den folgenden Abschnitten angewandt um ein ROA-Webservice zu entwerfen.

5.2.1 Data-set

Der Punkt eins des Entwurfsverfahren zum Herausfinden der Daten und Informationen die ausgeliefert oder gespeichert werden sollen, wurde bereits in der Anforderungsanalyse (siehe Kapitel 4) sowie im Datenbankschema (siehe Abschnitt 5.1) bewerkstelligt. Anhand dieser Abschnitte wurde eine Übersicht über die Daten verschafft. Bei den Daten die nur ausgeliefert werden sollen handelt es sich um die Speisepläne der Mensen, Events und Kursepläne von Universitäten der TU Berlin und UdK. Die TU Berlin und UdK stellen Kategorien dar. Bei den Daten/Informationen die ausgeliefert und/oder gespeichert werden sollen, handelt es sich um Benutzer (user), Kontakte (contacts), Applikationsdaten (appData) und Umfragen (polls), siehe dazu Abschnitt 5.1.

5.2.2

Split the data

Nach dem Punkt 2 des Entwurfsverfahren sollen die Daten und Information, die gespeichert oder ausgeliefert werden, in Ressourcen aufgeteilt werden (siehe Abschnitt 2.4.7). Die Speisepläne der Mensen werden in zwei verschiedene Arten von Ressourcen aufgeteilt. Einmal in eine einmalige vordefinierte Ressource, die alle Mensen zurückliefert und in eine Ressource für jedes Objekt. Das Objekt in diesem Fall ist die Mensa, für die der Wochenspeiseplan zurückgeliefert wird. Die Events werden in drei Ressourcen aufgeteilt. Die erste Ressource ist eine einmalig vordefinierte Ressource, die alle Kategorien für Events zurückliefert. Die anderen beiden Ressourcen sind Objekt-Ressourcen. Die eine Ressource liefert alle Events der Kategorie, die zweite Ressource liefert ein bestimmtes Event der Kategorie. Für die Kurspläne sind drei Ressourcen nötig. Die erste Ressource ist wieder eine einmalige vordefinierte Ressource die die Kategorien zurückliefert, für die Kurspläne existieren. Die zweite und dritte Ressource ist eine Objekt-Ressource. Die zweite Ressource liefert für die bestimmte Kategorie alle Kurspläne und die dritte liefert ein bestimmten Kursplan für eine bestimmte Kategorie. Die Daten bzw. Information für einen Benutzer wurden in mehreren Ressourcen unterteilt. Die erste Ressource ist eine Objekt-Ressource, die für jeden Benutzer besteht. Für die Kontakte eines Benutzers gibt es eine einmalig vordefinierte Ressource, die alle Kontakte zurückliefern kann. Zudem eine weitere Ressource für jeden einzelnen Kontakt (eine Objekt-Ressource). Für jede Applikation des Benutzers existiert eine Ressource. Diese Ressource listet die existierenden Versionen der Applikation. Für jede Version existiert zudem auch eine Ressource. Diese Ressource ist auch eine Objekt-Ressource. Unter der Versions-Ressource sind die Applikationsdaten zu finden. Eine weitere einmalig vordefinierte Ressource ist die Validierungs-Ressource, die abhängig von der Benutzer-Ressource, den Status der Validierung liefert. Die Umfragen eines Benutzers sind durch eine einmalig vordefinierte Ressource dargestellt. Diese ist abhängig von der Benutzer-Ressource. Für jede Umfrage besteht auch eine eigene Ressource unterhalb der vordefinierten Umfragen-Ressource. Es handelt sich dabei auch um eine Objekt-Ressource. Die Optionen für eine Umfrage sind als einmalige vordefinierte Ressource unter jeder Umfrage gelistet. Jede Option einer Umfrage hat eine eigene Objekt-Ressource. Genauso hat jeder Teilnehmer einer Umfrage eine eigene Ressource. Die Daten und Information wurden in Ressourcen aufgespalten. In dem Abschnitt 5.2.3 werden diese Ressourcen mit URIs versehen bzw. benannt.

5.2.3

Name the resources

Als dritter Schritt im ROA-Entwurfsverfahren müssen die Ressourcen die zuvor in Abschnitt 5.2.2 definiert wurden, mit URIs versehen werden. Die URI einer Ressource ist deren Name und identifiziert diese, siehe dazu Abschnitt 2.4.1.

Da in dem Abschnitt 5.2.2 verschiedene Ressourcen definiert wurden, erhalten diese auch verschiedene Arten von URIs. Zum einen "fest" definierbare, innerhalb des Entwurfes, zum anderen URI-Templates, siehe Abschnitt 2.4.7. Die Events-Ressourcen werden durch folgende URIs identifiziert: "/events", "/events/{category}" und "/events/{category}/{id}". Die URI "/events" identifiziert die einmalig vordefinierte Event-Ressource. Die beiden anderen URIs besitzen Templates. Das Template "{category}" kann mit einer Event-Kategorie ersetzt werden und identifiziert dann die Eventlisten-Ressource für die bestimmte Kategorie. Das Template "{id}" kann mit einer Event-ID ersetzt werden. Die URI identifiziert dann das Event einer bestimmten Kategorie. Ähnlich ist es bei den Ressourcen für die Kurspläne. Dort identifizieren folgende URIs die Ressourcen: "/courses", "/courses/{category}" und "/courses/{category}/{id}". Hier wird das Template "{category}" wieder mit einer Kategorie ersetzt, aber mit der "Universitäten-Kategorie". Das Template "{id}" kann mit einer Kursplan-ID ersetzt werden.
Die URI "/courses" identifiziert die Ressource für die Kursplankategorien, die URI "/courses/{category}" identifiziert die Ressource für die Kurspläne einer Kategorie und die URI "/courses/{category}/{id}" identifiziert die Ressource für einen bestimmten Kursplan innerhalb einer Kategorie. Wie bereits in Abschnitt 5.2.2 beschrieben, besitzen die Speisepläne nur zwei Ressourcen. Die erste Ressource liefert die Mensen und wird über die URI "/canteens" identifiziert. Die zweite Ressource liefert den Wochenplan für eine bestimmte Mensa und wird über die URI "/canteens/{id}" identifiziert. Das Template "{id}" muss mit einer Mensa-ID ersetzt werden. Da jeder Benutzer eine Objekt-Ressource ist, wird er durch folgende URI identifiziert "/{UUID}". Das Template muss mit einer UUID ersetzt werden, die das Gerät des Benutzers identifiziert. Die Kontakte-Ressource wird durch die URI "/{UUID}/contacts" und die Kontakt-Ressource durch die URI '/{UUID}/contacts/teleHash" identifiziert, da sie vom jeweiligen Benutzer abhängig sind. Das selbe gilt für die Validierungs-Ressource die über die URI "/{UUID}/validation" identifiziert wird. Die Applikations-Ressource des Benutzers wird durch die URI "/{UUID}/{app}" identifiziert, wobei das Template "{app}" mit einem beliebigen Applikationsnamen ersetzt werden kann. Da für jede Version einer Applikation auch eine Ressource existiert, wird diese auch durch ein Template identifiziert: "/{UUID}/{app}/{version}". Das Template kann mit einer beliebigen Version einer Applikation ersetzt werden. Die Umfragen-Ressource ist innerhalb des Entwurfs wieder eindeutig definierbar, da sie eine Ressource des Typs eins ist (siehe Abschnitt 2.4.7). Die Ressource wird über die URI "/{UUID}/polls" identifiziert. Wie man an der URI erkennen kann ist sie auch von der Benutzer-Ressource abhängig. Da jeder Benutzer an anderen Umfragen (polls) teilgenommen hat. Die einzelne Umfragen-Ressource wird über die URI "/{UUID}/polls/{id}" identifiziert. Das Template "{id}" muss mit einer Umfragen-ID ersetzt werden. Eine weitere Ressource ist die Optionen-Ressource. Diese existiert für jede Umfrage-Ressource und ist, wie zuvor in Abschnitt 5.2.2 beschrieben, eine vordefinierte Ressource. Deshalb ist sie über die URI "/{UUID}/polls/{id}/options" identifizierbar. Da jede Option eine eigene Ressource besitzt und abhängig von einer Umfrage ist, wird die Option über folgende URI identifiziert: "/{UUID}/polls/{id}/options/{option_id}". Das Template "{option_id}" muss mit einer Option-ID ersetzt werden.

Jeder Teilnehmer einer Umfrage ist auch eine Ressource und wird durch die URI "/{UUID}/polls/{id}/{telehash}" identifiziert. Das Template "{telehash}" muss mit einer gehashten Telefonnummer eines Kontaktes ersetzt werden. Welche Methoden für die einzelnen Ressourcen bzw. URIs vorgesehen sind, folgt im Schritt vier des Entwurfsverfahrens, siehe dazu Abschnitt 5.2.4.

5.2.4

Expose uniform interface

In diesem Abschnitt wird der Schritt vier des Entwurfsverfahrens bewältigt. Für die zuvor definierten Ressourcen und deren URIs wird in diesem Schritt definiert welche HTTP-Methoden diese unterstützen sollen. Da die Ressourcen für die Speisepläne, Events und Kurspläne sogenannte Read-Only Ressourcen sind, ist dieser Schritt für diese überflüssig, siehe dazu Abschnitt 2.4.7. In dem Schritt vier müssen für jede Ressource fünf Fragen beantwortet werden. Abhängig von den Antworten werden bestimmte HTTP-Methoden untertstützt (siehe zur Erklärung Abschnitt 2.4.7). Ein Benutzer soll erstellt werden können und der Client kann über die resultierende URI entscheiden. Des Weiteren kann er gelöscht, aktualisiert und abgefragt werden. Anhand dieser Antworten wurde für die Benutzer-Ressource die Tabelle 5.1 erstellt. Sie zeigt welche Methoden die Benutzer-Ressource unterstützt, welche Funktionalität sie haben und unter welcher URI das beschriebene Resultat erreichbar ist. So kann z.B. ein Benutzer via PUT (und Repräsentation) an die URI, die anhand der UUID erstellt wird, erstellt werden. Danach kann ein Benutzer verändert, gelöscht oder abgefragt werden. Für die Validierungs-Ressource

Beschreibung	**Methode**	**Ressource**
Benutzer erstellen/aktualisieren	PUT	"/{UUID}"
Benutzer löschen	DELETE	"/{UUID}"
Benutzer abfragen	GET	"/{UUID}"

Tabelle 5.1.: Benutzer-Ressource - Methoden

soll das Erstellen möglich sein aber der Server soll die URI bestimmen können. Das Aktualisieren sowie die Abfrage der Validierungs-Ressource soll möglich sein aber das Löschen nicht. Daraus resultiert die folgende Tabelle 5.2.

Beschreibung	**Methode**	**Ressource**
Validierung erstellen	POST	"/{UUID}"
Validierung aktualisieren	PUT	"/{UUID}/validation"
Validierung abfragen	GET	"/{UUID}/validation"

Tabelle 5.2.: Validierungs-Ressource - Methoden

Die Validierung kann per POST an der Benutzer-Ressource erstellt werden und an der Validerungs-Ressource per PUT aktualisiert und per GET abgefragt werden. Dem Client soll es möglich sein einem Benutzer einen Kontakt hinzuzufügen. Die URI kann der Client selbst bestimmen. Der Kontakt kann gelöscht, abgefragt und aktualisiert werden. Die Tabelle 5.3 definiert aus den gegebenen Antworten der fünf Fragen die zu unterstützenden HTTP-Methoden für die Kontakt-Ressource. Um ein Kontakt zu erstellen muss an die beliebige Kontakte-Ressource eine PUT-Anfrage gesendet werden. Das Löschen, Aktualisieren und Abfragen ist auch direkt an der Kontakt-Ressource möglich. Mithilfe der Kontakte-Ressource kann der Client abfragen welche Kontakte ein Benut-

Beschreibung	**Methode**	**Ressource**
Kontakt erstellen	PUT	”/{UUID}/contacts/{telehash}”
Kontakt löschen	DELETE	”/{UUID}/contacts/{telehash}”
Kontakt abfragen	GET	”/{UUID}/contacts/{telehash}”

Tabelle 5.3.: Kontakt-Ressource - Methoden

zer besitzt. Diese Ressource unterstützt aber keine weiteren Methoden außer GET und HEAD. Die Applikations-Ressource und Applikationsversions-Ressource wurde in der Tabelle 5.4 zusammengefasst. Das Erstellen einer Applikation sowie einer Version, soll möglich sein und die Vergabe der URI ist dem Client überlassen. Das Löschen beider Ressourcen ist nicht möglich, jedoch das Abfragen und Aktualisieren. Die Umfragen-

Beschreibung	**Methode**	**Ressource**
Applikation erstellen/aktualisieren	PUT	”/{UUID}/{app}”
Applikationsversionen liefern	GET	”/{UUID}/{app}”
Version erstellen/aktualisieren	PUT	”/{UUID}/{app}/{version}”
Daten der Version liefern	GET	”/{UUID}/{app}/{version}”

Tabelle 5.4.: Applikations- und Applikationsversions-Ressource - Methoden

Ressource soll auch durch einen Client erstellt werden können. Die URI wird dabei aber vom Server bestimmt. Eine Umfrage soll löschbar, aktualisierbar und abfragbar sein. Die daraus resultierenden HTTP-Methoden sind in der Tabelle 5.5 dargestellt. Die

Beschreibung	**Methode**	**Ressource**
Umfrage erstellen	POST	”/{UUID}/polls”
Umfrage liefern	GET	”/{UUID}/polls/{id}”
Umfrage aktualisieren	PUT	”/{UUID}/polls/{id}”
Umfrage löschen	DELETE	”/{UUID}/polls/{id}”

Tabelle 5.5.: Umfragen-Ressource - Methoden

Teilnehmer-Ressource ist abhängig von der Umfragen-Ressource.

Zudem ist diese Ressource erstellbar, löschbar, aktualisierbar und abfragbar. Der Client darf bei der Teilnehmer-Ressource die URI bestimmen. Diese wird aus der Telefonnummer *hash* des Kontaktes erstellt. Die unterstützten HTTP-Methoden sind in der Tabelle 5.6 dargestellt. Die Options-Ressource ist wie die Teilnehmer-Ressource abhängig von

Beschreibung	**Methode**	**Ressource**
Teilnehmer hinzufügen/aktualisieren	PUT	"/{UUID}/polls/{id}/{telehash}"
Teilnehmer liefern	GET	"/{UUID}/polls/{id}/{telehash}"
Teilnehmer löschen	DELETE	"/{UUID}/polls/{id}/{telehash}"

Tabelle 5.6.: Teilnehmer-Ressource - Methoden

der Umfrage. Sie ist gleichermaßen löschbar, aktualisierbar sowie abfragbar. Der einzige Unterschied besteht in der Erstellung der Ressource. Wenn eine Option vom Client erstellt wird, ist der Server berechtigt die URI für die Options-Ressource zu bestimmen, anders bei der Teilnehmer-Ressource. Aus den angegebenen Eigenschaften resultiert die Tabelle 5.7 und die dargestellten unterstützten HTTP-Methoden der Ressource. Für die

Beschreibung	**Methode**	**Ressource**
Option hinzufügen	POST	"/{UUID}/polls/{id}/options"
Option aktualisieren	PUT	"/{UUID}/polls/{id}/options/{optionID}"
Option löschen	DELETE	"/{UUID}/polls/{id}/options/{optionID}"
Option liefern	GET	"/{UUID}/polls/{id}/options/{optionID}"

Tabelle 5.7.: Option-Ressource - Methoden

Ressourcen die aktualisiert bzw. erstellt werden können, ist meist eine Repräsentation innerhalb der Anfrage im *entity body* notwendig. Das Aussehen sowie das Format, was vom Client gesendet und vom Server akzeptiert wird, wird in dem Abschnitt 5.2.5 definiert.

5.2.5 Accepted representations

Wie bereits zuvor erwähnt werden in dieser Abschnitt die Repräsentationen der Ressourcen definiert, die vom Client gesendet werden, um ein *resource state* zu verändern. Das Definieren der Repräsentationen die vom Client akzeptiert werden, ist der fünfte Schritt des ROA-Entwurfsverfahrens. Um eine Benutzer-Ressource zu aktualisieren oder zu erstellen muss die Repräsentation aus Beispiel 5.1 vom Client übermittelt werden. Das Datenaustauschformat der Repräsentation ist: JSON. Der Client kann auch ohne diese Repräsentation einen Benutzer erstellen, da die UUID bzw. Scoping-Information sich innerhalb der URI befindet. Die Repräsentation aus Beispiel 5.1 beinhaltet alle Eigenschaften die für den Benutzer gesetzt werden können.

Um eine Validierungs-Ressource zu erstellen, ist eine Repräsentation im Form-Encoded Datenaustauschformat notwendig, siehe dazu Abschnitt 2.2.4.

```
{
  "name":"USER-NAME",
  "teleHash":"TELE-Hash",
  "token":"TOKEN"
}
```

Beispiel 5.1: Akzeptierte Benutzer-Repräsentation

Die Repräsentation sollte das folgende Aussehen besitzen: "tele=TELEFONNUMMER" und sich bei der Anfrage im *entity body* befinden. Um einen Kontakt hinzuzufügen ist keine Repräsentation notwendig, da die Scoping-Information sich bereits in der URI befindet, siehe Abschnitt 5.2.4.

```
{
  "data":"DATA"
}
```

Beispiel 5.2: Akzeptierte Applikationsversion-Repräsentation

Das selbe gilt für das Erstellen einer neuen Applikation eines Benutzers. Dort ist wieder die nötige Information innerhalb der URI "/{uuid}/{app}". Bei der Erstellung einer Applikationsversion und der dazugehörigen Applikationsdaten wird nur JSON als Datenaustauschformat akzeptiert. Innerhalb der Versions-Repräsentation befinden sich die Applikationsdaten, siehe Beispiel 5.2. Es ist auch möglich eine Version ohne Applikationsdaten zu erstellen, da die Version innerhalb der URI übermittelt wird. Um eine Umfrage zu erstellen ist wieder eine Repräsentation im JSON Datenaustauschformat notwendig. Das Aussehen der JSON-Repräsentation ist in dem Beispiel 5.3 definiert.

```
{
  "member":[{"contact":"TELE-HASH"},...],
  "time":"UNIX-Timestamp",
  "options":[{"option":"CANTEEN","rating":"(1-10)"},...]
}
```

Beispiel 5.3: Akzeptierte Umfrage-Repräsentation

Beim Erstellen oder Aktualisieren einer Umfrage können beliebig viele Teilnehmer und Optionen innerhalb eines JSON-Arrays gesetzt werden. Die Teilnehmer besitzen jeweils nur eine Telefonnummer als hash. Die Optionen besitzen eine Mensa als "option" und eine dazugehörige Bewertung ("rating"). Die Bewertung einer Option ist eine natürliche Zahl zwischen 1 und 10. Da eine Umfrage erstellt wird um sich z.B. für ein Mittagessen in einer bestimmten Mensa zu verabreden, ist eine Zeit für die Verabredung notwendig. Diese Zeit muss als "time" im UNIX-Timestamp Format gesendet werden. Nachdem eine Umfrage erstellt wurde, kann ein Teilnehmer hinzugefügt werden. Dies erfordert keine Repräsentation, da die Scoping-Information innerhalb der URI ist. Um jedoch eine Option einer Umfrage hinzuzufügen muss eine Options-Repräsentation im JSON Datenaustauschformat vom Client gesendet werden. Die Repräsentation die vom Client als Options-Repräsentation erwartet wird, ist in dem Beispiel 5.4 definiert.

```
{
  "option":"OPTION",
  "rating":"RATING(1-10)"
}
```

Beispiel 5.4: Akzeptierte Option-Repräsentation

Dabei steht, wie zuvor bei dem Beispiel 5.3 erklärt, ”option” für die Mensa und ”rating” für die Bewertung der Option (zwischen 1 und 10). Die Repräsentationen die vom Client akzeptiert werden sind nicht so komplex wie diese, die dem Client ausgeliefert werden. Die Repräsentationen die dem Client geliefert werden, werden in dem Abschnitt 5.2.6 definiert.

5.2.6 Served representations

In diesem Abschnitt werden die Repräsentationen definiert, die den derzeitigen *resource state* darstellen. Das Definieren der Repräsentationen die an dem Client ausgeliefert werden, gehört zu dem Schritt sechs des ROA-Entwurfsverfahrens. Dieser wird aber mit dem Schritt sieben verbunden. Denn der Schritt sieben beinhaltet das Verlinken von den existierenden Ressourcen innerhalb ihrer Repräsentationen, siehe zur Erklärung Abschnitt 2.4.7. Die Repräsentation beinhaltet nicht nur die Eigenschaften eines Benutzers (innerhalb des JSON-Array's ”properties”) sondern auch die Referenzen auf benachbarte Ressourcen bzw. auf andere *resource states*. Dies ermöglicht das Erfüllen der HATEOAS-REST-Bedingung (siehe Abschnitt 2.4.7) bzw. des Level drei des ”Richardson Maturity Model” (siehe Abschnitt 2.3.5). Die Repräsentationen werden ausschließlich im JSON Datenaustauschformat ausgeliefert, siehe Abschnitt 2.2.3.

```
{
  "name":"USER-NAME","teleHash":"TELE-Hash","token":"TOKEN",
  "update/create":{
    "uri":"/{uuid}",
    "methode":"put",
    "type":"application/json",
    "parameter":{
      "name":"USER-NAME",
      "teleHash":"TELE-Hash",
      "token":"TOKEN"
    }
  },
  "delete":{
    "uri":"/{uuid}",
    "method":"delete"
  },
  "polls":"/{uuid}/polls",
  "contacts":"/{uuid}/contacts",
  "canteens": "/canteens",
  "eventCategories": "/events",
  "courseCategories": "/courses",
  "validation":"/{uuid}/validation",
  "apps":[{"app":"APP","uri":"/{uuid}/{app}"},...],
  "addApp":{"uri":"/{uuid}/{app}","method":"put"}
  "addAppVersion-Data":{"uri":"/{uuid}/{app}/{version}",
    "method":"put"}
}
```

Beispiel 5.5: Ausgelieferte Benutzer-Repräsentation

Innerhalb der JSON-Repräsentation wird somit durch ”create/update” mitgeteilt das dieses JSON-Objekt Informationen besitzt um eine Benutzer-Ressource zu erstellen oder zu aktualisieren. Die Informationen innerhalb dieser Art von JSON-Objekt sind immer gleich. Wenn z.B. ein anderes JSON-Objekt signalisiert das es Informationen, wie man z.B. eine Benutzer-Ressource löscht, besitzt, dann beinhaltet dieses JSON-Objekt die *keys* ”uri” und ”method”. Der *key* ”uri” besitzt als *value* den Pfad zu der Ressource.

Der *key* "method" besitzt die Methode als *value*, mit der z.B. eine Benutzer-Ressource gelöscht werden kann. Diese *key-value* Paare sind somit immer enthalten. Weitere mögliche Paare sind "parameter" und "type". Der *key* "parameter" beinhaltet die Parameter die nötig sind um die Anfrage auszuführen und "type" das Datenaustauschformat der Parameter. Also kann laut der Repräsentation, mithilfe einer DELETE-Anfrage an die URI "/{uuid}" eine Benutzer-Ressource gelöscht werden. Das Template wird dabei dann jeweils für die bestimmte Benutzer-Ressource ausgetauscht, von der die Repräsentation geliefert wurde. Die *key-value* Paare "polls", "contacts" und "validation" besitzen die URI als *value* zu den besagten Ressourcen, die sie als *key* besitzen.

```
{
  "contacts":[{"teleHash":"TELE-HASH","uri":"HTTP-URI","name":"CONTACT-NAME"},...],
  "create":{"method":"put","uri":"/contacts/{teleHash}"},
  "self":"/{uuid}"
}
```

Beispiel 5.6: Ausgelieferte Kontakte-Repräsentation

Das Beispiel 5.6 stellt die Repräsentation für die Kontakte-Ressource dar. Die Repräsentation beinhaltet unter anderem die Kontaktliste als JSON-Array. Aber auch die Information wie man einen Kontakt erstellt und wie man zur Benutzer-Ressource zurückgelangt. Das Beispiel 5.7 beinhaltet die Repräsentation einer Kontakt-Ressource. Die Repräsentation des Kontaktes beinhaltet den Namen sowie den Telefonnummer *hash* des Kontaktes. Außerdem die Information wie man einen Kontakt löschen und erstellen kann. Die *key-value* Paare "self" und "contacts" beinhalten Verweise auf andere verwandte Ressourcen. Wie bereits zuvor schon in Beispiel 5.6 beschrieben, besitzt "self" immer den Verweis auf die Benutzer-Ressource als *value*.

```
{
  "name":"VALUE",
  "teleHash":"TELE-HASH",
  "create":{"method":"put","uri":"/contacts/{teleHash}"},
  "delete":{"method":"delete","uri":"/contacts/{teleHash}"},
  "contacts":"/{uuid}/contacts",
  "self":"/{uuid}"
}
```

Beispiel 5.7: Ausgelieferte Kontakt-Repräsentation

Die Ressource Validierung ermöglicht das Abfragen des Validierungsstatuses. Die Repräsentation der Validierung beinhaltet aber nicht nur den Status sondern, wie die anderen Repräsentation, auch eine Referenz auf die Benutzer-Ressource. Falls der Benutzer noch nicht validiert wurde, beinhaltet die Repräsentation noch Informationen wie der Benutzer validiert werden kann, siehe dazu Beispiel 5.8.

```
{
  "status":"validated/not validated",
  //FALLS nicht validiert
  //"validate":{
  //  "method":"post",
  //  "uri":"/{uuid}/validation",
  //  "type":"application/x-www-form-urlencoded",
  //  "parameter":"tele:017...."}
  "self":"/{uuid}"
}
```

Beispiel 5.8: Ausgelieferte Validierungs-Repräsentation

Das Beispiel 5.9 zeigt die ausgelieferte Applikations-Repräsentation. Diese beinhaltet alle Applikationsversionen sowie die Referenzen zu diesen Ressourcen.

Die Repräsentation liefert des Weiteren Informationen wie man eine Applikationsversion erstellen kann und die Referenz zu der Benutzer-Ressource.

```
{
  "versions":[{"version":1.0, "uri":"/{uuid}/{app}/1.0"},...],
  "addVersion":{
    "uri":"/{uuid}/{app}/{version}",
    "method":"put"},
  "self":"/{uuid}"
}
```

Beispiel 5.9: Ausgelieferte Applikations-Repräsentation

Die Information zur Erstellung einer Applikationsversions-Ressource wurde nur auf PUT ohne Repräsentation beschränkt, da nachträglich die Applikationsdaten auch noch gesetzt werden können. Die Informationen dazu sind innerhalb der Applikationsversions-Repräsentation verfügbar, siehe Beispiel 5.10. Die Applikationsversions-Repräsentation beinhaltet die Applikationsdaten und wie bereits zuvor beschrieben, die Information wie man diese aktualisieren kann. Der *key* "timestamp" innerhalb der Repräsentation, siehe Beispiel 5.10, beinhaltet die Zeit als *value* wann die Applikationsdaten der Version das letzte mal aktualisiert wurden. Innerhalb der Repräsentation wird auf die Benutzer-Ressource und die Applikations-Ressource mithilfe der URIs verwiesen.

```
{
  "timestamp":"SAVE-TIME",
  "data":"DATA-AS-STRING",
  "updateData":{
    "uri":"/{uuid}/{app}/{version}",
    "method":"put",
    "type":"application/json",
    "parameter":{
      "data":"DATA"
    }},
  "appVersions":"/{uuid}/{app}",
  "self":"/{uuid}"
}
```

Beispiel 5.10: Ausgelieferte Applikationsversions-Repräsentation

Die Repräsentation für die Umfragen-Ressource ist in Beispiel 5.11 definiert. Die Repräsentation beinhaltet die Liste der zugehörigen Umfragen eines Benutzers. Auf diese Umfragen wird mithilfe von Links verwiesen. Des Weiteren beinhaltet die Repräsentation den *key* "self" der, wie zuvor beschrieben, immer auf den jeweiligen Benutzer verweist und die Information wie man eine Umfrage (poll) erstellen kann.

```
{
  "polls":[{"id":"POLL-ID", "time":"UNIX-Timestamp","uri":"/{uuid}/polls/{id}"},...],
  "create":{
    "method":"post",
    "uri":"/{uuid}/polls",
    "type":"application/json",
    "parameter":{
      "member":[{"contact":"TELE-HASH"},...],
      "time":"UNIX-Timestamp",
      "options":[{"option":"CANTEEN","rating":"(1-10)"},...]
    }
  },
  "self":"/{uuid}"
}
```

Beispiel 5.11: Ausgelieferte Umfragen-Repräsentation

Wie bereits zuvor beschrieben wird innerhalb der Umfragen-Repräsentation auf die einzelnen Umfrage-Ressourcen verwiesen. Die allgemeine Repräsentation für diese Ressource ist in Beispiel 5.12 dargestellt.

```
{
  "options":[{
    "option":"VALUE",
    "uri":"HTTP-URI",
    "memberRating":[{"name":"NAME","rating":"RATING","uri":"/{uuid}/polls/{id}/{TELE-HASH}"},...],
    ...
    }],
  "member":[{"name":"NAME","uri":"HTTP-URI"},...],
  "addMember":{"uri":"/{uuid}/polls/{id}/{TELE-HASH}","method":"put"},
  "addOption":{"uri":"/{uuid}/polls/{id}/options",
    "method":"post",
    "type":"application/json",
    "parameter":{
      "option":"OPTION",
      "rating":"(1-10)"
    }},
  "delete":{"uri":"/{uuid}/polls/{id}", "method":"delete"},
  "update":{"uri":"/{uuid}/polls/{id}",
    "method":"put",
    "type":"application/json",
    "parameter":{
      "member":[{"contact":"TELE-HASH"},...],
      "time":"UNIX-Timestamp",
      "options":[{"option":"CANTEEN"},...]
    }
  },
  "polls":"/{uuid}/polls",
  "self":"/{uuid}"
}
```

Beispiel 5.12: Ausgelieferte Umfrage-Repräsentation

Die ersten Informationen sind die zugehörigen Optionen und Teilnehmer und deren Optionsbewertungen bzw. deren Namen und dazu die Verlinkungen zu den Ressourcen. Die weiteren JSON-Objekte stellen die Informationen dar die benötigt werden um einen Teilnehmer oder eine Option der Umfrage hinzuzufügen. Des Weiteren die Informationen zum Löschen oder Aktualisieren dieser Umfrage. Werden bei diesen Vorgehensweisen Parameter im JSON Format gefordert, dargestellt mit dem *key-value* Paar "type", dann wird die geforderte JSON-Repräsentation als JSON-Objekt "parameter" mit ausgeliefert. Der *key* "self" ist wie bei jeder Repräsentation vorhanden. Jedoch existiert in dieser Repräsentation noch das *key-value* Paar "polls" das auf die Umfragen-Ressource verweist. In dem Beispiel 5.13 ist die Repräsentation für die Teilnehmer-Ressource dargestellt.

```
{
  "name":"NAME",
  "options":[{"id":"ID","uri":"HTTP-URI","rating":"RATING"},...],
  "delete":{"uri":"/{uuid}/polls/{id}/{TELE-HASH}","method":"delete"},
  "create":{"uri":"/{uuid}/polls/{id}/{TELE-HASH}","method":"put"},
  "poll":"/{uuid}/polls/{id}",
  "self":"/{uuid}"

}
```

Beispiel 5.13: Ausgelieferte Teilnehmer-Repräsentation

Sie beinhaltet wie man einen Teilnehmer erstellen oder löschen kann und die Verlinkungen zu der zugehörigen Benutzer- und Umfrage-Ressource. Die Repräsentation enthält des Weiteren einen Namen für den Teilnehmer, sowie seine abgegebenen Bewertungen für die existierenden Optionen, zu denen auch verwiesen wird. Die Informationen wie man eine Option erstellt ist in der Optionen-Repräsentation enthalten, siehe Beispiel 5.14. In dieser Repräsentation sind auch die Optionen als JSON-Array enthalten, die zur Umfrage dazugehören. Die *keys* "self" und "poll" beinhalten Verweise, zu den nächsthöheren Ressourcen, als *value*. Genauer zu der Umfrage- und Benutzer-Ressource.

```
{
  "options":[{"option":"VALUE","rating":"RATING","uri":"/{uuid}/polls/{id}/options/{id}"}],
  "create":{"uri":"/{uuid}/polls/{id}/options",
    "method":"post",
    "type":"application/json",
    "parameter":{
      "option":"OPTION",
      "rating":"(1-10)"
    }},
  "poll":"/{uuid}/polls/{id}",
  "self":"/{uuid}"
}
```

Beispiel 5.14: Ausgelieferte Optionen-Repräsentation

Die Option-Repräsentation beinhaltet nicht nur diese Verlinkungen sondern auch die Verlinkung zur Optionen-Ressource (siehe Beispiel 5.15). Die Repräsentation liefert die Informationen um welche Option es sich handelt, wie der Benutzer diese Option bewertet hat und wie andere Teilnehmer der Umfrage diese Option bewertet haben. Für jede Bewertung eines Teilnehmers existiert der Verweis zur dazugehörigen Teilnehmer-Ressource. Sodass diese auch direkt abgefragt werden können. Die Repräsentation beinhaltet dazu noch Informationen wie man eine Option erstellt, löscht oder aktualisiert.

```
{
  "option":"VALUE","rating":"RATING",
  "memberRating":[{"name":"NAME","rating":"RATING","uri":"/{uuid}/polls/{id}/{TELE-HASH}"},...],
  "delete":{"uri":"/{uuid}/polls/{id}/options/{id}","method":"delete"},
  "update":{"uri":"/{uuid}/polls/{id}/options/{id}",
    "method":"put",
    "type":"application/json",
    "parameter":{
      "option":"OPTION",
      "rating":"(1-10)"
  }},
  "create":{"uri":"/{uuid}/polls/{id}/options",
    "method":"post",
    "type":"application/json",
    "parameter":{
      "option":"OPTION",
      "rating":"(1-10)"
    }},
  "options":"/{uuid}/polls/{id}/options",
  "poll":"/{uuid}/polls/{id}",
  "self":"/{uuid}",
}
```

Beispiel 5.15: Ausgelieferte Option-Repräsentation

Da die Daten von den Parsern des Webservice stammen, wurden die nachfolgenden Ressourcen: Speisepläne, Kurspläne und Events vom bestehenden Webservice übernommen. Die Repräsentationen haben nur kleine Anpassungen erhalten. Diese werden in den folgenden Absätzen dargestellt. Die Ressourcen die dazu entworfen wurden wie z.B. Eventkategorien, werden komplett neu mit ihren Repräsentation definiert, siehe dazu Beispiel 5.16. Die Repräsentation für die Eventkategorien-Ressource, liefert alle existierenden Kategorien und die dazu gehörigen URIs zu den Ressourcen. Um die Verbindung zu anderen Ressourcen zu ermöglichen, enthält die Repräsentation die URIs zu den Kurskategorien und Mensen Ressourcen. Des Weiteren beinhaltet die Repräsentation die Information wie man einen Benutzer erstellt. So kann der Client seinen *application state* bzw. auch den *resource state* verändern. Zur Erklärung siehe Abschnitt 2.4.4 oder 2.3.5.

```
{
  "categories":[{"category":"CATEGORY","uri":"/events/{CATEGORY}"},...],
  "createUser":{"uri":"/{uuid}","method":"put"},
  "courseCategories": "/courses",
  "canteens": "/canteens"
}
```

Beispiel 5.16: Ausgelieferte Eventkategorien-Repräsentation

Die Verlinkungen innerhalb der Eventkategorien verweisen auf die jeweilige Kategorie-Ressource. Diese Ressource enthält in der Repräsentation eine Liste von Events, die der Kategorie angehören (siehe Beispiel 5.17). Es wird auf jedes einzelne Event referenziert. Die Repräsentation beinhaltet außerdem eine Referenz auf die Eventkategorien-Ressource.

```
{
  "events":[{"title":"NAME","uri":"/events/{CATEGORIES}/{id}"},...],
  "eventCategories":"/events"
}
```

Beispiel 5.17: Ausgelieferte Eventkategorie-Repräsentation

Wie bereits zuvor erwähnt wird die Repräsentation eines Events vom bestehenden Webservice übernommen. Im Beispiel 5.18 ist die einzige Anpassung zu sehen. Das ermöglicht die Verbindung zur vorhergehenden Eventkategorie-Ressource, die die Eventliste als Repräsentation beinhaltet.

```
{
  //OLD EVENT JSON
  "events":"/events/{CATEGORIES}"
}
```

Beispiel 5.18: Ausgelieferte Event-Repräsentation

Ähnliche Repräsentationen existieren für die Kursplan-Ressourcen, siehe Beispiel 5.19, 5.20 und 5.21.

```
{
  "categories":[{"category":"CATEGORY","uri":"/courses/{CATEGORY}"},...],
  "createUser":{"uri":"/{uuid}","method":"put"},
  "eventCategories": "/events",
  "canteens": "/canteens"
}
```

Beispiel 5.19: Ausgelieferte Kurskategorien-Repräsentation

```
{
  "courses":[{"title":"NAME","uri":"/courses/{CATEGORY}/{ID}"},...],
  "categories":"/courses"
}
```

Beispiel 5.20: Ausgelieferte Kurskategorie-Repräsentation

```
{
  //OLD COURSE JSON
  "courses":"/courses/{CATEGORIES}"
}
```

Beispiel 5.21: Ausgelieferte Kurs-Repräsentation

Bei den Mensen-Ressourcen existieren keine Kategorien. Somit existieren dort auch nur zwei verschiedene Ressourcen. Die Mensen-Ressource enthält als Repräsentationen alle existierenden Mensen auf die auch verwiesen wird (siehe Beispiel 5.22).

Wie bei den Kurskategorien oder Eventkategorien enthält diese Repräsentation die URIs zu den anderen Ressourcen und Information wie man einen Benutzer erstellt. Dies bietet dem Client die Möglichkeit, wie bereits zuvor erwähnt, von dieser Ressource auch an andere Informationen bzw. zu einem anderen *application state* zu gelangen.

```
{
  "canteens":[{"canteen":"CANTEEN","uri":"/canteens/{id}"},...],
  "createUser":{"uri":"/{uuid}","method":"put"},
  "courseCategories": "/courses",
  "eventCategories": "/events"
}
```

Beispiel 5.22: Ausgelieferte Mensen-Repräsentation

Die Mensa-Ressource liefert als Repräsentation den Wochenspeiseplan. Wie bereits zuvor erwähnt wird in Beispiel 5.23 nur die Veränderung des JSON abgebildet, da der Rest der Repräsentation vom alten Webservice bzw. Parser stammt. Die Repräsentation beinhaltet die Information wie man eine Umfrage erstellt, die als Option diese Mensa erhält. Außerdem beinhaltet diese die Verlinkung zu der Mensen-Ressource.

```
{
  //OLD CANTEEN JSON
  "createPoll":{
    "method":"post",
    "uri":"/{uuid}/polls/",
    "type":"application/json",
    "parameter":{
        "member":[{"contact":"TELE-HASH"},...],
        "time":"UNIX-TIMESTAMP",
        "options":[{"option":"CANTEEN","rating":"(1-10)"},...]
      }},
  "canteens":"/canteens"
}
```

Beispiel 5.23: Ausgelieferte Mensa-Repräsentation

Nach dem Entwurf der Repräsentationen, die vom Client akzeptiert und an diesem ausgeliefert werden, muss definiert werden was bei einer erfolgreichen Anfrage (siehe Abschnitt 5.2.7) und was bei einem Fehler (siehe Abschnitt 5.2.8) geschieht.

5.2.7 Supposed to Happen

Dieser Abschnitt repräsentiert den Schritt acht des ROA-Entwurfsverfahrens. In diesem Schritt wird die nachfolgende Aktion auf eine erfolgreiche Anfrage eines Clients, die allen Anforderungen entsprach, festgelegt. D.h. es wird definiert welche Statuscodes und welche HTTP-*Header* dem Client als Antwort gesendet werden. Wenn ein Client eine Ressource erfolgreich erstellt hat, entweder via POST oder PUT, dann soll als Antwort ein HTTP-Statuscode 201 (”Created”) und ein Location-*Header* gesendet werden. Der Location-*Header* beinhaltet die URI zu der erstellten neuen Ressource. Wird eine Ressource aktualisiert, via PUT oder gelöscht durch eine DELETE-Anfrage, soll der HTTP-Statuscode 200 (”OK”) gesendet werden. Sendet der Client eine GET-Anfrage an eine Ressource und beinhaltet diese keinen If-None-Match-*Header* so soll die Repräsentation der Ressource und der HTTP-Statuscode 200 (”OK”) gesendet werden.

Ist aber der *Header* vorhanden und der Etag stimmt überein dann soll als Antwort keine Repräsentation gesendet werden. Sondern nur der HTTP-Statuscode 304 ("Not Modified"). Falls der Etag aber nicht übereinstimmt soll die Repräsentation der Ressource, der HTTP-Statuscode 200 ("OK") sowie der neue Etag-*Header* gesendet werden. Siehe zur Erklärung 2.4.6. Eine HEAD-Anfrage soll sich genau wie eine GET-Anfrage verhalten. Nur das dabei die Repräsentationen nicht gesendet werden. Da die Daten der Ressourcen für Events einer Kategorie, für den Speiseplan einer Mensa sowie die Kurspläne einer Kategorie von einem Parser stammen und dieser regelmäßig um die gleiche Zeit gestartet wird, sollen bei GET-Anfragen auf die Ressourcen als Antwort noch ein Expires-HTTP-*Header* gesendet werden. Der *Header* beinhaltet die Zeit wann sich wahrscheinlich die Daten/Informationen wieder ändern. Nach dem die erfolgreichen Fälle definiert wurden bzw. das Verhalten darauf, muss definiert werden welche Antwort bei einem Fehler gesendet wird und welche Fehler eventuell existieren. Diese Definition folgt in dem Abschnitt 5.2.8.

5.2.8

What goes wrong

In diesem Abschnitt werden die möglichen Fehler sowie das darauf folgende Verhalten definiert. Dies ist somit der letzte Schritt des ROA-Entwurfsverfahrens. Die erwarteten Fehler auf eine Client-Anfrage sind in der Tabelle 5.8 zusammengefasst. Wenn ein Client

Fehler	**Antwort-Statuscode**
Ressource existiert nicht	404
Fehlerhafte Repräsentation	400
Falscher media type	415
Falls Methode nicht unterstützt wird	405
Falls Client falschen media type erwartet	406
Unerwarteter/Interner Fehler	500

Tabelle 5.8.: Erwartete Fehlanfragen

an eine nicht existierende Ressource eine Anfrage sendet muss als Fehlerbehandlung der Statuscode 404 ("Not Found") gesendet werden. Existiert die Ressource, aber der Client hat bei der Anfrage eine falsche Repräsentation oder diese im falschem Format gesendet, soll als Fehlerbehandlung der Statuscode 400 ("Bad Request") oder 415 ("Unsupported MediaType") gesendet werden. Sendet der Client eine Anfrage an eine existierende Ressource, mit einer nicht unterstützten HTTP-Methode, so soll als Antwort der Statuscode 405 ("Method not Allowed") gesendet werden. Falls der Client eine anderes Datenaustauschformat erwartet und dieses durch den Accept-*Header* bei der Anfrage dem Server mitteilt, so soll als Statuscode 406 ("Not Acceptable") gesendet werden. Diese Statuscodes sollen den Client darauf hinweisen das es bei seiner Anfrage zu einem Fehler kam.

Der Fehler wird mithilfe des Statuscodes identifiziert. Dem Client soll es durch diese Information möglich sein seine Anfrage zu ändern und diese erneut zu senden, sodass diese dann erfolgreich verarbeitet werden kann. Es ist durchaus möglich das ein Fehler auftritt der nicht erwartet wird. Oder das innerhalb des Servers ein Fehler auftritt. Als Beispiel wäre eine Java *Exception*. Dies soll dem Client anhand des Statuscodes 500 ("Internal Server Error") mitgeteilt werden.

5.2.9

Ergebnis

Durch Abschließen des Schrittes neun des ROA-Entwurfsverfahrens, wurde der neue MoCCha-Webservice in allen nötigen Details entworfen. Dadurch kann er in Kapitel 6 nach dem Entwurf implementiert werden. Der in diesem Kapitel entworfene Webservice, ist ohne nötigen Mehraufwand für einen Client leicht zu benutzen. Der Webservice bzw. deren Ressourcen sind verbunden, denn die Ressourcen referenzieren untereinander. Er ist selbstbeschreibend. Innerhalb der Repräsentationen wird gezeigt welche *resource states* existieren und wie sie zu erreichen sind. Er ermöglicht das *caching*, mithilfe des Expires-*Header* sowie das Einsparen von Bandbreite anhand der conditional GET-Methode. Jede Ressource besitzt ihre eigene URI. Zudem ist eine Ressource durch verschiedene HTTP-Methoden und deren Repräsentationen in andere *resource states* versetzbar. Jede Ressource wird durch eine eigene Repräsentation dargestellt. Die Bedingung des *uniform interface* wird durch diese Eigenschaften erfüllt. Dadurch das auch kein *application state* des Clients auf dem Server existiert, ist dieser somit auch stateless. Zusammenfassend ist der Webservice nach diesem Entwurf *RESTful* und resource-oriented.

Ressourcen

Dieser Abschnitt dient nur zum Vergleich der bestehenden Ressourcen des neuen und alten Webservices. Wie in der rechten Abbildung zu erkennen ist, ist dieser scheinbar kompakter, besitzt jedoch mehr Ressourcen als der vorherige Webservice. Denn der alte Webservice besaß zumeist nur Ressourcen des Typ eins, d.h. einmalig, vordefinierte Ressourcen. Der neue hingegen besitzt dazu viele Ressourcen des Typ zwei, die sogenannten Objekt-Ressourcen. Dieser Ressourcen Typ ist so gesehen unendlich, da für jedes Objekt eine Ressource existiert. Durch die sichtbare Hierarchie des zweiten Webservices ist es auch möglich auf die vorhergehenden bzw. benachbarten Ressourcen zu verweisen. Durch diese Referenzierung auf andere Ressourcen wird die HATEOAS-Bedingung erfüllt und das Level drei des "Richardson Maturity Model" erreicht.

VORHER:

```
/CampusCharlottenburg
├── /polls
├── /device
├── /info
├── /courses
├── /authenticate
├── /appID
├── /appData
│   └── /v1
│       └── /*
├── /get
│   ├── /staatsoperCatalog
│   ├── /canteenList
│   ├── /courses
│   └── /doodle
├── /post
│   ├── /friends
│   ├── /doodle
│   └── /logs
├── /user
│   ├── /register
│   ├── /updateDisplayName
│   ├── /validateRegistration
│   └── /contacts
└── /files
    └── /*
```

NACHHER:

```
/v2
├── /users
│   └── /{UUID}
│       ├── /{app}
│       │   └── /{version}
│       ├── /validate
│       ├── /contacts
│       │   └── /{telehash}
│       └── /polls
│           └── /{ID}
│               ├── /{telehash}
│               └── /options
│                   └── /{ID}
├── /canteens
│   └── /{ID}
├── /events
│   └── /{category}
│       └── /{ID}
└── /courses
    └── /{category}
        └── /{ID}
```

5.3 Testentwurf

Da der zu erstellende Webservice auch lauffähig sein soll, müssen bestimmte Tests durchgeführt werden. Zum einen Komponententests die jede Ressource einzeln testen. Die Tests überprüfen ob die unterstützten bzw. implementierten HTTP-Methoden funktionieren und geeignet auf Fehler reagieren, wie es im Entwurf in dem Abschnitt 5.2.8 definiert wurde. Die andere Art von Tests sind die Integrationstests. Diese werden benutzt um zu überprüfen ob die Ressourcen auch miteinander arbeiten bzw. funktionieren.

Es wird anhand dieser Tests überprüft ob z.B. eine Option-Ressource, die an der Optionen-Ressource via POST erstellt wurde, auch existiert. Sowie ob sie innerhalb der Repräsentation der Optionen-Ressource als existierende Option-Ressource mit ausgeliefert wird.

5.3.1 Komponententests

Methode	**Erwartete Antwort**
OPTIONS	Allow-*Header* mit bestimmten Methoden
HEAD	Statuscode 200 und den Content-Length-*Header*
GET	Die Repräsentation der Ressource und ein Etag
Conditinal GET	Not Modified wenn der Etag gleich ist
DELETE	200 Statuscode
Wiederholtes DELETE	404 "Not Found" Statuscode

Tabelle 5.9.: Ressource HTTP-Methoden Testschema

Wie bereits zuvor beschrieben sollen die Komponententests jede einzelne Ressource und deren unterstützten HTTP-Methoden testen. Es wird dabei nur der Statuscode der Antwort bzw. die Funktionsfähigkeit überprüft. D.h. es wird für jede einzelne Methode eine Anfrage an den Webservice gesendet und die Antwort überprüft. Der Webservice muss sich dabei auf dem MoCCha-Projekt Server befinden, sodass nicht lokal getestet wird. Für die GET-Methoden wird zudem überprüft ob das conditinal GET-Verhalten funktioniert. Die Methoden, die Parameter erwarten, z.B. POST oder PUT, werden mit falschen und korrekten Datenaustauschformaten (MIME media types), sowie Parametern angefragt. Die Antworten dieser Anfragen werden auf die Statuscodes überprüft. Z.B. das ein 415 HTTP-Statuscode gesendet werden sollte, wenn per POST-Anfrage an eine Ressource XML als Repräsentation gesendet wird. Dieser Statuscode wird erwartet, da nur JSON als Repräsentation unterstützt wird. Für jede erstellt Ressource sollten die Tests nach dem Schema aus der Tabelle 5.9 durchgeführt werden. Diese Tabelle beinhaltet alle möglichen unterstützten HTTP-Methoden die keine Parameter benötigen außer die URI. D.h. keinen HTTP *entity body* innerhalb der Anfrage. Für die Methoden PUT und POST sollten die Tests nach dem Schema aus Tabelle 5.10 erstellt werden. Dies ermöglicht das Aufschlüsseln von Fehlertests. Innerhalb der Spalte Beschreibung ist angegeben ob es sich um einen Fehlertest handelt und welche Parameter übergeben werden. Die Tests, die die geforderten Ergebnisse liefern sollen, werden durch einen Jersey Testclient durchgeführt. Dieser Testclient soll für jede Ressource eine eigene Test-Klasse besitzen, mit der er deren Methoden abfragen bzw. testen kann.

Beschreibung	**Erwartete Antwort**
Fehlertest: Falscher MIME media type	Statuscode 415
Test: Ressource erstellen	Statuscode 201 und Location-*Header*
Fehlertest: Falsche Repräsentation	Statuscode 400
Test: Wiederholter POST	Statuscode 201 und Location-*Header*
Test: Wiederholter PUT	Statuscode 200
Fehlertest: Falsche URI	Statuscode 404 oder 405

Tabelle 5.10.: Ressource POST/PUT Testschema

5.3.2 Integrationstests

Anhand der Integrationstests soll ermittelt werden ob die erstellten Ressourcen auch miteinander funktionieren bzw. ob sie auch auf die anderen Ressourcen referenzieren, so wie es nach der HATEOAS-Bedingung gefordert ist. D.h. es werden grundlegend nur die "Eltern" Ressourcen getestet und die gesendeten Repräsentationen überprüft, ob diese Informationen zum *resource state* bzw. zu anderen Ressourcen enthalten. Das Testen der "Eltern" Ressourcen soll zeigen, dass wenn mithilfe einer POST-Anfrage eine Ressource an der "Eltern" Ressource erstellt wurde, diese auch existiert und auch innerhalb der gesendeten Repräsentation der "Eltern" Ressource vorhanden ist. Ein Test sollte folgendermaßen ablaufen: Als erstes wird die POST-Anfrage mit der Repräsentation an die "Eltern" Ressource gesendet, z.b. an die Optionen-Ressource. Die Antwort dieser Anfrage beinhaltet den Location-*Header*, der auf die erstellte Ressource verweist. Diese URI sollte auf jeden Fall gültig sein und die Ressource sollte existieren. Nach der Prüfung der Existenz der Ressource sollte eine GET-Anfrage an die "Eltern" Ressource gesendet werden. Die Repräsentation die als Antwort gesendet wird sollte dann die erstellte Ressource, in diesem Beispiel die Option-Ressource, innerhalb einer Liste von Ressourcen beinhalten. Wurden alle Bedingungen eingehalten so ist der Test für diese "Eltern" Ressource bestanden. Diese Tests können in den erstellten Jersey Testclient integriert werden. Somit kann überprüft werden ob die späteren Clients, durch die gegebenen Informationen innerhalb der Repräsentation, selbst Ressourcen erstellen können und diese danach auch existent sind.

6 Implementierung

Das folgende Kapitel stellt das Vorgehen beim Re-Design des bestehenden Webservices dar und geht auf die entstandenen Probleme und Lösungen ein. Es wird zudem beschrieben in wie weit der Entwurf befolgt wurde oder wo, da es so nicht umsetzbar gewesen wäre, davon abgewichen werden musste. Als erstes wurde ein externes Java-Programm erstellt, welches die alten Parser beinhaltet. Es wurde mit dem Parser begonnen, da dieser die Daten liefert, die dem Client zur Verfügung gestellt werden. Die angegebenen URIs für die Ressourcen innerhalb dieses Kapitels beinhalten keinen Bezug auf die Domain des Servers. Des Weiteren beinhalten alle erstellten URIs als ersten Pfad "v2", welcher zur Versionierung des Webservices benutzt wird. Dieses Vorgehen wird auch von Richardson und Ruby für die Versionierung eines Webservices empfohlen. Dadurch ist es möglich eine weitere neue Version eines Webservices, der die URIs mit "v3" beginnend erhält, zu erstellen und diesen parallel mit der vorherigen Version einzusetzen. Durch dieses Vorgehen ist es möglich bei einer Änderung der URIs, innerhalb der neuen Version, die bisherigen Clients nicht auszuschließen (vgl. Richardson und Ruby 2007, S. 235 f.).

6.1 Parser

Wie bereits zuvor in den Kapiteln 4 und 5 erwähnt, existieren in dem bestehenden Webservice, Parser. Diese Parser stellen dem Webservice bestimmte Informationen bereit. Informationen über Speisepläne von Mensen, Kursdaten für die TU oder UdK und Events von der TU oder von der Staatsoper. Damit die Beschaffung dieser Daten und Information von der Auslieferung der Daten voneinander gekoppelt wird, wurde ein extra Java-Programm erstellt, welches diese Parser beinhaltet. Simultan zu dem externen Parser wurden die dazu gehörigen Ressourcen im Webservice erstellt. Sodass die erstellten Dateien bereits ausgelesen, weiterverarbeitet und die Ressource-Repräsentationen ausgeliefert werden konnten. D.h. für die Events, Kurse und die Mensen wurden die in Abschnitt 5.2.2 entworfenen Ressourcen erstellt. Die Daten die vom Parser geliefert werden, werden serialisiert in eine Datei gespeichert. Das bedeutet es wird z.B. für die Events eine "HashMap" erzeugt, die alle Events als *values* beinhaltet. Die *keys* der "HashMap" bestehen aus der ID und dem Titel des Events.

Diese Map wird serialisiert in eine Datei gespeichert. Durch die Serialisierung soll es dem Webservice möglich sein aus der Datei die HashMap mit den Informationen zu lesen und weiter zu verarbeiten. Die Daten und Informationen, die innerhalb der Datei gespeichert sind, können beliebig oft aktualisiert werden. Die Aktualisierung kann anhand der Ausführung des externen Java-Programms bewerkstelligt werden. Damit die Erweiterbarkeit gewährleistet ist, wurde innerhalb des Java-Programms und des Webservices ein *property file* mit dazugehörigem Parser, der die Eigenschaften ausliest, eingeführt. Solch ein *property file* beinhaltet für die Ausführung von Funktionalitäten, Informationen als *key-value* Paare. Siehe Beispiel 6.1. Innerhalb des *property file* kann angegeben werden welche URIs geparst werden sollen, sowie in welchem Ordner und welcher Datei die Informationen gespeichert werden sollen. Das *property file* innerhalb des Webservices beinhaltet den Ort und den Namen der Datei, die ausgelesen werden soll. Zudem kann das *property file* eine Liste von Eventkategorien beinhalten. So können die jetzigen Eventkateogrien, Staatsoper- und TU-Events, später z.B. für Events der UdK erweitert werden. Die angegebenen Kategorien stellen die Eventkategorien dar, die als Repräsentation zum Client gesendet werden. Für jede Kategorie kann eine Datei, die eine HashMap mit Events beinhaltet, angegeben werden. Die daraus resultierende Eventliste wird wiederum als Repräsentation an den Client ausgeliefert.

```
#EVENTS
events.dir=test
events.staatsoper.url=http://www.staatsoper-berlin.org/de_DE/calendar-next
events.staatsoper.file=staatsoper-events
events.tub.url=http://www.pressestelle.tu-berlin.de/menue/veranstaltungen/kalender/
events.tub.calendar.day.query=?view=day&showd=
events.tub.calendar.detail.query=?view=single&uid=
events.tub.file=tub-events
#COURSES
courses.dir=test
courses.tub.url=http://lsf2.tubit.tu-berlin.de/qisserver/servlet/de.his.servlet.RequestDispatcherServlet?state=
    wtree&search=1&category=veranstaltung.browse&menuid=lectureindex&menu_open=y
courses.tub.file=tub-courses
courses.udk.url=https://www.vdl.udk-berlin.de/qisserver/rds?state=wtree&search=1&trex=step&root120131=10125&P.vx=
    kurz
courses.udk.file=udk-courses
#CANTEEN
canteen.dir=test
canteen.file=canteen
canteen.url=http://www.studentenwerk-berlin.de/mensen/mensen_cafeterien/index.html
```

Beispiel 6.1: Mocchaparser property file

6.1.1 Events

Wie bereits zuvor beschrieben gibt es zwei Parser, die die sogenannten Events parsen. Zum einen einen Parser der die Staatsoper-Events parst und einen der die TU-Events parst. Diese verschiedenen Arten von Events werden als Kategorien bezeichnet. Für jede Kategorie kann eine eigene Eventliste bzw. auch eine genaue Eventbeschreibung vom Webservice geliefert werden, siehe Abschnitt 5.2.6.

6.1.2 Kurse

Genauso wie für die Events wurden die Kurskategorien erweiterbar gestaltet. D.h. es ist möglich innerhalb des bereits beschriebenen *property file*, siehe Abschnitt 6.1, mehrere Kategorien für die Kurse anzugeben. Diese Kategorien stehen für die geparsten Universitäten. Somit können später auch Kurspläne anderer Universitäten ausgeliefert werden. Zur Zeit werden nur die Kurspläne der TU Berlin und der UdK angeboten. Wird eine weitere Kategorie bzw. Universität in dem *property file* angegeben, muss ein Parser für diese Universität innerhalb des externen Programmes existieren. Mithilfe dieses Parsers wird dann die neue Universitätsdatei erstellt. Diese beinhaltet alle Kursdaten. Die erstellte Datei beinhaltet wie bei den Events eine serialisierte HashMap. Die HashMap beinhaltet als *key* die ID sowie den Titel des Kurses und als *value* alle Daten des Kurses. Sobald der Parser in dem externen Java Programm integriert ist und die geparsten Daten als Datei vorliegen, kann innerhalb des Webservice *property file* die Universität und der Dateipfad angegeben werden. Der Webservice liest dann die gespeicherten Daten aus und liefert sie bei Anfrage an den Client.

6.1.3 Mensen

Ähnlich wie bei den Kursen und Events werden die Daten der Mensen vom externen Java Program geparst, serialisiert und in eine Datei gespeichert. Die Daten der Mensen stammen allein vom Studentenwerk. Da keine weiteren Anbieter im Ausblick sind, die in den Webservice integriert werden sollen, wurde der Schritt der erweiterbaren Kategorien wie in Abschnitt 6.1.1 und 6.1.2 weggelassen. Jedoch ist es möglich innerhalb des *property file* anzugeben, in welche Datei die Daten gespeichert werden sollen und welche Webseite geparst werden soll. Die Repräsentationen der Mensen- und Mensa-Ressource wurde nach dem Entwurf umgesetzt. Siehe dazu Beispiel 5.23 und 5.22. Die Mensen-Ressource liefert eine Liste von Mensen. Jedes Element der Liste beinhaltet die ID, den Namen sowie die URI zu der jeweiligen Mensa-Ressource. Zudem enthält die Repräsentation die URIs zu den umliegenden Ressourcen (Kurskategorien, Eventkategorien) und die Information wie man eine Benutzer-Ressource erstellt. Die Mensa-Repräsentation, die auf eine GET-Anfrage als Antwort gesendet wird, enthält unter anderem die Informationen für eine Mensa wie z.b. den Namen und die Speisekarte. Aber auch Informationen wie man für diese Mensa eine Umfrage-Ressource erstellt. D.h eine Umfrage für diese Mensa. Die Repräsentation beinhaltet zudem eine URI auf die Mensen-Ressource.

6.2 Session-per-request

Wie bereits in dem Abschnitt 2.8.3 erwähnt wurde Hibernate für die Verbindung zur Datenbank verwendet. Für diese Verbindung wurde ein Pattern implementiert, das sich *session-per-request* nennt, siehe Abschnitt 2.8.3. Im Beispiel 6.2 ist eine statische Methode der Klasse DAOTransaction abgebildet. Diese Methode wird mit einem DAOAction Objekt aufgerufen. Dieses Objekt beinhaltet die Logik für die session bzw. transaction.

```
1  public static Response makeTransaction(DAOAction action) {
2    Transaction t = null;
3    Response r = null;
4    try {
5      Session s = HibernateUtil.getCurrentSession();
6      t = s.beginTransaction();
7      if (action != null)
8        r = action.action(s);
9      t.commit();
10
11   } catch (HibernateException he) {
12     if (t != null) {
13       t.rollback();
14     }
15     LOG.log(Level.SEVERE, "HibernateException", he);
16     r = Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
17   } finally {
18     return r;
19   }
20 }
```

Beispiel 6.2: Session-per-request

Somit ist das Einkapseln von Hibernate session Operationen möglich, wie z.B. das Auslesen und Löschen von Daten usw.. Mithilfe der Methode in Beispiel 6.2 ist es nicht erforderlich bei jeder Anfrage an einer Ressource die session und transaction zu erstellen und dies mit einem try-catch block zu umgeben. Somit wird doppelter Code erspart und durch den einmaligen Aufruf bei einer Anfrage an einer Ressource ist somit auch das *session-per-request* Pattern umgesetzt, da nur eine session und transaction verwendet wird. Alle Operationen die für die jeweilige Anfrage ausgeführt werden müssen, benutzen die selbe session und sind innerhalb der gestarteten transaction. Wie bereits erwähnt, beinhaltet das DAOAction Objekt die Logik. Diese Logik befindet sich innerhalb einer Methode namens action. Die Methode action wird von der makeTransaction Methode aufgerufen. Nach dem Aufruf wird die transaction commited und somit auch die derzeitige session geschlossen. Bei einem Fehlerfall wird die transaction zurückgerollt und die *Exception* geloggt. Des Weiteren wird dem Client eine Antwort mit dem Statuscode 500 (”Internal Server Error”) gesendet. In dem Beispiel 6.3 ist die GET-Methode für die Kontakte-Ressource dargestellt. Dies dient zur Erklärung und Darstellung der Benutzung der makeTransaction Methode. Zu Beginn der GET-Methode wird die makeTransaction Methode durch ein anonymes Klassenobjekt vom Typ DAOAction aufgerufen. Dieses Objekt implementiert die action Methode. Wie bereits zuvor erwähnt, werden alle nötigen Operationen innerhalb der action Methode umgesetzt und somit innerhalb einer transaction.

Das Response-Objekt das von der action Methode zurückgegegeben wird, wird von der makeTransaction weitergegeben, siehe Beispiel 6.2.

```
1  @GET
2  @Produces(MediaType.APPLICATION_JSON)
3  public Response getContacts(@HeaderParam("If-None-Match") final String ifNonMatch) {
4    return DAOTransaction.makeTransaction(new DAOTransaction.DAOAction() {
5
6      @Override
7      public Response action(org.hibernate.Session session) {
8        User u = MoCChaEntitiesProvider.readUserFromDB(uuid);
9        if (u == null)
10         return Response.status(Response.Status.NOT_FOUND).build();
11
12       ContactsRepresentation rep = createContactsRepresentation(uuid);
13       EntityTag tag = rep.getEntityTag();
14       if (ifNonMatch != null && ifNonMatch.equalsIgnoreCase(tag.getValue()))
15         return Response.notModified(tag).build();
16       else
17         return Response.ok(rep).tag(tag).build();
18     }
19   });
20 }
```

Beispiel 6.3: Kontakte GET-Methode

6.3 Webservice

In diesem Abschnitt werden alle anderen erstellten Ressourcen die noch nicht in dem vorherigen Abschnitt 6.1 beschrieben wurden, beschrieben. Einige der erstellten Ressourcen mussten vom Entwurf abgeändert werden. Diese Abänderungen sind in den jeweiligen Abschnitten genauer beschrieben.

6.3.1 Umfrage- & Optionen-Ressource

Die Umfrage-Ressource repräsentiert die Umfrage und wird mithilfe einer POST-Anfrage an die Umfragen-Ressource erstellt. Die URI ”/v2/{uuid}/polls/{pollID}” identifiziert die Umfrage-Ressource. Das Template ”{pollID}” beinhaltet die ID der Umfrage, die vom Server vergeben wird. Die Umfrage-Ressource unterstützt die HTTP-Methoden OPTIONS, HEAD, GET, PUT, DELETE. Durch eine PUT-Anfrage kann eine Umfrage komplett mit anderen Optionen sowie Teilnehmern, ersetzt werden. Das Löschen der Ressource erfolgt, wie bei allen Ressourcen, mit der DELETE-Methode. Diese Eigenschaften wurden im Kapitel 5 entworfen und auch so übernommen bzw. implementiert. Jedoch wurden Veränderungen an der Repräsentation, die an den Client ausgeliefert wird, vorgenommen. Wie in dem Beispiel 5.12 zu erkennen ist, werden in dieser Repräsentation die Optionen der Umfrage sowie die Bewertungen der Teilnehmer und die Informationen wie man eine Option-Ressource erstellt, ausgeliefert.

Da dies aber nicht der geeignete Ort für diese Informationen gewesen ist, wurde dies in die Optionen-Repräsentation verschoben. Die Umfrage-Repräsentation besitzt nun nur einen Verweis auf die Optionen-Ressource. Siehe Beispiel 6.4 und 6.5. Falls der Client Informationen zu den Optionen der Umfrage erhalten möchte, muss er diese Ressource anfragen. Die Umfrage-Repräsentation enthält somit Verweise auf die Optionen-, Umfragen- und Benutzer-Ressourcen. Des Weiteren die Informationen wie der Umfragen *resource state* verändert werden kann, wie man eine Teilnehmer-Ressource erstellt und eine Liste von Teilnehmer-Ressourcen. Jedes Element der Liste beinhaltet den Namen des Teilnehmers sowie die URI zur dazugehörigen Teilnehmer-Ressource. Da keine einmalig vordefinierte Ressource für die Teilnehmer existiert und auch nicht notwendig ist, wurde diese Information dagegen in der Umfrage-Repräsentation beibehalten.

```
{
  "options":"/{uudi}/polls/{id}/options",
  "member":[{"name":"NAME","uri":"HTTP-URI"},...],
  "addMember":{"uri":"/{uuid}/polls/{id}/{TELE-HASH}","method":"put"},
  "delete":{"uri":"/{uuid}/polls/{id}", "method":"delete"},
  "update":{"uri":"/{uuid}/polls/{id}",
    "method":"put",
    "type":"application/json",
    "parameter":{
      "member":[{"contact":"TELE-HASH"},...],
      "time":"UNIX-Timestamp",
      "options":[{"option":"CANTEEN"},...]
    }
  },
  "polls":"/{uuid}/polls",
  "self":"/{uuid}"
}
```

Beispiel 6.4: Veränderte Umfrage-Repräsentation

Aus dieser ”Verschiebung” von Informationen resultierte auch die Änderung der Optionen-Repräsentation. Diese Ressource ist unter der URI ”/v2/{uuid}/polls/{pollID}/options” erreichbar. Sie unterstützt die HTTP-Methoden OPTIONS, HEAD, GET und POST. Anhand der POST-Methode kann eine neue Option-Ressource erstellt bzw. eine Option der Umfrage hinzugefügt werden. Die Antwort auf eine POST-Anfrage beinhaltet, wenn die Erstellung erfolgreich war, einen HTTP-Statuscode 201 sowie den Location-*Header*, der die URI zur erstellten Ressource beinhaltet. Die Informationen wie man eine Option erstellt sind in der Repräsentation der Optionen-Ressource enthalten. Diese werden als Antwort auf eine GET-Anfrage gesendet. Außerdem beinhaltet sie zudem die URIs zur Umfrage- und Benutzer-Ressource und eine Liste von existierenden Optionen für die Umfrage. Diese Form der Repräsentation wurde bereits mit dem Beispiel 5.14 entworfen. Die einzige Änderung die zustande kam war die Abänderung der Liste von Optionen. Nach der Änderung bzw. Implementierung beinhaltete jedes Element der Liste, außer der Referenz auf die zugehörige Option-Ressource und der Option an sich, die Bewertungen der Teilnehmer für diese Option. Die veränderte Repräsentation ist dem Beispiel 6.5 zu entnehmen.

```
{
  "options":[{
    "option":"VALUE",
    "uri":"HTTP-URI",
    "memberRating":[{"name":"NAME","rating":"RATING","uri":"/{uuid}/polls/{id}/{TELE-HASH}"},...],
    ...
    }],
  "create":{"uri":"/{uuid}/polls/{id}/options",
    "method":"post",
    "type":"application/json",
    "parameter":{
      "option":"OPTION",
      "rating":"(1-10)"
    }},
  "poll":"/{uuid}/polls/{id}",
  "self":"/{uuid}"
}
```

Beispiel 6.5: Veränderte Optionen-Repräsentation

6.3.2 Ressourcen

In dem folgenden Abschnitt werden alle erstellten Ressourcen die keine Abänderung nötig hatten beschrieben. Sie werden deshalb auch nur kurz erläutert da sie im Kapitel 5 bereits erklärt wurden. Die Pfadangaben in den folgenden Abschnitten der Ressourcen sind alle nur relativ. D.h. unabhängig von der Domain des Servers.

Benutzer

Die Benutzer-Ressource wurde ganz nach dem Entwurf, in Kapitel 5, implementiert und unterstützt die HTTP-Methoden OPTIONS, HEAD, GET, PUT und DELETE. Die Ressource ist über folgenden Pfad erreichbar: ”/v2/{uuid}”. Der erste Pfad ist wie bereits zuvor beschrieben der Versionspfad. Der zweite Pfad beschreibt ein URI Template, welches mit einer ID eines Gerätes ersetzt werden muss. Zur Erklärung von URI Template siehe Abschnitt 2.4.7. Die ID soll vom Gerät des Benutzers bestimmt werden und dieses eindeutig identifizieren. Eine GET-Anfrage auf die Benutzer-Ressource liefert eine Repräsentation der Ressource und Informationen wie man einen Benutzer löschen, aktualisieren und erstellen kann außerdem wie man eine Applikations-Ressource und dazu eine Applikationsversions-Ressource erstellt. Des Weiteren liefert die Repräsentation URIs zu anderen Ressourcen wie z.B. Umfragen-, Kontakte- oder zur Validierungs-Ressource. Siehe dazu auch Beispiel 5.5.

Applikation

Die Applikations-Ressource unterstützt die HTTP-Methoden OPTIONS, HEAD, GET und PUT. Diese Ressource ist ”abhängig” von der Benutzer-Ressource. Dies ist auch in der URI Hierarchie erkennbar.
Die Applikations-Ressource ist über die URI ”/v2/{uuid}/{app}” erreichbar. Das URI Template ”{app}” sollte mit einem Applikationsnamen ersetzt werden. So kann z.B. eine PUT-Anfrage, an die URI ”/v2/{uuid}/MoCCha” eine Applikations-Ressource mit dem Namen ”MoCCha” erstellen.

Die Liste der bereits existierenden Applikationen bzw. Applikations-Ressourcen werden innerhalb der Benutzer-Repräsentation ausgeliefert. Eine GET-Anfrage auf die Applikations-Ressource liefert als Repräsentation die Informationen, wie man eine Applikationsversions-Ressource erstellt, welche Applikationsversions-Ressourcen bereits existieren und die Referenz auf die Benutzer-Ressource. Siehe dazu auch Beispiel 5.9.

Applikationsversion

Eine Applikation kann mehrere Versionen besitzen. Dies wird mit den Ressourcen Applikation und der Applikationsversion dargestellt. Die URI zu einer Applikationsversions-Ressource lautet: "/v2/{uuid}/{app}/{v}". Die Pfade vor dem Template "{v}" wurden bereits beschrieben. Das Template sollte mit einer Versionsnummer der Applikation des Benutzers ersetzt werden. Ein Beispiel wäre eine PUT-Anfrage an die URI "/v2/{uuid}/MoCCha/1.0". Mit dieser Anfrage wird der Applikation "MoCCha" die Version 1.0 hinzugefügt bzw. die Applikationsversions-Ressource mit der URI "/v2/{uuid}/MoCCha/1.0" erstellt. Die Applikationsversions-Ressource besitzt zudem versionsabhängige Applikationsdaten, die bei einer GET-Anfrage mit ausgeliefert werden. Mit den Applikationsdaten werden auch Informationen wie man diese Daten aktualisiert und die URIs zu den Ressourcen Applikation und Benutzer geliefert. Die ausgelieferte Repräsentation kann dem Beispiel 5.10 entnommen werden.

Kontakte

Die Liste der Kontakte eines Benutzers werden innerhalb der Kontakte-Repräsentation zurückgeliefert. Diese Ressource unterstützt die HTTP-Methoden OPTIONS, HEAD und GET und ist über die URI "/v2/uuid/contacts" erreichbar. Bei dieser Ressource handelt es sich um eine einmalig vordefinierte Ressource. Das Erstellen von Kontakt-Ressourcen wird mit der Methode PUT umgesetzt. Die Informationen wie man solch eine Ressource erstellt wird genauso wie die Liste der Kontakte und die URI, welche auf die Benutzer-Ressource verweist, innerhalb der Repräsentation geliefert. Jedes Element der Kontaktliste verweist auf die eigene Kontakt-Ressource. Siehe dazu auch Beispiel 5.6.

Kontakt

Jeder Kontakt eines Benutzers wird durch eine Kontakt-Ressource repräsentiert. Die Kontakt-Ressource unterstützt die HTTP-Methoden OPTIONS, HEAD, GET, PUT und DELETE. Wie bereits zuvor beschrieben, wird eine Kontakt-Ressource via PUT an die gewünschte Ressource erstellt. Eine Kontakt-Ressource besitzt folgende URI "/v2/uuid/contacts/{teleHash}". Das Template "{teleHash}" muss mit einem SHA 256 Telefonnummer *hash* ersetzt werden. Stimmt der angegebene *hash* zu keiner validen Benutzer Telefonhashnummer, so wird die Ressource nicht erstellt. Die Repräsentation der Kontakt-Ressource beinhaltet den Namen des Kontakts, die Informationen wie man eine Kontakt-Ressource erstellt sowie wie man diese löscht. Des Weiteren die URIs zu der Benutzer- und Kontakte-Ressource. Siehe dazu Beispiel 5.7.

Umfragen

Die Umfragen-Ressource stellt die Umfragen eines Benutzers, an denen er teilnimmt, dar. Die Ressource unterstützt die HTTP-Methoden OPTIONS, HEAD, GET und POST. Mithilfe der POST-Methode kann eine neue Umfrage bzw. eine Umfrage-Ressource erstellt werden. Die Antwort auf eine POST-Anfrage beinhaltet, wenn die Erstellung erfolgreich war, einen HTTP-Statuscode 201 und einen Location-*Header*. Dieser *Header* beinhaltet die URI zur erstellten Ressource. Die Informationen zur Erstellung einer Ressource wird mit der Umfragen-Repräsentation auf eine GET-Anfrage als Antwort ausgeliefert. Die Repräsentation beinhaltet wie zuvor erwähnt eine Liste von existierenden Umfrage-Ressourcen mit Verweisen auf diese sowie die URI der Benutzer-Ressource. Siehe dazu Beispiel 5.11. Die Umfragen-Ressource wird durch die URI "/v2/{uuid}/polls" identifiziert. Wie bei der Kontakte-Ressource handelt es sich bei dieser um eine einmalig vordefinierte Ressource. Da die Umfrage-Ressource kleine Abänderungen in ihrer Repräsentation erhalten hat, wurde diese bereits in dem Abschnitt 6.3.1 erklärt.

Teilnehmer

Da zu einer Umfrage mehrere Teilnehmer gehören, werden diese durch die Teilnehmer-Ressource dargestellt. Jeder Teilnehmer einer Umfrage besitzt seine eigene Ressource. Die Teilnehmer sind im Allgemeinen über dieses URI Template identifizierbar: "/v2/{uuid}/polls/{pollID}/{teleHash}". Das Template "{teleHash}" sollte genauso wie bei der Kontakt-Ressource mit einem SHA 256 Telefonnummer *hash* ersetzt werden. Die Repräsentation der Ressource ähnelt zudem stark der Kontakt-Ressource. Denn auch hier ist der Name des Teilnehmers, die Informationen wie man einen Teilnehmer erstellt und löscht, sowie die URI des Benutzers, in der Repräsentation enthalten. Die Repräsentation verweist des Weiteren noch auf die Umfrage-Ressource und besitzt die Informationen wie der Teilnehmer der Umfrage die existierenden Optionen bewertet hat. Es wird für jede abgegebene Bewertung mithilfe der URI, auf die dazugehörige Option-Ressource verwiesen. Siehe dazu Beispiel 5.13. Die HTTP-Methoden die von der Teilnehmer-Ressource unterstützt werden sind: OPTIONS, HEAD, GET, PUT und DELETE. Da eine Umfrage mehrere Teilnehmer besitzt ist die Liste der existierenden Teilnehmer-Ressourcen innerhalb der Umfrage-Repräsentation enthalten.

Option

Die Option für eine Umfrage wird mithilfe der Option-Ressource repräsentiert. Diese ist über die URI "/v2/{uuid}/polls/{pollID}/options/{optionID}" erreichbar. Das Template "{optionID}" beinhaltet die vom Server gegebene ID der Option. Die Option-Ressource unterstützt die HTTP-Methoden OPTIONS, HEAD, GET, PUT und DELETE. Mithilfe der PUT-Methode kann eine Option ersetzt bzw. überschrieben und mit DELETE gelöscht werden. Die Repräsentation der Ressource, die auf eine GET-Anfrage als Antwort gesendet wird, enthält Informationen wie man den *resource state* ändert (via PUT, DELETE), eine andere Option-Ressource erstellt, Referenzen bzw. URIs der Optionen-, Umfrage- und Benutzer-Ressourcen sowie Informationen zu der Option.

Die Informationen zur Optionen beinhaltet welche Mensa zur Umfrage steht, die Bewertung des Benutzers sowie die Bewertungen der anderen Teilnehmer. Haben die anderen Teilnehmer diese Option noch nicht bewertet, so werden sie dort nicht aufgelistet. Jede Bewertung beinhaltet den Namen des Teilnehmers, die Bewertung, sowie die URI zu dessen Teilnehmer-Ressource. Siehe dazu Beispiel 5.15.

6.3.3 Validierung

In dem Entwurf, siehe Kapitel 5, ist eine Validierung der Telefonnumer des Benutzers vorgesehen. Diese Validierung hat den Zweck das die angegebene Telefonnummer des Benutzers auch wirklich eine korrekte Nummer, vorallem die des Benutzers sein soll. Zur Validierung ist lauf Entwurf vorgesehen, siehe Abschnitt 5.2.4, dass der Benutzer seine Telefonnummer anhand einer POST-Anfrage an seine eigene Benutzer-Ressource sendet. Nach dieser POST-Anfrage wird eine Ressource mit dem Namen "Validierung" erstellt. Das Problem hierbei ist, dass diese Ressource im Grunde eine einmalig vordefinierte Ressource darstellt und keine Objekt-Ressource, siehe Abschnitt 2.4.7. Denn der Name der Ressource ist nicht abängig von der übergebenen Scoping-Information. Wäre die Ressource nach dem Entwurf umgesetzt worden, hätte man nur eine Ressource erstellen dürfen, die dann "Validierung" heißen würde. Nach einer weiteren POST-Anfrage würde es dann wahrscheinlich zu Problemen kommen. Würde zum Beispiel die bereits bestehende Ressource ersetzt werden, wäre dieses Verhalten innerhalb des HTTP-Standards kein definiertes Vorgehen für die POST-Methode. Siehe zur Erklärung von POST Abschnitt 2.4.6. Durch dieses nicht standardmässige Verhalten und durch die Tatsache das Objekt-Ressourcen nur erstellt werden können und keine einmalig vordefinierten Ressourcen, musste der Entwurf angepasst werden. Aus dieser einen Ressource wurden zwei Ressourcen erstellt. Eine einmalig vordefinierte Ressource namens "Validierung", die über die URI "/v2/{uuid}/validation" identifiziert wird und eine Objekt-Ressource namens "gültigeNummer" die durch die URI "/v2/{uuid}/validation/{teleHash}" identifiziert wird. Das Template: "{teleHash}" repräsentiert einen *hash* der vom Server generiert wird. Die Validierungs-Ressource unterstützt die HTTP-Methoden GET, HEAD und POST. Durch eine POST-Anfrage an die Validierungs-Ressource kann eine gültigeNummer-Ressource erstellt werden. Die POST-Methode erwartet ein *key-value* Paar in folgender Form: "tele:TELEFONNUMMER". Die Telefonnummer wird im Klartext erwartet, da somit später eine SMS an den Benutzer gesendet werden kann. Die SMS beinhaltet einen generierten Validierungsschlüssel. Die Umsetzung des Versendens einer SMS wurde, wie bereits im Kapitel 4 beschrieben, zurückgestellt. D.h. die erstellten Ressourcen dienen als Gerüst für die spätere Logik zum Versenden einer SMS. Diese SMS soll später zur Validierung der Telefonnummer des Benutzers dienen. Mithilfe des SHA 256 *hashs* der Telefonnummer wird die gültigeNummer-Ressource erstellt. Die Antwort auf eine POST-Anfrage beinhaltet den HTTP-Statuscode 201 und den Location-*Header* der auf die erstellte Ressource verweist. Wie bereits zuvor beschrieben, handelt es sich bei der gültigeNummer-Ressource um eine Objekt-Ressource.

Laut der Definition von Richardson und Ruby, siehe Abschnitt 2.4.7, kann eine Ressource vom Typ Objekt-Ressource, da sie wie der Name schon andeutet ein Objekt darstellt, unendlich oft erstellt werden. Dennoch hat jede Ressource ihren eigenen Bezeichner (die URI). Mit dieser Tatsache kann ein Benutzer mehrere Telefonnummern validieren, bzw. für die gegeben Nummern eine Ressource erstellen. Dies kann z.B. nützlich sein wenn man den Webservice so erweitern will, dass ein Benutzer als eigene Telefonnummer nur eine gültige Nummer angegeben kann. Die Liste der gültigen Nummern des Benutzers bzw. die Liste der gültigeNummer-Ressourcen, werden innerhalb der Validierungs-Repräsentation bereitgestellt, siehe Beispiel 6.6.

```
{
  "validNumbers":[{
  "teleHash":"TELE-HASH","uri":"URI"
  }],
  "create":{
    "method":"post",
    "uri":"/{uuid}/validation",
    "type":"application/x-www-form-urlencoded",
    "parameter":"tele:TELEPHONE-NUMBER"
  },
  "self":"/{uuid}"
}
```

Beispiel 6.6: Validation-Repräsentation

Die Liste der gültigeNummer-Ressourcen beinhaltet den Telefonnummer *hash* sowie die URI der Ressource. Des Weiteren beinhaltet die Validierungs-Repräsentation, die auf eine GET-Anfrage als Antwort gesendet wird, die Referenz zur Benutzer-Ressource und eine Beschreibung wie man eine gültigeNummer-Ressource erstellt. Wie bereits im Kapitel 4 festgelegt, werden die Repräsentationen nur im JSON Datenaustauschformat ausgeliefert. Da durch Erstellung der gültigeNummer-Ressource die Telefonnummer aber noch nicht validiert wurde, bietet die gültigeNummer-Ressource die PUT-Methode an. Anhand der PUT-Methode kann der *resource state* aktualisert/ersetzt werden. Um die Telefonnummer des Benutzers zu validieren muss der Validierungsschlüssel, der später per SMS versendet wird, an die erstellte gültigeNummer-Ressource, via PUT-Anfrage, gesendet werden. Der Validierungsschlüssel wird als *key-value* Paar innerhalb des *entity body* der Anfrage gesendet. Das *key-value* Paar sieht wie folgt aus: ”key:VALIDIERUNGSSCHLÜSSEL”, wobei VALIDIERUNGSSCHLÜSSEL mit dem empfangenen Schlüssel ersetzt werden muss. Stimmt der Validierungsschlüssel mit dem Schlüssel innerhalb der Datenbank überein, so wird die Telefonnummer bzw. der gespeicherte *hash* auf validiert gesetzt. Die Zeit der Validierung wird innerhalb der Datenbank gespeichert. Die PUT-Methode ist nicht die einzige HTTP-Methode die die gültigeNummer-Ressource unterstützt. Des Weiteren werden die Methoden GET und HEAD unterstützt. Das Beispiel 6.7 beschreibt die gültigeNummer-Repräsentation.

```
{
  "valid":"true/false",
  "validationDate":"DATE",
  "update":{
    "method":"put",
    "uri":"/{uuid}/validation/{teleHash}",
    "type":"application/x-www-form-urlencoded",
    "parameter":"key:VALIDATION-KEY"
  },
  "validNumbers":"/{uuid}/validation",
  "self":"/{uuid}",
}
```

Beispiel 6.7: gültigeNummer-Repräsentation

Die gültigeNummer-Repräsentation die bei der GET-Anfrage als Antwort gesendet wird, beinhaltet den Status der Validierung, das Validierungsdatum, eine Beschreibung wie man die gültigeNummer-Ressource aktualisiert und eine Referenz auf die Benutzer und Validierungs-Ressource. Wenn der Status der Validierung *false* ist, beinhaltet die Repräsentation auch kein Validierungsdatum, da diese Telefonnummer noch nicht validiert wurde. Die Beschreibung zum Aktualisieren des gültigeNummer *resource states* beschreibt, wie man die Telefonnummer mit den nötigen Informationen validiert. Wie bereits zuvor beschrieben kann durch den neuen Entwurf der Validierung, mehrere Telefonnummern zur Validierung gespeichert werden. Durch diese Veränderung musste das Datenbankschema angepasst werden. Es wurde eine neue Tabelle namens "ValidNumber" eingeführt. Diese Tabelle steht in einer N:1 Beziehung zur "User" Tabelle, siehe Abbildung 6.1. Denn ein Benutzer kann mehrere Telefonnummern validieren, aber eine Nummer kann nur von einem Benutzer validiert werden. Durch diese Beziehung ergibt sich der zusammengesetzte Schlüssel aus der Benutzer UUID und dem teleHash der "ValidNumber" Tabelle.

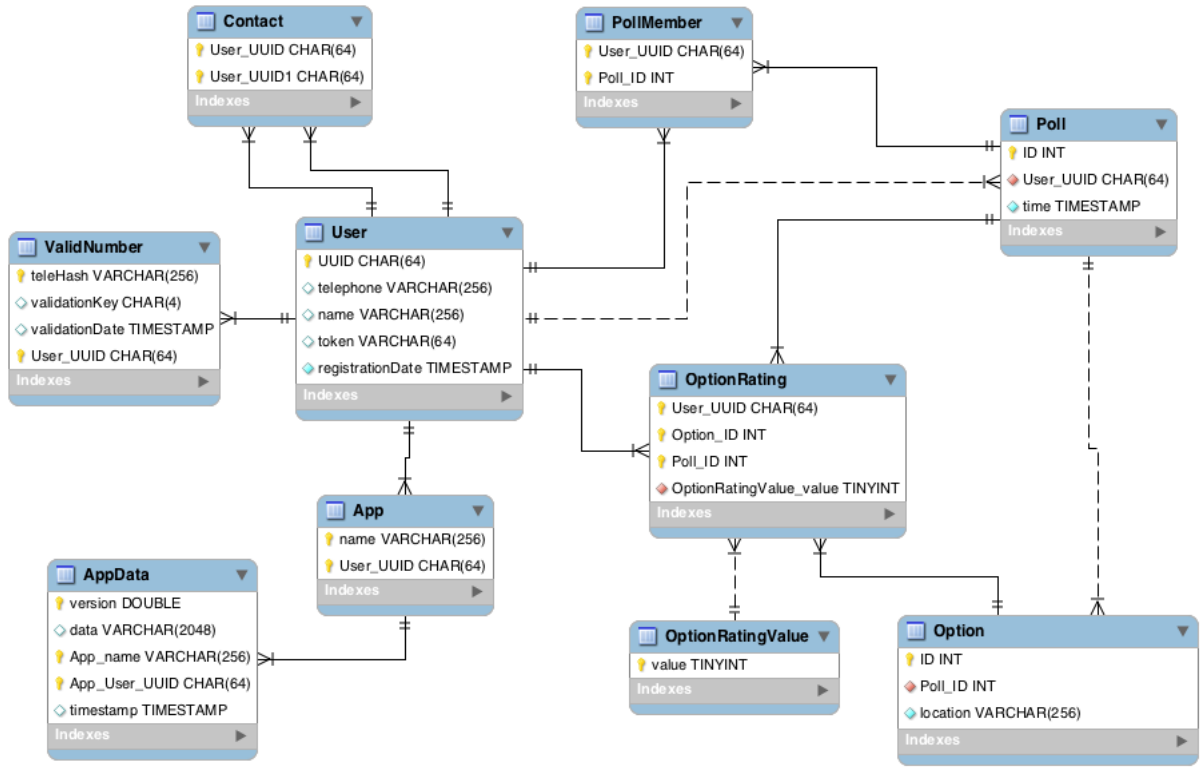

Abbildung 6.1.: Verändertes EER-Model

Die "ValidNumber" Tabelle speichert außerdem noch den Validierungsschlüssel, der für die Validierung benutzt wird und das Validierungsdatum. Nach den beschriebenen Änderungen konnte die Ressource, so wie sie in diesem Abschnitt 6.3.3 entworfen wurde, implementiert werden.

6.3.4 Conditional GET

Jede GET-Methode wurde als conditinal GET implementiert, siehe dazu Abschnitt 2.4.7. Der gesendete Etag beinhaltet einen MD5 *hash* wert. Die Ressourcen die die Daten des Parsers auslesen, generieren den Etag aus der Größe der Datei und dem Änderungsdatum. Die Etags der anderen Ressourcen werden anhand derer Repräsentationen generiert. In Beispiel 6.8 wird eine Antwort auf eine GET-Anfrage der Mensen-Ressource ohne If-None-Match-*Header* dargestellt. Hierbei wird aus Platzgründen nur der HTTP-*Header* der Antwort dargestellt.

```
1 Status Code: 200
2 Date: Wed, 17 Jul 2013 21:56:56 GMT
3 Server: Apache-Coyote/1.1
4 ETag: "463b8962264542f17b4c49f95db55941"
5 Content-Length: 4676
6 Content-Type: application/json
```

Beispiel 6.8: Mensen-Ressourcen Rückantwort

Der HTTP-*Header* beinhaltet unter anderem den Statuscode und den Etag-*Header*. Dieser kann vom Client wiederverwendet werden. Bei der nächsten GET-Anfrage ist es dem Client möglich den Wert aus dem Etag als If-None-Match-*Header* mit zu senden. Ist der Etag gleich bzw. hat sich die Repräsentation nicht verändert, so wird die in Beispiel 6.9 dargestellte Antwort an den Client gesendet. Die Antwort beinhaltet keinen HTTP *entity body* nur diesen HTTP-*Header*. Der Statuscode 304 bedeutet "Not Modified". D.h. das seit der letzten Anfrage die Ressource nicht verändert wurde. Wenn der Wert im If-None-Match-*Header* nicht mit dem Etag übereinstimmt, wird die Antwort aus Beispiel 6.8 gesendet. Dieses Verhalten erspart das Ausliefern einer Repräsentation, in diesem Beispiel in der Größe von 4676 Bytes (siehe Beispiel 6.8 Content-Length-*Header*).

```
1 Status Code: 304
2 Date: Wed, 17 Jul 2013 22:06:50 GMT
3 ETag: "463b8962264542f17b4c49f95db55941"
4 Server: Apache-Coyote/1.1
```

Beispiel 6.9: Mensen not modified

Tests
7

Die Tests die im Kapitel 5 entworfen wurden, wurden für jede Ressource umgesetzt. Um die Ressourcen geeignet zu testen, wurde ein externer Testclient mithilfe von Jersey erstellt. Dieser Testclient ermöglicht das automatisierte Abfragen aller Ressourcen. Nicht nur die Komponententests befinden sich in diesem Client, sondern auch diese sogenannten Integrationstests. Diese wurden für die Ressourcen Validierung, Umfragen und Optionen durchgeführt. Die meisten Fehltests waren nicht nötig, da die erwünschten Fehlerantworten automatisch von Jersey generiert wurden. So wird bei einem nicht akzeptierten Datenaustauschformat, das vom Client gesendet wird, automatisch die Antwort mit dem Statuscode 415 gesendet ("Unsupported Media Type"). Bei einer Anfrage mit dem Accept-*Header*, der ein Datenaustauschformat enthält welches wiederum nicht unterstützt wird, wird eine Antwort mit dem Statuscode 406 ("Not Acceptable") gesendet. Fragt der Client eine Ressource an die nicht existiert, wird von Jersey automatisch eine Antwort mit dem Statuscode 404 ("Not Found") generiert und zurückgesendet. Für den erstellten Webservice wurden keine Performancetests durchgeführt da nicht nur die Webservice-Architektur verändert wurde sondern auch die Datenbankverbindung bzw. das zu benutzende *framework*. Was sich auch erheblich auf die Performance auswirkt, siehe Abschnitt 3.3.

Hibernate

Anhand der erstellten und ausgeführten Tests konnte Hibernate bzw. deren Umsetzung im Webservice getestet werden. Durch diese Tests ist ein besonderer Fehler aufgetreten, der die TransactionException auslöste: "nested transactions not supported". Dies lag an der Implementierung der DAOTransaction wie sie im Kapitel 6 vorgestellt wurde, siehe Beispiel 6.2. Dort wird wie bereits in dem Kapitel beschrieben, eine Hibernate eigene transaction mit der derzeitigen session erstellt und alle Operationen innerhalb der transaction ausgeführt. Die Logik befindet sich in der action Methode des DAOAction Objekts. Wie in dem Beispiel 6.2 zu erkennen ist, werden HibernateExceptions also Fehler die von Hibernate ausgelöst werden, abgefangen sodass dann die transaction zurückgerollt werden kann. Durch die erstellten Tests ist aufgefallen, dass wenn eine andere *Exception* geschmissen wird, wie z.B. IllegalArgumentException, was bei einem Test der Fall war, die transaction nicht zurückgerollt wird. Das bedeutet das die transaction somit auch nicht beendet wird und die session immer noch eine offene transaction besitzt.

Da mehrere Anfragen innerhalb eines Tests durchgeführt wurden, z.B. Erstellen, Abfragen, Aktualisieren und Löschen eines Benutzers, wurde bei den darauffolgenden Anfragen die selbe session benutzt. Das hatte zur Folge das mithilfe der session eine neue transaction geöffnet werden sollte, obwohl bereits noch eine transaction bestand. Durch dieses Verhalten wurde die TransactionException ausgelöst. Nach Anpassung der makeTransaction Methode aus Beispiel 6.2 wurde eine neue Instanzmethode erstellt, die diesen Fehler nun nicht mehr auslöst, siehe Biepsiel 7.1. Es werden alle möglichen Errors und *Exceptions* gefangen, sodass es möglich ist die transaction zurückzurollen bzw. dem Client einen HTTP-Statuscode 500 ("Internal Server Error") als Antwort zu senden. Sollte dennoch eine transaction innerhalb der session offen sein, wird diese commited und eine neue session aus dem pool geholt. Da Jersey für jede Anfrage einen Thread erstellt, der für die angefragte Ressource ein neues Objekt erstellt, wurde die run Methode als Instanzmethode implementiert. So besitzt jedes Ressourcen Objekt sein eigenes DAOTransaction Objekt. Teilen sich zwei Threads jedoch das selbe Ressourcen Objekt und somit auch evtl. das DAOTransaction Objekt, müssen die Threads bei der Verarbeitung aufeinander warten, da die Methode synchronized ist. So können Seiteneffekte verhindert werden.

```
1  public synchronized Response run(DAOAction action) {
2    Transaction t = null;
3    Response r = null;
4    try {
5      Session s = HibernateUtil.getCurrentSession();
6      t = s.getTransaction();
7      if (t.isActive()) {
8        t.commit();
9        s = HibernateUtil.getCurrentSession();
10     }
11     t = s.beginTransaction();
12     if (action != null)
13       r = action.action(s);
14     t.commit();
15
16   } catch (Throwable tw) {
17     LOG.log(Level.SEVERE, "Exception", tw);
18     if (t != null) {
19       t.rollback();
20     }
21     r = Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
22   } finally {
23     return r;
24   }
25 }
```

Beispiel 7.1: Neue DAOTransaction Methode

8 Ergebnis

Innerhalb dieses Kapitels wird das Ergebnis dieser Arbeit vorgestellt und auf die Vorteile und die evtl. Nachteile des erstellten Webservice gegenüber den bestehenden Webservice hingewiesen.

8.1 Webservice

In dieser Arbeit wurde, wie bereits zuvor beschrieben, ein bestehender Webservice neu als *RESTful* Webservice erstellt. Der zu erstellende *RESTful* Webservice wurde nach der ROA, siehe Abschnitt 2.4, entwickelt. Da der bestehende Webservice bereits einige Ressourcen angeboten hatte und HTTP-Methoden benutzte, könnte man diesen in den Level 2 des "Richardson Maturity Model" eingliedern, siehe Abschnitt 2.3.5. Jedoch bot der bestehende Webservice nur einmalig vordefinierte Ressourcen an und benutzte die HTTP-Methoden an manchen Ressourcen entgegen deren Definitionen. Siehe z.B. Tabelle 3.6, die alle möglichen Funktionalitäten des Friends *Servlet* darstellt. Dort wird z.B. per POST ein Benutzer gelöscht oder einige Benutzereigenschaften aktualisiert. Dieses Verhalten entspricht erstens nicht dem vom HTTP definierten Verhalten, siehe 2.4.6 und zweitens auch nicht dem generellen Verhalten innerhalb des Webservices. Denn an anderen Ressourcen werden die eigentlichen HTTP-Methoden benutzt um z.B. eine Umfrage zu löschen, DELETE. Somit existiert kein *uniform interface*. Um diesen Webservice als *RESTful* Webservice neu zu erstellen, mussten die Ressourcen neu definiert werden sodass z.B. auch Objekt-Ressourcen benutzt werden, siehe Abschnitt 2.4.7. Des Weiteren musste die definierte Benutzung der HTTP-Methoden sichergestellt werden. Da ein Webservice erst *RESTful* ist wenn er auch das Level 3 des "Richardson Maturity Model" erreicht hat, bzw. die HATEOAS-Bedingung erfüllt, mussten die Repräsentationen der Ressource so gestalltet werden das sie auf andere Ressourcen und andere *resource states* verweisen, siehe Abschnitt 2.3.5 und 2.4.7. Innerhalb des Entwurfes des *RESTful* Webservices wurde das generische ROA-Entwurfsverfahren verwendet, siehe Kapitel 5. Das ermöglicht das Entwerfen der Ressourcen, deren Repräsentationen sowie der unterstützen HTTP-Methoden. Mit diesen Änderungen gegenüber dem bestehendem Webservice, konnte das *uniform interface* auch gewährleistet werden.

Durch die erfolgreiche Implementierung der HATEOAS-Bedingung, konnte einfach ein Testclient erstellt werden, der diesen Webservice und deren Ressourcen anfragt. Die Referenzen auf andere Ressourcen oder die Beschreibung wie man den *resource state* verändert, ermöglicht es späteren Entwicklern diesen Webservice zu benutzen bzw. einen Client einfacher gegen den Webservice zu entwickeln. Da er selbstbeschreibend ist, ist es nicht nötig eine Dokumentation, wie bei dem vorherigen Webservice, zu lesen, bevor der Webservice benutzt werden kann. Diese Eigenschaft macht es nicht nur leichter den Webservice zu benutzen sondern ermöglicht auch das Ändern von verschiedenen URIs, was beim alten Webservice nicht der Fall war. Da diese URIs mithilfe von Repräsentationen ausgeliefert werden, wird kein Client ausgeschlossen. Um den Webservice zu versionieren wurde als erster URI Pfad "v2" eingeführt. Dies soll somit die Versionsnummer des Webservice darstellen. Das ermöglicht das Entwickeln einer neuen Version des Webservices ohne das alte Clients wiederum ausgesperrt werden (vgl. Richardson und Ruby 2007, S. 235 f.). Da die Repräsentationen mehrere Referenzen auf andere Ressourcen besitzen oder Informationen wie man den *resource state* der Ressource ändert, wird um einiges mehr Bandbreite beansprucht. Dies wiegt sich aber durchaus mit deren Vorteilen auf (vgl. ebd., S. 223 ff.). Die Bandbreite kann durch die conditional GET-Methode zudem vermindert werden, siehe Abschnitt 2.4.6. Was bei mobilen Clients durchaus von Vorteil ist. Da somit nicht unnötig die Repräsentationen gesendet werden, obwohl der Client diese Informationen bereits besitzt. Das ist natürlich auch ein großer Vorteil gegenüber dem zuvor bestehendem Webservice, da dieser bei jeder GET-Anfrage die Repräsentationen mit auslieferte. Ein Vorteil von REST und somit auch des erstellten *RESTful* Webservice ist ist die Zustandslosigkeit und somit die Skalierbarkeit. Dadurch das der Service keinen *application state* auf dem Server speichert, ist es möglich den Service im Bedarfsfall auf weitere Maschinen mithilfe eines load balancers zu skalieren (vgl. ebd., S. 222). Der erstellte Webservice beinhaltet alte Funktionalitäten, die verbessert oder erweitert wurden. Das Parsen der Webseiten die die Daten für die Kantinen, Kurse oder Events liefern, wurde ausgelagert. Das hat den Vorteil das dieser Parser unabhängig vom erstellten Webservice ist und ausgeführt werden kann. Im bestehenden Webservice war dies Teil des Webservice und wurde durch Java cronjobs zu bestimmten Zeitpunkten innerhalb des Webservices ausgeführt. Dadurch war es nicht möglich die Daten anders zu aktualisieren bzw. den Parser auszuführen und die Zeitpunkte der cronjobs zu ändern, ohne das der Webservice neu deployed werden musste. Innerhalb des externen Parsers können nun auch neue Parser für z.B. Events oder Kurse hinzugefügt werden. Diese können innerhalb der externen *property file* des Webservice angegeben werden, sodass dieser die Daten der neuen Parser mit benutzt. Nicht nur die Erweiterbarkeit der Parser ist gewährleistet, sondern auch der Ressourcen des Webservice. Durch die Benutzung von Jersey für die Implementierung des *RESTful* Webservices, ist es möglich einfach weitere Ressourcen hinzuzufügen. Siehe dazu den Vergleich zwischen *Servlet* und Jersey Implementierung im Abschnitt 3.2.2. Durch einfaches Benutzen der JAX-RS *Annotation* @Path kann eine weitere Ressource dem Webservice hinzugefügt werden. Um für andere Entwickler die Erweiterbarkeit zu gewährleisten, sowie die Wartbarkeit von Funktionalitäten oder Ressourcen, wurde die Implementierung des Webservice dokumentiert und in Folge dessen ein JavaDoc generiert.

Dies ist ein weiterer Vorteil gegenüber dem alten Webservice, da dort die Dokumentation gänzlich fehlte. Anhand der Implementation, die in dieser Arbeit getätigt wurde, kann behauptet werden das durch die Erfüllung der HATEOAS-Bedingung eines *RESTful* Webservice, er an Komplexität und Aufwand zu nimmt. Dieser getätigte Mehraufwand ermöglicht es aber, dass der Service nach Außen hin leichtgewichtig erscheint und leicht zu benutzen ist. Dies ist vorallem bei der Clienterstellung bemerkbar und von Vorteil. Der Implementationsaufwand kann auch durch die Wahl der Implementationsweise weiter steigen, d.h. ob dieser Webservice z.B. mit *frameworks* oder in Java z.B., nur mit *Servlets*, umgesetzt wurde. Um diesen Mehraufwand darzustellen wurden zwei *RESTful* Beispiele erstellt und deren Aufwand verglichen, siehe Abschnitt 3.2. Anhand dieses Vergleiches kann behauptet werden das die Verwendung von *frameworks* ratsam ist und im Falle der REST Implementierung der Aufwand reduziert werden kann. Zudem bieten *frameworks* einige Funktionalitäten die schwer bzw. aufwendig nachzuimplementieren wären. Diese Erkenntnis war eines der Ziele dieser Arbeit. Ein weiteres Ziel war die Implementierung eines *RESTful* Webservices nach dem Schema eines bestehenden Webservices und das Aufzeigen der Vorteile des *RESTful* Webservices. Dieses Ziel wurde durch Kapitel 5 und 6 und den gezeigten Vorteilen innerhalb dieses Abschnittes bewerkstelligt.

8.2 Ausblick

Für den erstellten *RESTful* Webservice gibt es noch einige Features, die in Ausblick stehen. Zum einen ist ein weiteres Feature, was der bestehende Webservice bereits besitzt, das Versenden einer SMS bei der Validierung der Telefonnummer eines Benutzers. Dies wurde außen vor gelassen da das Augenmerk auf die Umstellung auf einen *RESTful* Webservice lag. Des Weiteren wäre zum Beispiel denkbar, dass wenn eine Umfrage erstellt und Teilnehmer von einem Benutzer hinzugefügt wurden, diese per Push Notification auf ihrem Smartphone benachrichtigt werden können. Außerdem wäre es möglich das das Benutzer-Model erweitert wird, sodass dieser sich authentifizieren kann. Anhand der Authentifizierung wäre es möglich zu gewährleisten das nur bestimmte Benutzer Zugriff auf Ressourcen haben, bzw. deren *resource state* ändern können (vgl. Richardson und Ruby 2007, S. 238). Das hätte zur Folge das die Repräsentationen auch abhängig von der Authentifizierung gestaltet werden. Sodass wenn ein Benutzer durch Authentifizierung berechtigt ist den *resource state* zu ändern, die Repräsentation der Ressource die Information, wie man diesen ändert enthält. Ist der Benutzer jedoch nicht berechtigt so werden nur die Informationen zum jetzigen *resource state* und die Referenzen auf andere Ressourcen ausgeliefert.

Abkürzungsverzeichnis
9

bzw. beziehungsweise

d.h. das heißt

DB Datenbank

DBMS Datenbankmanagementsystem

DBS Datenbanksystem

ebd. ebenda

EER Enhanced Entity-Relationship

engl. englisch

HATEOAS Hypermedia as the engine of application state

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

ID Identifikationsnummer

JSON Javascript Object Notation

MIME Multipurpose Internet Mail Extensions

MoCCha Mobiler Campus Charlottenburg

ORM Object Relation Mapping

POJO Plain Old Java Object

REST Representational State Transfer

ROA Resource-Oriented Architecture

RPC Remote Procedure Call

SOA Service-Ortiented Architecture

SOAP Simple Object Access Protocol

T-Labs Telekom Innovation Laboratories

TU Technische Universität

u.a. und andere

UdK Universität der Künste

URI Uniform Resource Identifier

URL Uniform Resource Locator

usw. und so weiter

vgl. vergleiche

XHTML Extensible HyperText Markup Language

XML Extensible Markup Language

z.B. zum Beispiel

Glossar
10

Annotation Englisch für Beschriftung oder Vermerk, wird in dieser Arbeit im Zusammenhang mit der Programmiersprache Java verwendet. Die Annotations sind ein Bestandteil der Programmiersprache Java und ermöglichen Metadaten zur Kompiler- und Laufzeit mit Java Programmelementen zu verbinden (vgl. R. Liguori, P. Liguori und Schulten 2008, S. 62). Innerhalb dieser Arbeit wurden die Annotations aus dem Package javax.ws.rs verwendet.

Deskriptor Englisch für Beschreiber, wird in dieser Arbeit im Zusammenhang vom Deployment und web.xml benutzt. Der Deskriptor oder Deployment-Deskriptor bezeichnet etwas, was den Entwicklungsprozess bzw. die Aufstellung eines Webservices beschreibt.

Envelope Englisch für Umschlag oder Hülle, wird in dieser Arbeit im Zusammenhang von HTTP benutzt. Der HTTP-Envelope oder Umschlag ist das was von einem Client zum Server gesendet wird und zurück. Innerhalb dieses Umschlages befindet sich ein entity body oder auch Repräsentation die dabei übertragen wird (vgl. Richardson und Ruby 2007, S. 5).

Exception Englisch für Ausnahme oder Ausnahmefall, wird in dieser Arbeit im Zusammenhang mit der Programmiersprache Java verwendet. Java Exceptions beschreiben einen Ausnahme- oder Fehlerfall innerhalb eines Java Programmes.

Header Englisch für Kopfzeile, wird in dieser Arbeit im Zusammenhang von HTTP benutzt. Bei den HTTP-Headers unterscheidet man zwischen Anfrage- und Antwort-Header. Die HTTP-Header gelten als Sticker auf dem HTTP-Envelope, der zwischen Client und Server versendet wird. Die Header gelten als Metadaten und bestehen aus key-value Paaren (vgl. ebd., S. 6).

RESTful Sind alle REST Bedingungen erfüllt spricht man von RESTful. Siehe zur Erklärung Abschnitt 2.3.

Servlet Bezeichnet eine Java Technologie die es ermöglicht Funktionalitäten innerhalb des Web's bereitzustellen. Siehe zur Erklärung Abschnitt 2.6

application state Der Zustand der Applikation der beim Client liegen sollte. Siehe zur Erklärung Abschnitt 2.4.4.

cache Ein *cache* ermöglicht das Zwischenspeichern von Informationen für die Wiederbenutzung (vgl. Fielding 2000, S. 48).

entity body Dieser Begriff wird in dieser Arbeit im Zusammenhang von HTTP benutzt. Entity body steht für das Dokument oder Repräsentation innerhalb des HTTP-Evenlope (vgl. Richardson und Ruby 2007, S. 6).

entity Englisch für Etwas oder einen Gegenstand, wird in dieser Arbeit im Zusammenhang mit der Programmiersprache Java und ORM verwendet. Entity beschreibt in dieser Arbeit eine Klasse oder ein Objekt, das einen Eintrag aus einer Tabelle einer Datenbank repräsentiert. Z.B. die Entity-Klasse User repräsentiert einen Eintrag aus der User-Tabelle.

framework Englisch für Rahmen oder Gerüst. Wird in dieser Arbeit benutzt um ein Softwaregerüst zu bezeichnen das verwendet wird um Software unter bestimmten Aspekten zu entwickeln. Ein framework stellt meist bestimmte Schnittstellen, Architekturmerkmale und Funktionalitäten bereit.

key-value Key englisch für Schlüssel und value englisch für Wert beschreiben ein Schlüssel-Wert Paar. Der Schlüssel ermöglicht es den Wert eindeutig zu identifizieren. Solche key-value Paare werden z.B. im JSON und Form-Encoded Format oder innerhalb eines property files verwendet. Der Schlüssel und der dazugehörige Wert wird meist mit einem Zeichen getrennt. Das Zeichen kann z.B. =, : oder anderes sein.

layered system Ein System das in Schichten eingeteilt ist. Die untersten Schichten stellen Funktionalitäten für die oberen bzw. äußersten Schichten bereit (vgl. Fielding 2000, S. 46).

marshaller Englisch für Einweiser, wird in dieser Arbeit im Zusammenhang mit der Programmiersprache Java und der JAXB Spezifikation verwendet. Der marshaller ermöglicht es die Daten/Informationen aus einem Java-Objekt in ein XML oder JSON Format zu konvertieren. Dieses Vorgehen wird als marshalling bezeichnet.

overhead Englisch für Aufwand oder Mehraufwand, wird in dieser Arbeit zur Beschreibung des Mehraufwands bzw. Mehraufwand von Ressourcenkosten verwendet. Ressourcenkosten können Bandbreite oder Latenz bei der Übertragung von Nachrichten sein.

property file Property englisch für Eigenschaft. Der Begriff property file wird in dieser Arbeit im Zusammenhang mit der Programmiersprache Java verwendet. Das property file hat normalerweise die Endung ”properties”. Innerhalb dieses file werden Eigenschaften bzw. Informationen, als key-value Paare, gespeichert. Diese Informationen können das Verhalten des zugehörigen Java Programms beeinflussen.

resource identifier Englisch für den Ressourcenbezeichner. Der Bezeichner identifiziert die Ressource eindeutig. Siehe zur Erklärung Abschnitt 2.3.4.

resource state Der Zustand der Ressource der beim Server liegen sollte. Siehe zur Erklärung Abschnitt 2.4.4.

uniform interface Englisch für einheitliche Schnittstelle, beschreibt eine der REST- bzw. ROA-Bedingungen. Siehe zur Erklärung Abschnitt 2.4.6.

unmarshaller Dieser Begriff wird in dieser Arbeit im Zusammenhang mit der Programmiersprache Java und der JAXB Spezifikation verwendet. Der unmarshaller ermöglicht die Umkehroperation des marshallings, bezeichnet als das unmarshalling. Er konvertiert die Daten aus dem XML oder JSON Format zurück in ein Java-Objekt.

Literatur

Apache. *Apache CXF Dokumentation*. URL: http://cxf.apache.org/docs/index.html (besucht am 16. 06. 2013).

Barry, Douglas. *Service-Oriented Architecture (SOA) Definition*. URL: http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html (besucht am 14. 06. 2013).

Booth, David u. a. *Web Services Architecture*. URL: http://www.w3.org/TR/ws-arch/#whatis (besucht am 07. 06. 2013).

– *Web Services Architecture - SOA*. URL: http://www.w3.org/TR/ws-arch/#service_oriented_architecture (besucht am 14. 06. 2013).

Bray, Tim u. a. (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: http://www.w3.org/TR/REC-xml/ (besucht am 28. 07. 2013).

Crockford, D. (2006). *JSON - RFC*. URL: http://www.ietf.org/rfc/rfc4627.txt (besucht am 19. 06. 2013).

DataNucleus. *Data Nucleus Dokumentation*. URL: http://www.datanucleus.org/products/datanucleus/index.html (besucht am 25. 07. 2013).

DeMichiel, Linda. *JSR 338: JavaTM Persistence 2.1*. URL: http://jcp.org/en/jsr/detail?id=338 (besucht am 25. 07. 2013).

DeMichiel, Linda und Mike Keith. *JSR 220: Enterprise JavaBeansTM 3.0*. URL: http://jcp.org/en/jsr/detail?id=220 (besucht am 25. 07. 2013).

Elmasri, Ramez und Shamkant Navathe. *Chapter 8 - The Enhanced EntityRelationship (EER) Model*. URL: http://tinman.cs.gsu.edu/~raj/4710/f11/Ch08.pdf (besucht am 26. 06. 2013).

Erl, Thomas (2008). *SOA - Entwurfsprinzipien für serviceorientierte Architektur*. Addison-Wesley Verlag.

Fielding, Roy Thomas (2000). »Architectural Styles and the Design of Network-based Software Architectures«. dissertation. UNIVERSITY OF CALIFORNIA, IRVINE. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Fowler, Martin. *Richardson Maturity Model*. URL: http://martinfowler.com/articles/richardsonMaturityModel.html (besucht am 28. 06. 2013).

Freed, N., Innosoft und N. Borenstein (1996). *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. URL: http://www.ietf.org/rfc/rfc2046.txt (besucht am 27. 07. 2013).

Garcia-Molina, Hector, Jeff Ullman und Jennifer Widom. *Database Systems: The Complete Book - Chapter 1.* URL: http://infolab.stanford.edu/~ullman/fcdb/ch1.pdf (besucht am 22. 07. 2013).

Git. *Git - About.* URL: http://git-scm.com/about (besucht am 22. 07. 2013).

JBoss. *Hibernate Documentation.* URL: http://docs.jboss.org/hibernate/orm/4.2/devguide/en-US/html/ (besucht am 22. 07. 2013).

– *RESTeasy 2.3.6 Final Dokumentation.* URL: http://docs.jboss.org/resteasy/docs/2.3.6.Final/userguide/html_single/index.html (besucht am 16. 06. 2013).

Kemper, Prof. Dr. Alfons und Dr. Andre Eickler (2009). *Datenbanksysteme - Eine Einfuehrung.* Oldenbourg Verlag Muenchen.

Liguori, Robert, Patricia Liguori und Lars Schulten (2008). *Java - kurz und gut.* O'REILLY MEDIA.

Maven, Apache. *Maven in 5 Minutes.* URL: http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html (besucht am 22. 07. 2013).

Murata, M. u. a. (2001). *XML Media Types - RFC.* URL: http://www.rfc-editor.org/rfc/rfc3023.txt (besucht am 28. 07. 2013).

ObjectDB Software Ltd. *Comparison of DataNucleus with MySQL server vs Hibernate with MySQL server.* URL: http://www.jpab.org/DataNucleus/MySQL/server/Hibernate/MySQL/server.html (besucht am 25. 07. 2013).

Oracle. *1.3.1. What is MySQL?* URL: http://dev.mysql.com/doc/refman/4.1/en/what-is-mysql.html (besucht am 22. 07. 2013).

– *1.3.2. The Main Features of MySQL.* URL: http://dev.mysql.com/doc/refman/4.1/en/features.html (besucht am 22. 07. 2013).

– *Chapter 20 - Building RESTful Web Services with JAX-RS.* URL: http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html (besucht am 16. 06. 2013).

– *Java Servlet Technology Overview.* URL: http://www.oracle.com/technetwork/java/overview-137084.html (besucht am 15. 06. 2013).

– *Javadoc - Interface Servlet.* URL: http://docs.oracle.com/javaee/5/api/javax/servlet/Servlet.html (besucht am 15. 06. 2013).

– *Jersey 2.0 Dokumentation.* URL: https://jersey.java.net/documentation/2.0/index.html (besucht am 16. 06. 2013).

– *MySQL - Chapter 1. General Information.* URL: http://dev.mysql.com/doc/refman/4.1/en/introduction.html (besucht am 22. 07. 2013).

– *NetBeans IDE - The Smarter and Faster Way to Code.* URL: https://netbeans.org/features/index.html (besucht am 22. 07. 2013).

– *The @Produces Annotation.* URL: http://docs.oracle.com/cd/E19226-01/820-7627/gipxf/index.html (besucht am 21. 06. 2013).

Pawson, Dave. *Discover Restlet Framework*. URL: `http://restlet.org/discover/faq` (besucht am 16. 06. 2013).

Potociar, Marek und Oracle. *JSR 311: JAX-RS: The JavaTM API for RESTful Web Services*. URL: `http://jcp.org/en/jsr/detail?id=311` (besucht am 16. 06. 2013).

Restlet. *Restlet Dokumentation*. URL: `http://restlet.org/learn/tutorial/0.0` (besucht am 16. 06. 2013).

Richardson, Leonard und Sam Ruby (2007). *RESTful Web Services*. O'REILLY MEDIA.

Rittmeyer, Wolfram. *Anhang II: Servlets - Die Grundlagen*. URL: `http://www.jsptutorial.org/content/appendix_II` (besucht am 15. 06. 2013).

Russell, Craig. *JSR 243: JavaTM Data Objects 2.0 - An Extension to the JDO specification*. URL: `http://jcp.org/en/jsr/detail?id=243` (besucht am 25. 07. 2013).

Thijssen, Joshua. *The RESTful CookBook - What is the Richardson Maturity Model?* URL: `http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/` (besucht am 28. 06. 2013).

TLabs. *Moccha - Homepage*. URL: `http://moccha.org/` (besucht am 21. 06. 2013).

– (2013b). *TLabs - Ueber uns*. URL: `http://www.laboratories.telekom.com/public/Deutsch/ueber_uns` (besucht am 04. 06. 2013).

Unbekannt. *GSON - Library*. URL: `https://code.google.com/p/google-gson/` (besucht am 19. 06. 2013).

Wehmeier, Sally u. a. (2005). *Oxford Advanced Learner's Dictionary of Current English*. OXFORD UNIVERSITY PRESS.

Anhang A

A.1 HTTP-Statuscodes

Die Tabelle A.1 beinhaltet alle in dieser Arbeit verwendeten HTTP-Statuscodes. Sie wurde mithilfe von Richardson und Ruby 2007, S. 373 ff. erstellt.

Statuscode	**Name**	**Erklärung**
200	OK	Zeigt an das die Anfrage erfolgreich bearbeitet wurde.
201	Created	Zeigt an das eine Ressource erfolgreich erstellt wurde.
304	Not Modified	Zeigt an das die Repräsentation der Ressource sich seit der letzten Anfrage nicht geändert hat.
400	Bad Request	Zeigt an das etwas mit der Anfrage des Clients nicht stimmt. Z.B. wenn die Repräsentation nicht sinnvoll oder falsch ist.
401	Unauthorized	Zeigt an das der Anfragende keine nötige Autorisierung besitzt, um solch eine Anfrage zu stellen.
404	Not Found	Zeigt an das die angefragte Ressource nicht existiert.
405	Method Not Allowed	Zeigt an das die Methode mit der die Anfrage gestellt wurde von der Ressource nicht unterstützt wird.
406	Not Acceptable	Zeigt an das die Repräsentation die der Client im Accept-*Header* angegeben hat nicht unterstützt wird.
409	Conflict	Zeigt an das die Anfrage, die Ressource in einen inkonsistenten Zustand versetzen würde oder das dieser Zustand mit einem anderen in Konflikt steht.
415	Unsupported Media Type	Zeigt an das der Client eine Repräsentation im falschen Datenaustauschformat gesendet hat. D.h. der Content-Type-*Header* entspricht nicht dem erwarteten MIME media type des Servers.
500	Internal Server Error	Zeigt an das innerhalb des Servers ein Fehler bzw. eine Exception aufgetreten ist.

Tabelle A.1.: Verwendete HTTP-Statuscodes

A.2 HTTP-Header

Die Tabelle A.2 beinhaltet alle in dieser Arbeit verwendeten HTTP-*Header*. Sie wurde mithilfe von Richardson und Ruby 2007, S. 391 ff. erstellt.

Name	**Art**	**Beschreibung**
Accept	Anfrage	Beinhaltet welches Datenaustauschformat der Client bevorzugt.
Allow	Antwort	Beinhaltet welche HTTP-Methoden die Ressource unterstützt.
Authorization	Anfrage	Beinhaltet Benutzername und Passwort in codierter Form.
Content-Length	Beides	Beinhaltet die Größe des *entity body* in Bytes.
Content-Type	Beides	Beinhaltet den MIME media type des *entity body*.
Etag	Antwort	Beinhaltet eine Art von Repräsentation der Ressource als String. Sobald sich die Repräsentation der Ressource ändert, ändert sich der Etag-Wert. Verwendung für die conditional GET-Methode.
Expires	Antwort	Beinhaltet eine Zeitangabe nach der sich vermutlich die Ressourcenrepräsentation ändert. Ermöglicht es die Antwort des Servers Zwischenzuspeichern.
If-Modified-Since	Anfrage	Beinhaltet die Zeitangabe aus dem Antwort-*Header* Last-Modified. Verwendung für die conditional GET-Methode.
If-None-Match	Anfrage	Beinhaltet die Zeitangabe aus dem Antwort-*Header* Etag. Verwendung für die conditional GET-Methode.
Last-Modified	Antwort	Beinhaltet die Zeit an der die Repräsentation der Ressource das letzte mal verändert wurde. Verwendung für die conditional GET-Methode.
Location	Antwort	Beinhaltet die URI einer Ressource. Wird nach der Erstellung einer Ressource an den Client gesendet. Beinhaltet die URI der neuen Ressource.

Tabelle A.2.: Verwendete HTTP-Header

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum **Christopher Zell**